MarkLogic Server

Search Developer's Guide

MarkLogic 10
May, 2019

Last Revised: 10.0-8, October, 2021

Copyright © 2021 MarkLogic Corporation. All rights reserved.

Table of Contents

Search Developer's Guide

1.0 Developing Search Applications in MarkLogic Server 22
1.1 Overview of Search Features in MarkLogic Server 22
1.1.1 High Performance Full Text Search 22
1.1.2 APIs for Multiple Programming Languages 23
1.1.3 Support for Multiple Query Styles 24
1.1.4 Support for Multiple Query Types 26
1.1.5 Full XPath Search Support in XQuery 28
1.1.6 Lexicon and Range Index-Based APIs 28
1.1.7 Stemming, Wildcard, Spelling, and Much More Functionality 28
1.1.8 Alerting API and Built-Ins 28
1.2 Where to Find Search Information 29

2.0 Search API: Understanding and Using 30
2.1 Understanding the Search API 30
2.1.1 Making the Search API Available to Your Application 31
2.1.2 Simple search:search Example and Response Output 31
2.1.3 Automatic Query Text Parsing and Grammar 32
2.1.4 Constrained Searches and Faceted Navigation 34
2.1.5 Built-In Snippetting 36
2.1.6 Search Term Completion 36
2.1.7 Search Customization Via Options and Extensions 36
2.1.8 Speed and Accuracy 37
2.2 Controlling a Search With Query Options 37
2.3 Search Term Completion Using search:suggest 38
2.3.1 default-suggestion-source Option 38
2.3.2 Choose Suggestions With the suggestion-source Option 39
2.3.3 Use Multiple Query Text Inputs to search:suggest 40
2.3.4 Make Suggestions Based on Cursor Position 41
2.3.5 search:suggest Examples 41
2.4 Creating a Custom Constraint 42
2.4.1 Implementing the parse Function 42
2.4.2 Implementing the start-facet Function 45
2.4.3 Implementing the finish-facet Function 46
2.4.4 Example: Creating a Simple Custom Constraint 47
2.4.5 Example: Creating a Custom Constraint for Structured Queries 48
2.4.6 Example: Creating a Custom Constraint Geospatial Facet 50
2.5 Search Grammar 53
2.6 Returning Lexicon Values With search:values 54

2.6.1 Specifying the Input Lexicons 54
2.6.2 Constraining and Filtering Your Results 55
2.6.3 Example: Using a Query to Constrain Results 56
2.6.4 Example: Filtering with Starting Value, Limit, and Page Length 58
2.6.5 Example: Finding Value Co-Occurrences 60
2.6.6 Additional Interfaces 60
2.7 JSON Support in the Search API 60
2.8 More Search API Examples 62
2.8.1 Buckets Example 62
2.8.2 Computed Buckets Example 64
2.8.3 Sort Order Example 66

3.0 Searching Using String Queries 67
3.1 String Query Overview 67
3.2 The Default String Query Grammar 68
3.2.1 Query Components and Operators 68
3.2.2 Operator Precedence 71
3.2.3 Using Relational Operators on Constraints 72
3.2.4 String Query Examples 73

4.0 Searching Using Structured Queries 74
4.1 Structured Query Overview 74
4.2 Structured Query Concepts 75
4.2.1 Major Query Categories 76
4.2.2 Understanding the Difference Between Term and Word Queries 77
4.2.3 Understanding Containment 77
4.2.4 Text Match Semantics 79
4.2.5 Structured Query Sub-Query Taxonomy 80
4.3 Constructing a Structured Query 81
4.4 Syntax Summary 82
4.5 Examples of Structured Queries 83
4.5.1 Example: Simple Structured Search 83
4.5.2 Example: Structured Search With Constraint References as Text 84
4.5.3 Example: Structured Search With Constraint References 85
4.5.4 Example: Structured Search on Key-Value Metadata Fields 86
4.6 Syntax Reference 87
4.6.1 query 89
4.6.2 term-query 91
4.6.3 and-query 93
4.6.4 or-query 94
4.6.5 and-not-query 96
4.6.6 not-query 98
4.6.7 not-in-query 99
4.6.8 true-query 101
4.6.9 false-query 102

4.6.10 near-query 103
4.6.11 boost-query 105
4.6.12 properties-fragment-query 107
4.6.13 directory-query 110
4.6.14 collection-query 112
4.6.15 container-query 113
4.6.16 document-query 115
4.6.17 document-fragment-query 116
4.6.18 locks-fragment-query 118
4.6.19 range-query 119
4.6.20 value-query 123
4.6.21 word-query 127
4.6.22 geo-elem-query 130
4.6.23 geo-elem-pair-query 134
4.6.24 geo-attr-pair-query 138
4.6.25 geo-path-query 142
4.6.26 geo-json-property-query 146
4.6.27 geo-json-property-pair-query 150
4.6.28 geo-region-path-query 154
4.6.29 range-constraint-query 159
4.6.30 value-constraint-query 162
4.6.31 word-constraint-query 165
4.6.32 collection-constraint-query 167
4.6.33 container-constraint-query 169
4.6.34 element-constraint-query 172
4.6.35 properties-constraint-query 174
4.6.36 custom-constraint-query 175
4.6.37 geospatial-constraint-query 178
4.6.38 geo-region-constraint-query 181
4.6.39 lsqt-query 185
4.6.40 period-compare-query 187
4.6.41 period-range-query 189
4.6.42 operator-state 192

5.0 Searching Using Query By Example 195
5.1 QBE Overview 195
5.1.1 Search Criteria Based on Document Structure 197
5.1.2 Logical Operators 201
5.1.3 Comparison Operators 202
5.1.4 Query by Value or Word 203
5.1.5 Search Result Customization 204
5.1.6 Options for Controlling Search Behavior 204
5.2 Example 204
5.2.1 XML Example 205
5.2.2 JSON Example 206
5.3 Understanding QBE Sub-Query Types 208

5.3.1 Value Query 209
5.3.2 Word Query 210
5.3.3 Range Query 211
5.3.4 Composed Query 213
5.3.5 Container Query 214
5.4 Search Criteria Quick Reference 217
5.4.1 XML Search Criteria Quick Reference 218
5.4.2 JSON Search Criteria Quick Reference 220
5.4.3 Searching Entire Documents 221
5.5 QBE Structural Reference 223
5.5.1 Top Level Structure 224
5.5.2 Query Components 225
5.5.3 Response Components 227
5.5.4 XML-Specific Considerations 228
5.5.5 JSON-Specific Considerations 230
5.6 How Indexing Affects Your Query 234
5.7 Adding Options to a QBE 235
5.7.1 Specifying Options in XML 235
5.7.2 Specifying Options in JSON 235
5.7.3 Option List 236
5.7.4 Using Persistent Query Options 238
5.8 Customizing Search Results 240
5.8.1 When to Include a Response in Your Query 240
5.8.2 Using the snippet Formatter 241
5.8.3 Using the extract Formatter 243
5.8.4 Example: Search Customization 244
5.9 Scoping a Search by Document Type 245
5.10 Converting a QBE to a Combined Query 246
5.11 Validating a QBE 246

6.0 Composing cts:query Expressions 248
6.1 Understanding cts:query 248
6.1.1 cts:query Hierarchy 249
6.1.2 Use to Narrow the Search 251
6.1.3 Understanding cts:element-query 251
6.1.4 Understanding cts:element-word-query 251
6.1.5 Understanding Field Word and Value Query Constructors 252
6.1.6 Understanding the Range Query Constructors 252
6.1.7 Understanding the Reverse Query Constructor 252
6.1.8 Understanding the Geospatial Query Constructors 253
6.1.9 Specifying the Language in a cts:query 253
6.2 Creating a Query From Search Text With cts:parse 253
6.2.1 String Query Overview 254
6.2.2 Grammar Components and Operators 255
6.2.3 Including Options and Weights in Query Text 262
6.2.4 Binding a Tag to a Reference, Field, or Query Generator 264

6.2.5 Customizing Naked Term Handling With Bindings 274
6.2.6 Query Text Parsing Examples 275
6.3 Combining multiple cts:query Expressions 278
6.3.1 Using cts:and-query and cts:or-query 278
6.3.2 Proximity Queries using cts:near-query 279
6.3.3 Using Bounded cts:query Expressions 279
6.3.4 Matching Nothing and Matching Everything 280
6.4 Joining Documents and Properties with cts:properties-query or cts:document-fragment-query 280
6.5 Registering cts:query Expressions to Speed Search Performance 281
6.5.1 Registered Query APIs 281
6.5.2 Must Be Used Unfiltered 282
6.5.3 Registration Does Not Survive System Restart 282
6.5.4 Storing Registered Query IDs 283
6.5.5 Registered Queries and Relevance Calculations 283
6.5.6 Example: Registering and Using a cts:query Expression 283
6.6 Adding Relevance Information to cts:query Expressions: 283
6.7 Serializations of cts:query Constructors 284
6.7.1 Serializing a cts:query as XML 284
6.7.2 Serializing a cts.query as JSON 284
6.7.3 Add Arbitrary Annotations With cts:annotation 285
6.7.4 Constructing a cts:query From XML 285
6.7.5 Constructing a cts.query From a JavaScript Object or JSON String 286
6.8 Example: Creating a cts:query Parser 286

7.0 Creating JavaScript Search Applications 289
7.1 JSearch Introduction 289
7.1.1 JSearch Feature Summary 290
7.1.2 Top Level Function Summary 290
7.1.3 Query Design Pattern 291
7.1.4 How JSearch Relates to Other MarkLogic Search APIs 294
7.1.5 Running the Examples in This Chapter 294
7.2 Scoping Operations by Collection 295
7.3 Searching Documents 295
7.3.1 Document Search Basics 296
7.3.2 Example: Basic Document Search 298
7.4 Creating a cts.query 300
7.4.1 Using byExample to Create a Query 301
7.4.2 Using Query Text to Create a cts.query 306
7.4.3 Using cts.query Constructors 308
7.5 Including Facets in Search Results 308
7.5.1 Introduction to Facets 309
7.5.2 Basic Steps for Generating Facets 310
7.5.3 Example: Generating Facets From JSON Properties 312
7.5.4 Creating a Facet Definition 313
7.5.5 Understanding the Output of Facets 315

7.5.6 Sorting Facet Values with OrderBy 318
7.5.7 Retrieving Facets and Content in a Single Operation 319
7.5.8 Multi-Facet Interactions Using othersWhere 322
7.5.9 Example: Multi-Facet Interactions Using othersWhere 323
7.6 Controlling the Ordering of Results 328
7.6.1 Sorting Document Search Results 328
7.6.2 Sorting Values or Tuples Query Results 330
7.6.3 Sorting Word Lexicon Query Results 330
7.6.4 Sorting Facet Values 331
7.7 Returning a Result Subset 331
7.8 Including Snippets of Matching Content in Search Results 332
7.8.1 Enabling Snippet Generation 333
7.8.2 Configuring the Built-In Snippet Generator 334
7.8.3 Returning Snippets and Documents Together 335
7.8.4 Generating Custom Snippets 336
7.8.5 Standalone Snippet Generation 336
7.9 Extracting Portions of Each Matched Document 337
7.9.1 Extraction Overview 337
7.9.2 How selected Affects Extraction 339
7.9.3 Combining Extraction With Snippeting 340
7.10 Using Options to Control a Query 341
7.11 Transforming Results with Map and Reduce 343
7.11.1 Map and Reduce Overview 343
7.11.2 Configuring the Built-In Mapper 344
7.11.3 Using a Custom Mapper 345
7.11.4 Configuring the Built-In Reducer 346
7.11.5 Using a Custom Reducer 347
7.11.6 Example: Returning Only Documents 348
7.11.7 Example: Using a Custom Mapper for Content Transformation 349
7.11.8 Example: Custom Reducer For Document Search 351
7.11.9 Example: Custom Reducer For Values Query 353
7.12 Querying Lexicons and Range Indexes 354
7.12.1 Querying the Values in a Lexicon or Index 354
7.12.2 Finding Value Co-Occurrences in Lexicons and Indexes 357
7.12.3 Querying Values in a Word Lexicon 359
7.12.4 Computing Aggregates Over Range Indexes 362
7.12.5 Constructing Lexicon and Range Index References 366
7.13 Grouping Values and Facets Into Buckets 367
7.13.1 Bucketing Overview 367
7.13.2 Example: Generating Buckets With makeBuckets 369
7.13.3 Example: Grouping Using Custom Buckets 372
7.14 Preparing to Run the Examples 375
7.14.1 Configuring the Database 375
7.14.2 Loading the Sample Documents 379

8.0 Search Customization Using Query Options 381

8.1 Introduction 381
8.2 Getting the Default Query Options 382
8.3 Checking Query Options for Errors 382
8.4 Constraint Options 382
8.4.1 Value Constraint Example 389
8.4.2 Word Constraint Examples 389
8.4.3 Collection Constraint Example 390
8.4.4 Bucketed Range Constraint Example 391
8.4.5 Exact Match (Unbucketed) Range Constraint Example 393
8.4.6 Geospatial Constraint Example 393
8.5 Operator Options 395
8.6 Return Options 398
8.7 Searchable Expression Option 398
8.8 Fragment Scope Option 399
8.9 Searching Key-Value Metadata Fields 400
8.10 Modifying Your Snippet Results 401
8.10.1 Specifying transform-results Options 401
8.10.2 Specifying Your Own Code in transform-results 403
8.11 Extracting a Portion of Matching Documents 404
8.12 Customizing Search Results with a Decorator 408
8.12.1 Understanding Search Result Decorators 408
8.12.2 Writing a Custom Search Result Decorator 409
8.12.3 Installing a Custom Search Result Decorator 410
8.12.4 Using a Custom Search Result Decorator 410
8.13 Other Search Options 411
8.14 Query Options Examples 411
8.14.1 Example: Values and Tuples Query Options 412
8.14.2 Example: Field Constraint Query Options 415
8.14.3 Example: Collection Constraint Query Options 415
8.14.4 Example: Path Range Index Constraint Query Options 416
8.14.5 Example: Element Attribute Range Constraint Query Options 418
8.14.6 Example: Geospatial Constraint Query Options 420

9.0 Relevance Scores: Understanding and Customizing 422
9.1 Understanding How Scores and Relevance are Calculated 422
9.1.1 log(tf)*idf Calculation 423
9.1.2 log(tf) Calculation 423
9.1.3 Simple Term Match Calculation 424
9.1.4 Random Score Calculation 424
9.1.5 Term Frequency Normalization 424
9.2 How Fragmentation and Index Options Influence Scores 425
9.3 Using Weights to Influence Scores 425
9.4 Proximity Boosting With the distance-weight Option 426
9.4.1 Example of Simple Proximity Boosting 426
9.4.2 Using Proximity Boosting With cts:and-query Semantics 427
9.4.3 Using cts:near-query to Achieve Proximity Boosting 428

9.5 Boosting Relevance Score With a Secondary Query 429
9.6 Including a Range or Geospatial Query in Scoring 430
9.6.1 How a Range Query Contributes to Score 431
9.6.2 Use Cases for Range Query Score Contributions 431
9.6.3 Enabling Range Query Score Contribution 431
9.6.4 Understanding Slope Factor 433
9.6.5 Performance Considerations 435
9.6.6 Range Query Scoring Examples 436
9.7 Interaction of Score and Quality 440
9.8 Using cts:score, cts:confidence, and cts:fitness 440
9.9 Relevance Order in cts:search Versus Document Order in XPath 441
9.10 Exploring Relevance Score Computation 442
9.11 Sample cts:search Expressions 444
9.11.1 Magnify the Score Boost for Documents With Quality 444
9.11.2 Increase the Score for some Terms, Decrease for Others 444

10.0 Browsing With Lexicons 445
10.1 About Lexicons 445
10.2 Creating Lexicons 446
10.3 Word Lexicons 447
10.3.1 Word Lexicon for the Entire Database 447
10.3.2 Element/Element-Attribute Word Lexicons 448
10.3.3 JSON Property Word Lexicons 448
10.3.4 Field Word Lexicons 449
10.4 Element/Element-Attribute/Path Value Lexicons 449
10.5 Field Value Lexicons 450
10.6 Value Co-Occurrences Lexicons 451
10.7 Geospatial Lexicons 453
10.8 Range Lexicons 454
10.9 URI and Collection Lexicons 454
10.10 Performing Lexicon-Based Queries 455
10.10.1Lexicon APIs 455
10.10.2Constraining Lexicon Searches to a cts:query Expression 456
10.10.3Using the Match Lexicon APIs 457
10.10.4Determining the Number of Fragments Containing a Lexicon Term 457

11.0 Using Range Queries in cts:query Expressions 459
11.1 Overview of Range Queries 459
11.1.1 Uses for Range Queries 459
11.1.2 Requirements for Using Range Queries 460
11.1.3 Performance and Coding Advantages of Range Queries 460
11.2 Range Query cts:query Constructors 461
11.3 Examples of Range Queries 461

12.0 Using Aggregate Functions 463

12.1 Introduction to Aggregate Functions 463
12.2 Using Builtin Aggregate Functions 463
12.3 Using Aggregate User-Defined Functions 465

13.0 Highlighting Search Term Matches 468
13.1 Overview of cts:highlight 468
13.1.1 All Matching Terms, Including Stemmed, and Capitalized 468
13.2 General Search and Replace Function 469
13.3 Built-In Variables For cts:highlight 469
13.3.1 Using the $cts:text Variable to Access the Matched Text 470
13.3.2 Using the $cts:node Variable to Access the Context of the Match 470
13.3.3 Using the $cts:queries Variable to Feed Logic Based on the Query 471
13.3.4 Using $cts:start to Capture the String-Length Position 472
13.3.5 Using $cts:action to Stop Highlighting 472
13.4 Using cts:highlight to Create Snippets 472
13.5 cts:walk Versus cts:highlight 473
13.6 Common Usage Notes 473
13.6.1 Input Must Be a Single Node 474
13.6.2 Using xdmp:set Side Effects With cts:highlight 474
13.6.3 No Highlighting with cts:similar-query or cts:element-attribute-*-query 475

14.0 Geospatial Search Applications 476
14.1 Terms and Definitions 477
14.2 Licensing Requirements for Geospatial Features 478
14.3 Geospatial Features Overview 479
14.3.1 Search for Points, Polygons, and Other Regions 479
14.3.2 Geospatial Type System 480
14.3.3 Multiple Coordinate Systems 480
14.3.4 Support for Common Geospatial Representations 481
14.3.5 Flexible Data Layout 481
14.3.6 Support for Single and Double Precision Coordinates 481
14.3.7 Geospatial Computational Utility Functions 482
14.3.8 Geospatial Format Conversion Functions 482
14.3.9 Support in Multiple APIs 482
14.4 Understanding Coordinate Systems 483
14.4.1 Understanding Points 483
14.4.2 Understanding Geodetic Coordinates 483
14.4.3 Understanding Euclidean Coordinates 484
14.4.4 Supported Coordinate Systems 485
14.4.5 The Governing Coordinate System 486
14.4.6 How Precision Affects Geospatial Operations 486
14.5 Understanding MarkLogic Geospatial Region Types 487
14.5.1 Boxes 487
14.5.2 Polygons 489

14.5.3 Complex Polygons 491
14.5.4 Linestrings 492
14.5.5 Circles 492
14.6 Understanding Geospatial Query and Index Types 493
14.6.1 Introduction to Geospatial Query and Index Types 493
14.6.2 Geospatial Query Creation 496
14.6.3 Geospatial Index Creation 496
14.6.4 Geospatial XML Element Point Queries and Indexes 497
14.6.5 Geospatial XML Element Child Point Queries and Indexes 498
14.6.6 Geospatial XML Element Pair Point Queries and Indexes 499
14.6.7 Geospatial XML Attribute Pair Point Queries and Indexes 500
14.6.8 Geospatial Path Point Queries and Indexes 501
14.6.9 Geospatial JSON Property Point Queries and Indexes 503
14.6.10 Geospatial JSON Property Child Point Queries and Indexes 504
14.6.11 Geospatial JSON Property Pair Point Queries and Indexes 505
14.6.12 Geospatial Region Queries and Indexes 506
14.6.13 Geospatial Index Positions 508
14.6.14 Geospatial Lexicons 508
14.6.15 Index Reference Resolution 508
14.7 Searching for Matching Points 509
14.7.1 Point Search Overview 510
14.7.2 Example: Point Query Using XQuery 512
14.7.3 Example: Point Query Using JavaScript 514
14.7.4 Constructing a Point Query in XQuery 518
14.7.5 Constructing a Point Query in JavaScript 519
14.7.6 Constructing a Point Query from Query Text 520
14.7.7 Creating Point Queries with the Client APIs 522
14.7.8 Creating Geospatial Facets 525
14.8 Searching for Matching Regions 528
14.8.1 Region Match Overview 529
14.8.2 Example: Simple Intersection Region Query 530
14.8.3 Example: Using Region Queries in a Composed Query 537
14.8.4 Constructing a Region Query Using a Constructor 538
14.8.5 Constructing a Region Query from Query Text 540
14.8.6 Creating Region Queries Using the Client APIs 542
14.8.7 Example: Using the Envelope Pattern to Encode Regions 548
14.9 Controlling Coordinate System and Precision 549
14.9.1 The Relationship Between Precision and Coordinate System 550
14.9.2 Determining the Best Precision for Your Application 550
14.9.3 How MarkLogic Selects the Governing Coordinate System 551
14.9.4 Probing the Governing Coordinate System Name 554
14.9.5 Specifying the App Server Default Coordinate System 554
14.9.6 Specifying the Per-Module Coordinate System 555
14.9.7 Specifying a Per-Operation Coordinate System and Precision 556
14.9.8 Specifying Coordinate System During Index Creation 557
14.10 Understanding Tolerance 558

14.10.1How Tolerance Affects Geometric Comparisons 558
14.10.2Considerations for Tolerance Selection 560
14.11 Summary of Other Geospatial Operations 560
14.12 Converting To and From Common Geospatial Representations 562
14.12.1Conversion Overview 562
14.12.2WKT and WKB Conversions in XQuery 564
14.12.3WKT and WKB Conversions in JavaScript 565
14.12.4Mapping of WKT and WKB Types to MarkLogic Types 566
14.13 Constructing Geospatial Point and Region Values 567
14.14 Geospatial Query Support in Other APIs 569
14.15 Preparing to Run the Examples 569
14.15.1Overview of the Sample Data 569
14.15.2Configuring the Indexes 573
14.15.3Creating the Input Data Files 577
14.15.4Loading the Sample Data 582

15.0 Entity Extraction and Enrichment 586
15.1 Overview of Entity Extraction and Enrichment 586
15.2 Understanding Dictionary-Based Extraction and Enrichment 588
15.3 Creating an Entity Dictionary 589
15.3.1 Understanding Entity Dictionaries 590
15.3.2 Creating a Dictionary Using Entity Constructors 591
15.3.3 Creating a Dictionary From Text 591
15.3.4 Creating a Dictionary From a SKOS Ontology 592
15.3.5 Persisting or Retrieving an Entity Dictionary 595
15.3.6 Serializing a Dictionary as Text 597
15.4 Dictionary-Based Entity Enrichment 598
15.4.1 API Summary 599
15.4.2 Using entity:enrich or entity.enrich 599
15.4.3 Using cts:entity-highlight or cts.entityHighlight 600
15.4.4 XQuery Example: entity:enrich 601
15.4.5 XQuery Example: cts:entity-highlight 603
15.4.6 JavaScript Example: entity.enrich 605
15.4.7 JavaScript Example: cts.entityHighlight 607
15.5 Dictionary-Based Entity Extraction 609
15.5.1 API Summary 610
15.5.2 Extraction Using entity:extract or entity.extract 611
15.5.3 Extraction Using cts:entity-walk or cts.entityWalk 612
15.5.4 XQuery Example: entity:extract 614
15.5.5 XQuery Example: cts:entity-walk 616
15.5.6 JavaScript Example: entity.extract 618
15.5.7 JavaScript Example: cts.entityWalk 620
15.6 Using an Entity Type Map for Extraction or Enrichment 622
15.6.1 Entity Type Map Basics 622
15.6.2 The Default Entity Type Map 624
15.6.3 Handling Compound Entity Types 626

15.6.4 Filtering Entity Types With a Mapping 626
15.7 Overlapping Entity Match Handling 627
15.7.1 Understanding Entity Overlaps 627
15.7.2 Overlap Handling Options 628
15.7.3 Example: Overlap Handling in entity:extract and entity.extract 628
15.7.4 Example: Overlap Handling in entity:enrich and entity.enrich 629
15.7.5 Interaction with the Walk and Highlight Functions 631
15.8 Entity Identification Using Reverse Query 631
15.9 Entity Enrichment Pipelines 634
15.9.1 Sample Pipelines Using Third-Party Technologies 634
15.9.2 Custom Entity Enrichment Pipelines 634

16.0 Creating Alerting Applications 635
16.1 Overview of Alerting Applications in MarkLogic Server 635
16.2 cts:reverse-query Constructor 636
16.3 XML Serialization of cts:query Constructors 636
16.4 Security Considerations of Alerting Applications 637
16.4.1 Alert Users, Alert Administrators, and Controlling Access 637
16.4.2 Predefined Roles for Alerting Applications 637
16.5 Indexes for Reverse Queries 638
16.6 Alerting API 639
16.6.1 Alerting API Concepts 639
16.6.2 Using the Alerting API 641
16.6.3 Using CPF With an Alerting Application 644
16.7 Alerting Sample Application 646

17.0 Using fn:count vs. xdmp:estimate 647
17.1 fn:count is Accurate, xdmp:estimate is Fast 647
17.2 The xdmp:estimate Built-In Function 647
17.3 Using cts:remainder to Estimate the Size of a Search 648
17.4 When to Use xdmp:estimate 648
17.4.1 When Estimates Are Good Enough 650
17.4.2 When XPaths Meet The Right Criteria 650
17.4.3 When Empirical Tests Demonstrate Correctness 651

18.0 Understanding and Using Stemmed Searches 652
18.1 The Role of Stemming and Tokenization in Search 652
18.2 Stemming in MarkLogic Server 653
18.3 Enabling Stemming 653
18.4 Stemmed Searches Versus Word Searches 654
18.5 Using cts:highlight to Emphasize a Query Match 655
18.6 Using cts:contains to Test for a Stemmed Match 655
18.7 Interaction With Wildcard Searches 656
18.8 Using a User-Defined Stemmer Plugin 656
18.8.1 When to Consider a User-Defined Stemmer 656

18.8.2 StemmerUDF Interface Summary 657
18.8.3 Understanding User-Defined Stemmer Control Flow 658
18.8.4 Implementation Guidelines for User-Defined Stemmers 661
18.8.5 Creating and Deploying a User-Defined Stemmer Plugin 661
18.8.6 Registering a User-Defined Stemmer with MarkLogic 662
18.8.7 Testing a User-Defined Stemmer 663
18.8.8 Error Handling and Logging 663

19.0 Custom Dictionaries for Tokenizing and Stemming 665
19.1 Custom Dictionaries in MarkLogic Server 665
19.2 Custom Dictionary Format 666
19.3 Custom Dictionary Function Summary 667
19.4 Example: Managing a Custom Dictionary in XQuery 668
19.4.1 Install the Dictionary 668
19.4.2 Modify and Update the Dictionary 669
19.4.3 Delete the Dictionary 670
19.5 Example: Managing a Custom Dictionary in JavaScript 670
19.5.1 Install the Dictionary 670
19.5.2 Modify and Update the Dictionary 671
19.5.3 Delete the Dictionary 672
19.6 Example: Exercising a Custom Dictionary 672

20.0 Extracting Metadata and Text From Binary Documents 674
20.1 Metadata and Text Extraction Overview 674
20.2 Usage Examples 674
20.2.1 Microsoft Word 674
20.2.2 File Archives 676
20.2.3 PowerPoint 678
20.3 Supported Binary Formats 679
20.3.1 **Archives 679**
20.3.2 **Databases 680**
20.3.3 Email and Messaging 680
20.3.4 Multimedia 680
20.3.5 Other 680
20.3.6 ....Presentation 680
20.3.7 Raster Image 681
20.3.8 Spreadsheet 681
20.3.9 Text and Markup 681
20.3.10Vector Image 681
20.3.11Word Processing and General Office 682

21.0 Understanding and Using Wildcard Searches 683
21.1 Wildcards in MarkLogic Server 683
21.1.1 Wildcard Characters 683
21.1.2 Rules for Wildcard Searches 683

21.2 Enabling Wildcard Searches 684
21.2.1 Specifying Wildcards in Queries 685
21.2.2 Recommended Wildcard Index Settings 685
21.2.3 Understanding the Wildcard Indexes 686
21.3 Interaction with Other Search Features 687
21.3.1 Wildcarding and Stemming 688
21.3.2 Wildcarding and Punctuation Sensitivity 688

22.0 Collections 693
22.1 The collection() Function 693
22.2 Collections Versus Directories 694
22.3 Defining Collections 695
22.3.1 Implicitly Defining Unprotected Collections 695
22.3.2 Explicitly Defining Protected Collections 696
22.4 Collection Membership 697
22.5 Collections and Security 697
22.5.1 Unprotected Collections 698
22.5.2 Protected Collections 698
22.6 Performance Characteristics 699
22.6.1 Number of Collections to Which a Document Belongs 700
22.6.2 Adding/Removing Existing Documents To/From Collections 700

23.0 Using the Thesaurus Functions 701
23.1 The Thesaurus Module 701
23.2 Function Reference 701
23.3 Thesaurus Schema 702
23.4 Capitalization 702
23.5 Managing Thesaurus Documents 702
23.5.1 Loading Thesaurus Documents in XQuery 703
23.5.2 Loading Thesaurus Documents in JavaScript 703
23.5.3 Lowercasing Terms When Inserting a Thesaurus Document 704
23.5.4 Loading the XML Version of the WordNet Thesaurus 704
23.5.5 Updating a Thesaurus Document 705
23.5.6 Security Considerations With Thesaurus Documents 706
23.5.7 Example Queries Using Thesaurus Management Functions 706
23.6 Expanding Searches Using a Thesaurus in XQuery 712
23.7 Expanding Searches Using a Thesaurus in JavaScript 713

24.0 Using the Spelling Correction Functions 714
24.1 Overview of Spelling Correction 714
24.2 Function Reference 714
24.2.1 The Spelling Built-In Functions 715
24.2.2 The Spelling Dictionary Management Module Functions 715
24.3 Dictionary Documents 716
24.3.1 XML Dictionary Document 716

24.3.2 JSON Dictionary Document 716
24.4 Capitalization 717
24.5 Managing Dictionary Documents 717
24.5.1 Loading Dictionary Documents in XQuery 718
24.5.2 Loading Dictionary Documents in JavaScript 718
24.5.3 Loading one of the Sample XML Dictionaries 718
24.5.4 Updating a Dictionary Document 719
24.5.5 Security Considerations With Dictionary Documents 721
24.6 Testing if a Word is Spelled Correctly 721
24.7 Getting Spelling Suggestions for Incorrectly Spelled Words 722

25.0 Distinctive Terms and cts:similar-query 723
25.1 Understanding cts:similar-query 723
25.2 Finding the Distinctive Terms of a Set of Nodes 723
25.3 Understanding the cts:distinctive-terms Output 724
25.4 Example Design Pattern: Making a Tag Cloud 725

26.0 Training the Classifier 727
26.1 Understanding How Training and Classification Works 727
26.1.1 Training and Classification 727
26.1.2 XML SVM Classifier 727
26.1.3 Hyper-Planes and Thresholds for Classes 728
26.1.4 Training Content for the Classifier 732
26.2 Classifier API 732
26.2.1 XQuery Built-In Functions 732
26.2.2 Data Can Reside Anywhere or Be Constructed 733
26.2.3 API is Extremely Tunable 733
26.2.4 Supports Versus Weights Classifiers 733
26.2.5 Kernels (Mapping Functions) 734
26.2.6 Find Thresholds That Balance Precision and Recall 734
26.3 Leveraging XML With the Classifier 734
26.4 Creating a Training Set 734
26.4.1 Importance of the Training Set 735
26.4.2 Defining Labels for the Training Set 735
26.5 Methodology For Determining Thresholds For Each Class 736
26.6 Example: Training and Running the Classifier 737
26.6.1 Shakespeare's Plays: The Training Set 738
26.6.2 Comedy, Tragedy, History: The Classes 738
26.6.3 Partition the Training Content Set 738
26.6.4 Create Labels on the First Half of the Training Content 739
26.6.5 Run cts:train on the First Half of the Training Content 739
26.6.6 Run cts:classify on the Second Half of the Content Set 740
26.6.7 Use cts:thresholds to Compute the Thresholds on the Second Half 741
26.6.8 Evaluating Your Results, Make Changes, and Run Another Iteration 741
26.6.9 Run the Classifier on Other Content 742

27.0 Results Clustering Using cts:cluster 743
27.1 Understanding cts:cluster 743
27.2 Options to cts:cluster 744
27.2.1 Clustering (cts:cluster) Options 744
27.2.2 Indexing (db:) Options 745
27.3 Understanding the cts:cluster Output 745
27.4 Example that Creates an HTML Report of the Cluster 747

28.0 Language Support in MarkLogic Server 751
28.1 Overview of Language Support in MarkLogic Server 751
28.2 Tokenization and Stemming 752
28.2.1 Language-Specific Tokenization 752
28.2.2 Stemmed Searches in Different Languages 754
28.3 Language Aspects of Loading and Updating Documents 755
28.3.1 Tokenization and Stemming 755
28.3.2 xml:lang Attribute 755
28.3.3 Language-Related Notes About Loading and Updating Documents 757
28.3.4 Protecting JSON Files That Should not be Stemmed 757
28.4 Querying Documents By Languages 758
28.4.1 Tokenization, Stemming, and the xml:lang Attribute 758
28.4.2 Language-Aware Searches 758
28.4.3 Unstemmed Searches 759
28.4.4 Unknown Languages 760
28.5 Supported Languages 761
28.6 Generic Language Support 762
28.7 Stemming and Tokenization Customization 762
28.7.1 Tokenization Customization 763
28.7.2 Stemming Customization 763
28.8 Configuring Tokenization and Stemming Plugins 764
28.8.1 Function Summary for Custom Language Management 765
28.8.2 Customization Using a Built-In Lexer or Stemmer 766
28.8.3 Customization Using a User-Defined Lexer or Stemmer 768
28.8.4 Example: Adding Configuration for a Language 770
28.8.5 Example: Removing Configuration for a Language 773
28.8.6 Example: Resetting Configuration for All Languages 774
28.8.7 Understanding Stemming Delegation 775
28.8.8 Custom Dictionary Security Considerations 776
28.8.9 Built-in Lexer Plugin Reference 778
28.8.10 Built-in Stemmer Plugin Reference 779
28.9 Language Support in JSON 782
28.9.1 Overview 782
28.9.2 API Changes 783
28.9.3 JSON Serialization 784
28.9.4 Upgrade Considerations 784

29.0 Custom Tokenization 785
29.1 Custom Tokenizer Overrides 785
29.1.1 Introduction to Custom Tokenizer Overrides 785
29.1.2 How Character Classification Affects Tokenization 786
29.1.3 Using xdmp:describe to Explore Tokenization 787
29.1.4 Performance Impact of Using Tokenizer Overrides 788
29.1.5 Defining a Custom Tokenizer Override 788
29.1.6 Examples of Custom Tokenizer Overrides 789
29.2 User-Defined Lexer Plugins 795
29.2.1 When to Consider a User-Defined Lexer 796
29.2.2 LexerUDF Interface Summary 796
29.2.3 Understanding User-Defined Lexer Control Flow 797
29.2.4 Implementation Guidelines for User-Defined Lexers 799
29.2.5 Creating and Deploying a User-Defined Lexer Plugin 800
29.2.6 Registering a Custom Tokenizer with MarkLogic 800
29.2.7 Testing a User-Defined Lexer 801
29.2.8 Error Handling and Logging 801

30.0 Encodings and Collations 803
30.1 Character Encoding 803
30.2 Collations 804
30.2.1 Overview of Collations 804
30.2.2 Two Common Collation URIs 805
30.2.3 Collation URI Syntax 805
30.2.4 Backward Compatibility with 3.1 Range Indexes and Lexicons 809
30.2.5 UCA Root Collation 809
30.2.6 How Collation Defaults are Determined 809
30.2.7 Specifying Collations 811
30.3 Collations and Character Sets By Language 811

31.0 Appendix: Query Options Reference 816
31.1 How to Use This Reference 817
31.2 Options Summary 818
31.3 additional-query 820
31.3.1 Syntax Summary 820
31.3.2 Component Description 821
31.3.3 Examples 821
31.3.4 See Also 821
31.4 concurrency-level 821
31.5 constraint 822
31.5.1 Syntax Summary 823
31.5.2 Component Description 824
31.5.3 Examples 825
31.5.4 See Also 825
31.5.5 range 825

31.5.6 value 831
31.5.7 word 835
31.5.8 collection 839
31.5.9 container 842
31.5.10element-query 844
31.5.11properties 846
31.5.12geo-attr-pair 847
31.5.13geo-elem 852
31.5.14geo-elem-pair 856
31.5.15geo-json-property 860
31.5.16geo-json-property-pair 863
31.5.17geo-path 867
31.5.18geo-region-path 871
31.5.19custom 874
31.5.20heatmap 877
31.5.21bucket 880
31.5.22computed-bucket 884
31.5.23path-index 888
31.6 debug 890
31.7 default-suggestion-source 890
31.7.1 Syntax Summary 891
31.7.2 Component Description 892
31.7.3 Examples 893
31.7.4 See Also 894
31.8 extract-document-data 895
31.8.1 Syntax Summary 896
31.8.2 Component Description 897
31.8.3 Examples 898
31.8.4 See Also 898
31.9 forest 898
31.10 fragment-scope 899
31.11 grammar 900
31.11.1Syntax Summary 900
31.11.2Component Description 901
31.11.3Examples 902
31.11.4starter 903
31.11.5joiner 905
31.12 operator 908
31.12.1Syntax Summary 909
31.12.2Component Description 909
31.12.3Examples 911
31.12.4See Also 912
31.13 page-length 912
31.14 quality-weight 913
31.15 result-decorator 913
31.15.1Syntax Summary 914

31.15.2Component Description 914
31.15.3Examples 915
31.16 return-aggregates 915
31.17 return-constraints 916
31.18 return-facets 916
31.19 return-frequencies 917
31.20 return-metrics 917
31.21 return-plan 917
31.22 return-qtext 918
31.23 return-query 918
31.24 return-results 919
31.25 return-similar 919
31.26 return-values 920
31.27 search-option 920
31.28 searchable-expression 921
31.28.1Syntax Summary 922
31.28.2Component Description 922
31.28.3Examples 923
31.28.4See Also 923
31.29 sort-order 923
31.29.1Syntax Summary 924
31.29.2Component Description 925
31.29.3Examples 927
31.30 suggestion-source 928
31.30.1Syntax Summary 928
31.30.2Component Description 929
31.30.3Examples 930
31.30.4See Also 931
31.31 term 931
31.31.1Syntax Summary 933
31.31.2Component Description 933
31.31.3Examples 934
31.32 transform-results 936
31.32.1Syntax Summary 937
31.32.2Component Description 938
31.32.3Examples 939
31.32.4See Also 940
31.33 tuples 941
31.33.1Syntax Summary 941
31.33.2Component Description 942
31.33.3Examples 944
31.33.4See Also 944
31.34 values 945
31.34.1Syntax Summary 946
31.34.2Component Description 946
31.34.3Examples 948

31.34.4See Also 949
31.35 Term Options 950
31.36 Facet Options 950
31.37 Range Options 951
31.38 Geospatial Point Query Options 952
31.39 Geospatial Region Query Options 953
31.40 Suggestion Options 953
31.41 Values Options 954

32.0 Technical Support 956

33.0 Copyright 958

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, and most of them are also available through the REST and Java interfaces. This section provides a brief overview some of the main search features in MarkLogic Server and includes the following parts:

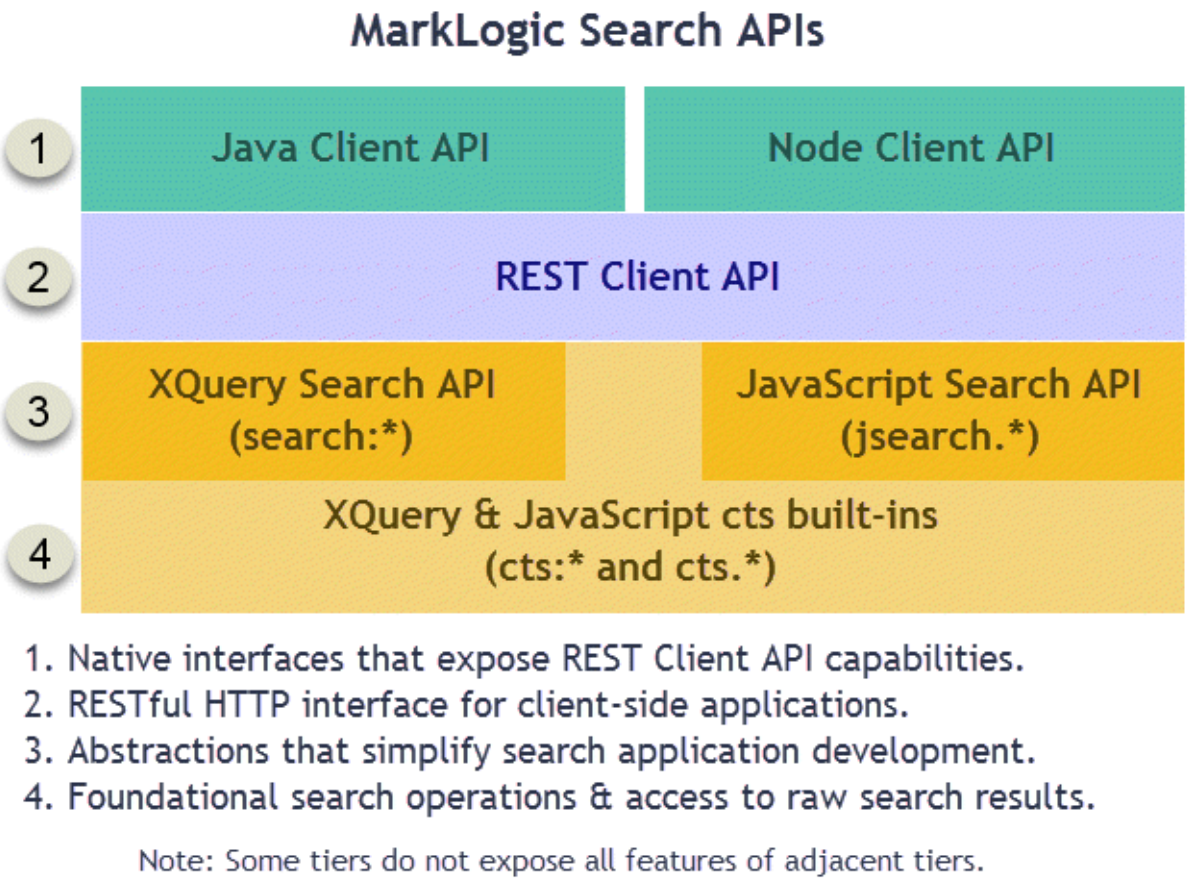
- High Performance Full Text Search
- APIs for Multiple Programming Languages
- Support for Multiple Query Styles
- Support for Multiple Query Types
- Full XPath Search Support in XQuery
- Lexicon and Range Index-Based APIs
- Stemming, Wildcard, Spelling, and Much More Functionality
- Alerting API and Built-Ins

1.1.1 High Performance Full Text Search

MarkLogic Server is designed to scale to extremely large databases (100s of terabytes or more). All search functionality 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. Features such as the `xdmp:estimate` XQuery function and the `unfiltered` search option allow you to return results directly out of the MarkLogic indexes.

1.1.2 APIs for Multiple Programming Languages

MarkLogic Server provides search features through a set of layered APIs that support multiple programming languages. The following diagram illustrates the layering of the MarkLogic search APIs. These APIs are extensible and work in a large number of applications.



The core text search foundation in MarkLogic Server is the cts API, a set of built-in XQuery functions in the `cts` namespace that perform full-text search. These capabilities are also available in Server-Side Javascript as functions with a "cts." prefix.

The APIs above the cts foundation provide a higher level of abstraction that enables rapid development of search applications using XQuery, Server-Side JavaScript, Java, Node.js, or any programming language with support for making HTTP requests. For example, the XQuery Search API leverages functions such as `cts:search`, `cts:word-query`, and `cts:element-value-query` internally.

The Search API, `jsearch` API, and the Client APIs are sufficient for most applications. Use the cts built-ins for advanced application features, such as creating alerting applications with reverse queries or creating content classifiers. The higher level APIs offer benefits such as the following:

- Abstraction of queries from the constraints and indexes that support them.

- Built in support for search result snippeting, highlighting, and performance analysis.
- An extensible simple string query grammar.
- Easy-to-use syntax for query composition.
- Built in best practices that optimize performance.

You can use more than one of these APIs in an application. For example, a Java application can include an XQuery or Server-Side JavaScript extension to perform custom search result transformations on the server. Similarly, an XQuery application can call both `search:*` and `cts:*` functions.

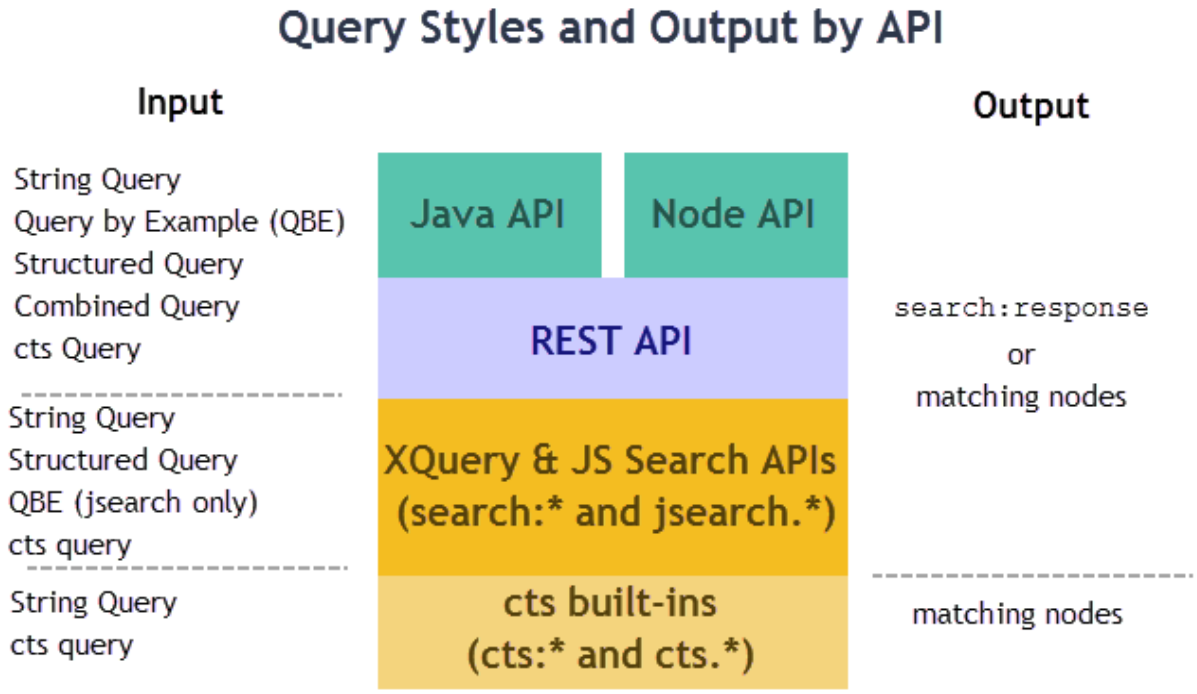
1.1.3 Support for Multiple Query Styles

Each of the APIs described in "APIs for Multiple Programming Languages" on page 23 supports one or more input query styles for searching content and metadata, from simple string queries (`cat OR dog`) to XML or JSON representations of complex queries. Search results are returned in either raw or report form. The supported query styles and result format vary by API.

For example, the primary search function for the CTS API, `cts:search`, accepts input in the form of a `cts:query`, which is a composable query style that enables you to perform fine-grained searches. The `cts:search` function returns raw results as a sequence of matching nodes.

The Search, REST, Node.js and Java APIs accept more abstract query styles such as string and structured queries, and return results either in report form, as an XML `search:response` (or equivalent JSON structure) or matching documents. The customizable `search:response` can include details such as snippets with highlighting of matching terms and query metrics. The REST and Java APIs can also return the results report as JSON.

The following diagram summarizes the query styles and results formats each API provides for searching content and metadata:



The following table provides a brief description of each query style. The level of complexity of query construction increases as you read down the table.

Query Style	Supporting APIs	Description
String Query	all	Construct queries as text strings using a simple grammar of terms, phrases, and operators such as AND and ">". String queries are easily composable by end users typing into a search text box. NOTE: The `cts` and `jsearch` APIs use a slightly different grammar than the higher level APIs. For details, see "Creating a Query From Search Text With cts:parse" on page 253 and "Searching Using String Queries" on page 67.
Query By Example	• REST • Java • Node.js • jsearch	Construct queries in XML or JSON using syntax that resembles your document structure. Conceptually, QBE enables developers to easily search for "documents that look like this". For details, see "Searching Using Query By Example" on page 195.

Query Style	Supporting APIs	Description
Structured Query	• Search • REST • Java • Node.js	Construct queries in JSON or XML using an Abstract Syntax Tree (AST) representation, while still taking advantage of Search API based abstractions and options. Useful for modifying or adding to a query originally expressed as a string query. For details, see "Searching Using Structured Queries" on page 74.
Combined Query	• REST • Java • Node.js	Search using XML or JSON structures that bundle a string, structured, QBE, and/or cts query with Search API query options. This enables searching without pre-defining query options as is otherwise required by the Client APIs. For details, see Specifying Dynamic Query Options with Combined Query in the *REST Application Developer's Guide*, Apply Dynamic Query Options to Document Searches in the *Java Application Developer's Guide*, or Searching with Structured Queries in the *Node.js Application Developer's Guide*.
`cts:query`	• Search • jsearch • cts	Construct queries in XML from low level `cts:query` elements such as `cts:and-query` and `cts:not-query`. This representation is tree structured like Structured Query, but more complicated to work with. For details, see "Composing cts:query Expressions" on page 248. These functions are available in Server-Side JavaScript using the `cts.*` functions such as `cts.andQuery`.

1.1.4 Support for Multiple Query Types

A query encapsulates your search criteria. When you search for documents matching a query, your criteria fall into one or more of the query types described in this section, no matter what query style you use (string, structured, QBE, etc.).

The following query types are basic search building blocks that describe the content you want to match.

- Range: Match values that satisfy a relational expression. You can express conditions such as "less than 5" or "not equal to true". A range query must be backed by a range index.
- Value: Match an entire literal value, such as a string or number, in a specific JSON property or XML element. By default, value queries use exact match semantics. For example, a search for "mark" will not match "Mark Twain".
- Word: Match a word or phrase in a specific JSON property, XML element, or XML attribute. In contrast to a value query, a word query will match a subset of a text value and

does not not use exact match semantics by default. For example, a search for "mark" will match "Mark Twain" in a specific context.

- Term: Match a word or phrase anywhere it appears. In contrast to a value query, a term query will match a subset of a text value and does not use exact match semantics by default. For example, a search for "mark" will match "Mark Twain" anywhere it appears in a document.

Additional query types enable you to build up complex queries by combining the basic content queries with each other and with criteria that add additional constraints. The additional query types fall into the following categories.

- Logical Composers: Express logical relationships between criteria. You can build up compound logical expressions such as "*x* AND (*y* OR *z*)".
- Document Selectors: Select documents based on collection, directory, or URI. For example, you can express criteria such as "*x* only when it occurs in documents in collection *y*".
- Location Qualifiers: Further limit results based on where the match appears. For example, "*x* only when contained in JSON property *z*", or "*x* only when it occurs within *n* words of *y*", or "*x* only when it occurs in a document property".

The CTS API includes query constructors for all the above query types, such as `cts:*-range-query`, `cts:*-value-query`, `cts:*-word-query`, `cts:and-query`, `cts:collection-query`, and `cts:near-query`. For details, see "Composing cts:query Expressions" on page 248.

With no additional configuration, string queries support term queries and logical composers. For example, the query string "cat AND dog" is implicitly two term queries, joined by an "and" logical composer. However, you can easily extend the expressive power of a string query using constraint bindings to enable additional query types. For example, if you use a range constraint binding to tie the identifier "cost" to a specific indexed JSON property, you enable string queries of the form "cost GT 10". For details, see "Searching Using String Queries" on page 67.

In a QBE, content matches are value queries by default. For example, a QBE search criteria of the form `{'my-key': 'desired-value'}` is implicitly a value query for the JSON property `'my-key'` whose value is exactly `'desired-value'`. However, the QBE syntax includes special property names that enable you to construct other types of query. For example, use `$word` to create a word query instead of a value query: `{'my-key': {'$word': 'desired-value'}}`. For details, see "Searching Using Query By Example" on page 195.

Structured query includes components that encompass all the query types, such as value-query, range-query, term-query, and-query, and directory-query. Some of the Client APIs include a structured query builder interface to assist you with structured query composition. For details, see "Searching Using Structured Queries" on page 74.

1.1.5 Full XPath Search Support in XQuery

MarkLogic Server implements the XQuery language, which includes XPath 2.0. XPath expressions are searches which can search XML 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.

MarkLogic Server extends XPath so that you can also use it to address JSON content. For details, see Traversing JSON Documents Using XPath in the *Application Developer's Guide*.

1.1.6 Lexicon and Range Index-Based APIs

MarkLogic Server enables you to define range indexes which index XML structures such as elements, element attributes; XPath expressions; and JSON properties. You can also define range indexes over geospatial values. Each of these range indexes has lexicon APIs associated with them. The lexicon APIs enable you to return values directly from the indexes. Lexicons are very useful in constructing facets and in finding fast counts of XML element, XML attribute, and JSON property values. The Search API and Node.js, Java, and REST Client APIs makes extensive use of the lexicon features. For details about lexicons, see "Browsing With Lexicons" on page 445.

1.1.7 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.

1.1.8 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 635.

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 30
- "Composing cts:query Expressions" on page 248
- "Relevance Scores: Understanding and Customizing" on page 422
- "Browsing With Lexicons" on page 445
- "Using Range Queries in cts:query Expressions" on page 459
- "Highlighting Search Term Matches" on page 468
- "Geospatial Search Applications" on page 476
- "Entity Extraction and Enrichment" on page 586
- "Creating Alerting Applications" on page 635
- "Using fn:count vs. xdmp:estimate" on page 647
- "Understanding and Using Stemmed Searches" on page 652
- "Understanding and Using Wildcard Searches" on page 683
- "Collections" on page 693
- "Using the Thesaurus Functions" on page 701
- "Using the Spelling Correction Functions" on page 714
- "Language Support in MarkLogic Server" on page 751
- "Encodings and Collations" on page 803

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 a Search With Query Options
- Search Term Completion Using search:suggest
- Creating a Custom Constraint
- Search Grammar
- 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, Node.js, and Java, using a variety of query styles, as described in "Support for Multiple Query Styles" on page 24.

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:

- Making the Search API Available to Your Application
- 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 Making the Search API Available to Your Application

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, Node.js, and Java Client APIs; for details, see *REST Application Developer's Guide*, *Node.js Application Developer's Guide*, or *Java Application Developer's Guide*. Server-Side JavaScript applications can access similar features through the JSearch library; for details, see "Creating JavaScript Search Applications" on page 289.

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 namespace 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. You can customize the data returned in each `search:result` using the `result-decorator` query option.

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, Node.js, and REST, through interfaces such as the following:

- XQuery: The `search:search, search:parse`, and `search:resolve` functions
- Java: The `com.marklogic.client.query.QueryManager` class
- Node.js: The `DatabaseClient.documents.query` and `queryBuilder.parsedFrom` functions.
- REST: The `/search` service

The default string query grammar is similar to the Google grammar. The default grammar supports simple terms and double-quoted phrases, logical and relational operators (`AND`, `OR`, `LT`, `GT`), grouping with parentheses (`( )`), negation with a minus sign (`-`), and user-configured constraints with a colon (`:`).

The following is a summary of the default grammar. For details, see “The Default String Query Grammar” on page 68.

- Terms can be free standing:

```
cat
```

- `AND` and `OR` operators, with `AND` having higher precedence.
- Parentheses 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 382 and "Operator Options" on page 395):

```
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`.

String query parsing takes into account constraints and operators specified in an options node at search runtime. For details on the options node for the Search API, see "Controlling a Search With Query Options" on page 37.

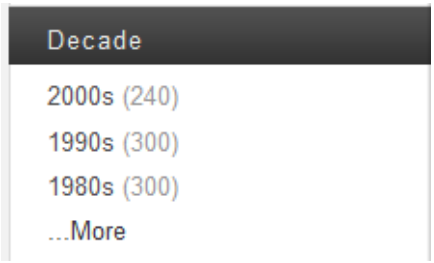
2.1.4 Constrained Searches and Faceted Navigation

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. The default string query grammar makes it easy to express these kinds of searches in a simple query text string. For example, you create a constraint through query options such that the following string query represents a search that constrains matches 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. The Search, REST, Node.js, and Java Client APIs return 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 Client APIs can return this information as XML or JSON; the Node.js Client API returns this information as JSON.

The facets returned by a search 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.

You can generate facets from range, collection, geospatial, and custom constraints. To generate facets from a constraint and include them in your search results, set the `facet` XML attribute or JSON property to true on a constraint definition in your search options. For example:

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

For more details, see “Appendix: Query Options Reference” on page 816.

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 382.

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 and the Node.js, Java, and REST Client APIs include snippets in the `search:response` output, making 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 401.

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 38.

2.1.7 Search Customization Via Options and Extensions

The Search, REST and Java APIs make it easy to customize your searches. A wide range of customizations are available directly through the query 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 XQuery extension mechanism. The mechanism includes hooks 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 local name 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 401.

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, and the Client APIs (Node.js, Java, REST) that build upon it, are 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 query options are designed to be sensible for most application. All applications are different, however, and MarkLogic 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 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 a `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 647.

2.2 Controlling a Search With Query Options

Most search operations in the XQuery Search API and the Client APIs make use of optional query options. Query options enable you to specify the behavior and results format for a search. Default query options are pre-defined. You can override the defaults by supplying custom query options. For example, the XQuery function `search:search` accepts a `search:options` XML node as input.

The REST and Java Client APIs supports query options expressed in either JSON or XML. The Node.js Client API abstracts the representation from your application, but in most cases, this API uses the JSON representation.

For more details, see "Search Customization Using Query Options" on page 381 and "Appendix: Query Options Reference" on page 816.

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.

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

For information on using this feature with the Client APIs, see the following:

- REST: Generating Search Term Completion Suggestions in the *REST Application Developer's Guide*.
- Java: Generating Search Term Completion Suggestions in the *Java Application Developer's Guide*.
- Node.js: Generating Search Term Completion Suggestions in the *Node.js Application Developer's Guide*.

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>
```

For more details, see “default-suggestion-source” on page 890.

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:

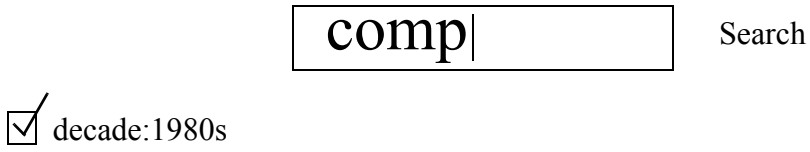
```
<suggestion-source ref="socialsecuritynumber" />
```

For more details, see "suggestion-source" on page 928.

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:

```
search: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 local name 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.

This section covers the following topics:

- Choosing a Parser Interface
- Implementing a String Query parse Function
- Implementing a Structured Query parse Function
- Implementing a Multi-Format parse Function

2.4.1.1 Choosing a Parser Interface

The signature of your constraint parsing function varies depending on the type of query input (string query or structured query) and the API through which you make your queries.

If your constraint can be used in queries initiated from XQuery, such as by calling `cts:search` or `search:search`, choose one of the following solutions:

- If the input is always a string query, see "Implementing a String Query parse Function" on page 43.
- If the input is always a structure query, see "Implementing a Structured Query parse Function" on page 44.
- If the input can be either a string or structured query, see "Implementing a Multi-Format parse Function" on page 44.

If your constraint is only used in queries initiated through the REST, Java, or Node.js Client API and never through XQuery, you can use the structured query parse interface to service both string and structured queries; your query is converted internally as needed. The selections described above for XQuery are also usable with the REST, Node.js and Java Client APIs.

2.4.1.2 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 local name 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 248.

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 50.

2.4.1.3 Implementing a Structured Query parse Function

To use a custom constraint in a structured query, your custom parse function must have a signature compatible with the following:

```
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 local name for the function, but the number and order of the parameters must be compatible and the return type must be compatible. For a full example, see "Example: Creating a Custom Constraint for Structured Queries" on page 48.

The `$query-elem` parameter is `custom-constraint-query` structured query that references your constraint. For details, see "custom-constraint-query" on page 175.

The custom constraint can return either a `cts:query` or the XML serialization of a `cts:query`. MarkLogic recommends that you return a `cts:query`.

2.4.1.4 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 43 and "Implementing a Structured Query parse Function" on page 44.

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 445. 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 local name 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 50.

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 445.

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 local name 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 50.

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 for use with string 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. 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 built-in geospatial contraint types (`geo-attr-pair`, `geo-elem-pair`, and `geo-elem`) as described in "Constraint Options" on page 382.

```
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>
            <yns:regions
xmlns:yns="http://yourcompany.com/yournamespace">
               <yns:region label="A">[0, -180, 30, -90]</yns:region>
               <yns:region label="B">[0, -90, 30, 0]</yns:region>
               <yns:region label="C">[30, -180, 45, -90]</yns:region>
               <yns:region label="D">[30, -90, 45, 0]</yns:region>
               <yns:region label="E">[45, -180, 60, -90]</yns:region>
               <yns:region label="F">[45, -90, 60, 0]</yns:region>
               <yns:region label="G">[45, 90, 60, 180]</yns:region>
               <yns:region label="H">[60, -180, 90, -90]</yns:region>
               <yns:region label="I">[60, -90, 90, 0]</yns:region>
               <yns:region label="J">[60, 90, 90, 180]</yns:region>
            </yns: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 XQuery Search API and the REST, Node.js, and Java Client APIs use a built-in grammar to generate a search query from simple query text, which is typically text entered by an end-user in a simple HTML form. 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`.

Customization of the string query grammar is available using the `grammar` query option.

For details, see "Searching Using String Queries" on page 67

2.6 Returning Lexicon Values With search:values

A lexicon is a list of unique words or values, either throughout an entire database (words only) or over a named element, attribute, or field (words or values). The `search:values` Search API function returns values from lexicons. You can optionally constrain the values with a structured query, choose a subset of the matching values, calculate aggregates based on the lexicon values, and find co-occurrences of values in multiple lexicons.

For general information about lexicons, see "Browsing With Lexicons" on page 445. This section covers the following related topics specific to the Search API.

- Specifying the Input Lexicons
- Constraining and Filtering Your Results
- Example: Using a Query to Constrain Results
- Example: Filtering with Starting Value, Limit, and Page Length
- Example: Finding Value Co-Occurrences
- Additional Interfaces

2.6.1 Specifying the Input Lexicons

The most basic `search:values` call has the following form:

```
search:values($spec-name, $options)
```

Where `$spec-name` is the name of a `values` or `tuples` specification defined in the `search:options` passed as the second parameter. Use a `values` specification to work with the values in a single lexicon. Use a `tuples` specification to work with co-occurrences of values in multiple lexicons.

Before you can query the values or words in an element, attribute, or field, you must define a corresponding range index or a word lexicon using the Admin Interface or Admin API. To you query the URI or collection lexicon, it must be enabled on the database. For details, see "Creating Lexicons" on page 446.

The following example returns all values of the `<first-name/>` element, assuming the existence of an element range index over the element.

```
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">
  <values name="names">
    <range type="xs:string">
      <element ns="" name="first-name" />
    </range>
  </values>
</options>
return
search:values("names", $options)

<values-response name="names" type="xs:string"
    xmlns="http://marklogic.com/appservices/search"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <distinct-value frequency="1">George</distinct-value>
  <distinct-value frequency="1">Fred</distinct-value>
  ...
</values-response>
```

For more examples of values and tuples specifications, see the API reference for `search:values`.

The `search:values` function accepts additional parameters you can use to constrain and filter your results; for details, see "Constraining and Filtering Your Results" on page 55. You can also apply a pre-defined or user-defined aggregate function to values or tuples by defining an aggregate in the search options; for details, see "Using Aggregate Functions" on page 463.

2.6.2 Constraining and Filtering Your Results

The `search:values` function has the following interface. Only the `$spec-name` and `$options` parameters are required.

```
search:values($spec-name, $options, $query,
              $limit, $start, $page-start, $page-length)
```

Use the `$query`, `$limit`, `$start`, `$page-start`, and `$page-length` parameters to filter the results returned by `search:values`, as described in the following table:

Parameter	Description
`$query`	Limit results to values in document that match the provided query. Default: None; return values from all documents.
`$limit`	The maximum number of values to retrieve from the lexicon. Default: No limit; return all values in the lexicon, or all values in the subset selected by `$query`.
`$start`	The first value to return. If this value is not in the lexicon, then values are returned beginning with the next logical value. Default: The first value in the lexicon, or the first value in the subset selected by `$query`.
`$page-start` `$page-length`	Define a subset of the results to return to your application. Default: Return all values selected by `$query`, `$limit`, and `$start`.

The `$query`, `$limit`, and `$start` parameters limit the values selected from the lexicon. The `$page-start` and `$page-length` parameters retrieve a subset of the selected values and can be used to "page through" the selected values in successive invocations.

You cannot use $page-start and $page-length to retrieve values outside the subset selected by `$limit` and/or `$start`. For example, if `$page-start` + `$page-length` exceeds `$limit`, then only (`$limit` - `$page-start` + 1) values are returned.

Most of the filtering parameters can be used independent of one another. That is, you can specify a limit without a query or a start value without a limit. However, if you specify `$page-start`, then you must also specify `$page-length`.

2.6.3 Example: Using a Query to Constrain Results

Imagine a set of documents describing animals. Each document includes an animal name and kind. For example, each document is of the following form:

```
<animal>
  <name>aardvark</name>
  <kind>mammal</kind>
</animal>
```

If an element or field range index is defined on `/animal/name`, then the following query returns a result for all the animal names in the 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">
  <values name="animals">
    <range type="xs:string">
      <field name="animal-name" />
    </range>
  </values>
</options>
return
search:values("animals", $options)

<values-response name="animals" type="xs:string"
    xmlns="http://marklogic.com/appservices/search"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <distinct-value frequency="1">aardvark</distinct-value>
  <distinct-value frequency="1">badger</distinct-value>
  <distinct-value frequency="1">camel</distinct-value>
  <distinct-value frequency="1">duck</distinct-value>
  <distinct-value frequency="1">emu</distinct-value>
  ...
  <distinct-value frequency="1">zebra</distinct-value>
</values-response>
```

The following example adds a query that limits the results to values in documents that match the query “mammal OR marsupial”, eliminating `duck`, `emu` and other “bird” values from the result set. This example uses a structured query derived from a string query by calling `search:parse`, but you can use any structured query.

```
search:values("animals", $options,
    search:parse("mammal OR marsupial", (), "search:query")
)

<values-response name="animals" type="xs:string"
    xmlns="http://marklogic.com/appservices/search"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <distinct-value frequency="1">aardvark</distinct-value>
  <distinct-value frequency="1">badger</distinct-value>
  <distinct-value frequency="1">camel</distinct-value>
  <distinct-value frequency="1">fox</distinct-value>
  <distinct-value frequency="1">hare</distinct-value>
  ...
  <distinct-value frequency="1">zebra</distinct-value>
</values-response>
```

If you include other filtering parameters, such as `$limit`, they are applied after the query. For example, adding a limit of 4 returns the value set [`aardvark badger camel fox`] from the above results.

```
search:values("animals", $options,
    search:parse("mammal OR marsupial", (), "search:query"), 4)
)
```

2.6.4 Example: Filtering with Starting Value, Limit, and Page Length

Assume your lexicon contains a string value for each lower-case letter in the alphabet so that the following query returns results for the values `a,b,c...,z`:

```
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">
  <values name="alphabet">
    <range type="xs:string">
      <field name="letter" />
    </range>
  </values>
</options>
return
search:values("alphabet", $options)
```

The following query supplies a limit of 10, a start value of "c", a page start of 4, and page length of 3 to the above query:

```
search:values("alphabet", $options, (), 10, "c", 4, 3)
  (: $limit       = 10 :)
  (: $start       = "c" :)
  (: $page-start  = 4 :)
  (: $page-length = 3 :)
```

The `$limit` and `$start` parameter values result in a subset of 10 values, beginning with "c", that are retrieved from the lexicon. The example below uses square brackets (`[ ]`) to delimit the selected subset.

```
a b [ c d e f g h i j k l ] m n ... x y z
```

Then, `$page-start` and `$page-length` parameter values define the final “page” of values returned by `search:values`. Since "f" is the 4th value in subset defined by `$limit` and `$start`, the final result subset contains the value f..h. The example below uses curly braces ({ }) to delimit the selected page of values:

```
a b [ c d e { f g h } i j k l ] m n ... x y z
```

Note that `$page-start` and `$page-length` can never yield a result set that extends past the last value in the subset of values defined by `$limit`. Thus, in the example above, no value beyond "l" can be returned without varying `$start` or `$limit`.

The table below illustrates the values returned when applying various combinations of the `$start`, `$limit`, `$page-start`, and `$page-length` parameters and how `search:values` arrives at the final results. As above, square brackets (`[ ]`) delimit the values selected by `$limit` and/or `$start`, and curly braces (`{ }`) delimit the values selected by `$page-start` and `$page-length`.

Filtering Parameters	Returned Values	How the Results Are Derived
`$limit: 5`	`a b c d e`	`[ a b c d e ] f g ... x y z`
`$start: "c"`	`c d e ... z`	`a b [ c d e f g ... x y z]`
`$limit: 5` `$start: "c"`	`c d e f g`	`a b [ c d e f g ] ... x y z`
`$page-start: 1` `$page-length: 3`	`a b c`	`{ a b c } d e f g ... x y z`
`$page-start: 4` `$page-length: 3`	`d e f`	`a b c { d e f } g ... x y z`

`$limit: 5`
`$start: "c"`
`$page-start: 2`
`$page-length: 3`
	`d e f`	`a b [ c { d e f } g ] h ... x y z`

`$limit: 5`
`$page-start:4`
`$page-length: 3`
	`d e`	`[ a b c { d e } ] f g ... x y z`

If a query parameter is included, the above filtering is applied to the results after applying the query.

2.6.5 Example: Finding Value Co-Occurrences

The following shows how to return co-occurrences (tuples) from the URI lexicon and an element, constraint 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"))
```

2.6.6 Additional Interfaces

You can also query lexicons using the following interfaces:

- The `cts:values` XQuery function. For details, see “Browsing With Lexicons” on page 445.
- The REST Client API methods `GET:/v1/values/{name}` and `POST:/v1/values/{name}`. For details, see Querying the Values in a Lexicon or Range Index and Finding Value Co-Occurrences in Lexicons in the *REST Application Developer’s Guide*.
- The Java Client API `ValuesDefinition` interface. For details, see the Javadoc and Search On Tuples (Tuples Query / Values Query) in the *Java Application Developer’s Guide*.
- The Node.js Client API DatabaseClient.values interface. For details, see Querying Lexicons and Range Indexes in the *Node.js Application Developer’s Guide*.

2.7 JSON Support in the Search API

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

```
<!-- Example of enhanced options structures supporting JSON -->
```

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

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

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

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

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

    <constraint name="baz">
        <value>
            <json-property>baz</json-property>
        </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-property>asc</json-property>
            </sort-order>
        </state>
    </operator>
    <sort-order type="xs:int" direction="descending">
        <json-property>desc</json-property>
    </sort-order>

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

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

2.8 More Search API Examples

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

- Buckets Example
- Computed Buckets Example
- Sort Order Example

2.8.1 Buckets Example

The following example 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 382.

Each bucket defines boundary conditions that determines what values fit into the bucket (`@ge`, `@lt`, etc.). Each bucket has a unique name (`@name`) that identifies the bucket search terms. For example, “`decade:1940s`” matches values that fit into the bucket with the name “1990s”.

A bucket can also have a label as the element text data. The label has no functional use in a search, but it is returned in the facet data and can be used by the application for display purposes.

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">Noughts</search:bucket>
      <search:bucket lt="2000" ge="1990"
        name="1990s">Nineties</search:bucket>
      <search:bucket lt="1990" ge="1980"
        name="1980s">Eighties</search:bucket>
      <search:bucket lt="1980" ge="1970"
         name="1970s">Seventies</search:bucket>
      <search:bucket lt="1970" ge="1960"
         name="1960s">Sixties</search:bucket>
      <search:bucket lt="1960" ge="1950"
         name="1950s">Fifties</search:bucket>
      <search:bucket lt="1950" ge="1940"
         name="1940s">Forties</search:bucket>
      <search:bucket lt="1940" ge="1930"
         name="1930s">Thirties</search:bucket>
      <search:bucket lt="1930" ge="1920"
         name="1920s">Twenties</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">Forties</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>
```

2.8.2 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 62.

2.8.3 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 String Queries

This chapter describes how to perform searches using simple string queries with Search API. This chapter includes the following sections:

- String Query Overview
- The Default String Query Grammar

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

3.1 String Query Overview

A *string query* is a plain text search string composed of terms, phrases, and operators that can be easily composed by end users typing into an application search box. For example, “cat AND dog” is a string query for finding documents that contain both the term “cat” and the term “dog”.

For historical reasons, MarkLogic supports two similar string query grammars. The XQuery Search API, and the REST, Java, and Node.js Client APIs support the grammar discussed in this chapter. The XQuery `cts:parse` function, the Javascript `cts.parse` function, and the Javascript `jsearch` API support a similar grammar; for details, see “Creating a Query From Search Text With cts:parse” on page 253. The two grammars share the same basic set of operators, but differ in how you define constraints and the degree of customizability.

The syntax of a string query is determined by a configurable grammar. A powerful default grammar is pre-defined. You can modify or extend the grammar through the `grammar` search option. For details, see “The Default String Query Grammar” on page 68.

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`.

You can use string queries to search contents and metadata with the following MarkLogic Server APIs:

- XQuery Search API. For details, see `search:search` and `search:parse`.
- 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 The Default String Query Grammar

The Search API has a built-in default grammar for interpreting string querys such as "cat AND dog". The default grammar enables you to write applications that perform complex queries against a database based on simple search strings.

- Query Components and Operators
- Operator Precedence
- Using Relational Operators on Constraints
- String Query Examples

3.2.1 Query Components and Operators

Use the following components and operators to form string queries with the default search grammar:

Query	Example	Description
any terms	`dog` `dog cat`	Match one or more terms, as with a `cts:and-query`. Adjacent terms and phrases are implicitly joined with `AND`. For example, `dog cat` is the same as `dog AND cat`.

`" "`	`"dog tail"`
	`"dog tail" "cat whisker"`
	`dog "cat whisker"`
		Terms in double quotes are treated as a phrase. Adjacent terms and phrases are implicitly joined with `AND`. For example, `dog "cat whisker"` matches documents containing both the term `dog` and the phrase `cat whisker`.

Query	Example	Description
( )	`(cat OR dog) zebra`	Parentheses indicate grouping. The example matches documents containing at least one of the terms `cat` or `dog`, and also contain the term `zebra`.

-query	`-dog`
	`-(dog OR cat)`
	`cat -dog`
		A NOT operation, as with a `cts:not-query`. For example, `cat -dog` matches documents that contain the term `cat` but that do not contain the term `dog`.

query1 `AND` *query2*	`dog AND cat` `(cat OR dog) AND zebra`	Match two query expressions, as with a `cts:and-query`. For example, `dog AND cat` matches documents containing both the term `dog` and the term `cat`. `AND` is the default way to combine terms and phrases, so the previous example is equivalent to `dog cat`.
query1 `OR` *query2*	`dog OR cat`	Match either of two queries, as with a `cts:or-query`. The example matches documents containing at least one of either of terms `cat` or `dog`.
query1 `NOT_IN` *query2*	`dog NOT_IN "dog house"`	Match one query when the match does not overlap with another, as with `cts:not-in-query`. The example matches occurrences of `dog` when it is not in the phrase `dog house`.
query1 `NEAR` *query2*	`dog NEAR cat` `(cat food) NEAR mouse`	Find documents containing matches to the queries on either side of the `NEAR` operator when the matches occur within 10 terms of each other, as with a `cts:near-query`. For example, `dog NEAR cat` matches documents containing `dog` within 10 terms of `cat`.

Query	Example	Description
query1 `NEAR/`*N* *query2*	`dog NEAR/2 cat`	Find documents containing matches to the queries on either side of the NEAR operator when the matches occur within *N* terms of each other, as with a `cts:near-query`. The example matches documents where the term `dog` occurs within 2 terms of the term `cat`.

constraint`:`*value*	`color:red`
	`decade:1980s`
	`birthday:1999-12-31`
		Find documents that match the named constraint with given value, as with a `cts:element-range-query` or other range query. For details, see "Using Relational Operators on Constraints" on page 72.

operator`:`*state*	`sort:relevance` `sort:date`	Apply a runtime configuration operator such as sort order, defined by an `operator` XML element or JSON property in the search options. For details, see "Operator Options" on page 395.
constraint `LT` *value*	`color LT red` `birthday LT 1999-12-31`	Find documents that match the named range constraint with a value less than *value*. For details, see "Using Relational Operators on Constraints" on page 72.
constraint `LE` *value*	`color LE red` `birthday LE 1999-12-31`	Find documents that match the named range constraint with a value less than or equal to *value*. For details, see "Using Relational Operators on Constraints" on page 72.
constraint `GT` *value*	`color GT red` `birthday GT 1999-12-31`	Find documents that match the named range constraint with a value greater than *value*. For details, see "Using Relational Operators on Constraints" on page 72.

Query	Example	Description
constraint `GE` *value*	`color GE red` `birthday GE 1999-12-31`	Find documents that match the named range constraint with a value greater than or equal to *value*. For details, see "Using Relational Operators on Constraints" on page 72.
constraint `NE` *value*	`color NE red` `birthday NE 1999-12-31`	Find documents that match the named range constraint with a value that is not equal to *value*. For details, see "Using Relational Operators on Constraints" on page 72.
query1 `BOOST` *query2*	`george BOOST washington`	Find documents that match *query1*. Boost the relevance score of documents that also match *query2*. The example returns all matches for the term "george", with matches in documents that also contain "washington" having a higher relevance score. For more details, see `cts:boost-query`.

3.2.2 Operator Precedence

The precedence of operators in the default grammar, from highest to lowest, is shown in the following table. Each row in the table represents a precedence level. Where multiple operators have the same precedence, evaluation occurs from left to right. Query sub-expressions using operators higher in the table are evaluated before sub-expressions using operators lower in the table.

Operator
`:`, `LT`, `LE`, `GT`, `GE`, `NE`
`-`
`NOT_IN`

Operator
BOOST
( ), NEAR, NEAR/*N*
AND
OR

For example, AND has higher precedence than OR, so the following queries:

```
A AND B OR C
A OR B AND C
```

Evaluate as if written as follows:

```
(A AND B) OR C
A OR (B AND C)
```

3.2.3 Using Relational Operators on Constraints

The relational query operators :, LT, LE, GT, GE, and NE accept a constraint name on the left hand side and a value on the right hand side. That is, queries using these operators are of the following form:

```
constraint op value
```

These relational operators match fragments that meet the named constraint with a value that matches the relationship defined by the operator (equals, less than, greater than, etc.). For example, if your query options define an element word constraint named `color`, then `color:red` matches documents that contain elements meeting the `color` constraint with a value of `red`. For details and more examples, see "Constraint Options" on page 382.

The constraint name must be the name of a `<constraint/>` XML element or `"constraint"` JSON object defined by the query options governing the search. The constraint can be a word, value, range, or geospatial constraint. There must be a range index associated with the constraint.

If the constraint is unbucketed, the value on the right hand side of the operator must be convertible to the type of the constraint. For example, if the range index behind the constraint has type `xs:date`, then the value to match must represent an `xs:date`.

If the constraint is bucketed, then the value must be the name of a bucket defined by the constraint. For example, if searching using the `decade` bucketed constraint defined in "Bucketed Range Constraint Example" on page 391, then the value on the right hand side must be a bucket name such as `1920s` or `2000s`, such as `decade:1920s`.

3.2.4 String Query Examples

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`

4.0 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
- Structured Query Concepts
- Constructing a Structured Query
- Syntax Summary
- Examples of Structured Queries
- Syntax Reference

4.1 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*.

4.2 Structured Query Concepts

The concepts covered in this section should help you understand the purpose and scope of the various structured query building blocks. For detailed information about a particular sub-query, see "Syntax Reference" on page 87.

The following topics are covered:

- Major Query Categories
- Understanding the Difference Between Term and Word Queries
- Understanding Containment
- Text Match Semantics
- Structured Query Sub-Query Taxonomy

4.2.1 Major Query Categories

A query encapsulates your search criteria. You can combine search criteria into complex search expressions using many different types of query combinations. The many sub-query components of structured query fall into one of the categories described in this section.

The query categories in the following table are "leaf" queries that never contain other queries. These type of query are the basic search building blocks that describe what content you want to match.

Query Type	Description
value	Match an entire value in a specific place, such as a phrase or number in a JSON property or XML element. Value queries on JSON property values must use properly typed criteria; if you do not specify a type, string is assumed. For example, to match the value of a property with "number" type, you must explicitly set the criteria value type to "number" in your query. For details, see "value-query" on page 123.
word	Match a word or phrase in a specific place, such as in a specific JSON property, XML element or attribute, or field. A word query will match a subset of a text value. A word query only matches text, so it will never match JSON property values that have number, boolean, or null type. For details, see "word-query" on page 127.
term	Match a word or phrase anywhere it appears in a document or container. A term query will match a subset of a text value. A term query only matches text, so it will never match JSON property values that have number, boolean, or null type. For details, see "term-query" on page 91.
range	Match values that satisfy a relational expression applied to a typed value. You can express conditions such as "less than 5" or "not equal to true". A range query must either be backed by a range index or used in a filtered search operation. For details, see "range-query" on page 119.

Additional sub-queries enable you to combine the basic content queries with each other and with additional criteria and constraints. The additional query types fall into the following general categories.

- Logical Composers: Express logical relationships between criteria. You can build up compound logical expressions such as "*x* AND (*y* OR *z*)".
- Document Selectors: Select documents based on collection, directory, or URI. For example, you can express criteria such as "*x* only when it matches in documents in collection *y*".

- Location Qualifiers: Further limit a criteria based on where the match appears. For example, "*x* only when contained in JSON property *z*", or "*x* only when it matches within *n* words of *y*", or "*x* only when it matches in a document property".

4.2.2 Understanding the Difference Between Term and Word Queries

Term queries and word queries differ primarily in how they handle containment. A term query finds matches anywhere within its context container, while a word query matches only immediate children. For example, suppose your JSON or XML document has the following structure:

JSON	XML

`{ "a": {`
`  "b": "value",`
`  "c": { "d": "value" }`
`} }`
	`<a>`
	`  <b>value</b>`
	`  <c>`
	`  <d>value</d>`
	`  </c>`
	`</a>`

A term query for "value" in "a" finds 2 matches: The occurrence in "b", and the one in "d". However, a word query for "value" in "a" finds no matches because "value" does not occur as an immediate child of "a".

To locate occurrences of "value" using a word query, you must constrain the word query to the scope of "b" or "d". For example, the following sub-queries match "value" in "b" in the JSON and XML documents, respectively:

JSON	XML

`"word-query": {`
`  "json-property": "b",`
`  "text": "value"`
`}`
	`<search:word-query>`
	`  <search:element name="b">`
	`  <search:text>value</search:text>`
	`</search:word-query>`

4.2.3 Understanding Containment

Many sub-query types constrain matches to the context of a particular container. A *container* is a JSON property, XML element, or XML element attribute.

A query such as a word or value query that includes the name of a container only matches occurrences within that container. However, the container can appear at any level within the enclosing document or container. That is, it does not have to be an immediate child.

For example, the following word queries only matches occurrences of "value" when appears in the value of a JSON property or XML element named "a". (The examples use `json-property` in the JSON version and `element` in the XML version, but you can use these specifiers independent of query format.)

JSON	XML

`"word-query": {`
`  "json-property": "b",`
`  "text": "value"`
`}`
	`<search:word-query>`
	`  <search:element name="b">`
	`  <search:text>value</search:text>`
	`</search:word-query>`

However, in the absence of other restrictions, the container named "b" can occur anywhere. For example, the following documents each contain two matches because there are 2 JSON properties (or XML elements) containing "value".

JSON	XML

`{"a": {`
`  "b": "value",`
`  "c": { "b": "value" }`
`} }`
	`<a>`
	`  <b>value</b>`
	`  <c>`
	`  <b>value</b>`
	`  </c>`
	`</a>`

You can wrap a query in a container-query to further limit the scope of the matches. For example, the following sub-queries only match "value" in "b" when "b" occurs inside "c":

JSON	XML

`"container-query": {`
`  "json-property": "c",`
`  "word-query": {`
`  "json-property": "b",`
`  "text": "value"`
`  }`
`}`
	`<search:container-query>`
	`  <search:element name="c" />`
	`  <search:word-query>`
	`  <search:element name="b">`
	`  <search:text>value</search:text>`
	`  </search:word-query>`
	`</search:container-query>`

You can limit the scope of matches in other ways, such as collection, database directory, or property fragment scope. For details, see "Location Qualifiers" on page 81 and "Document Selectors" on page 81.

4.2.4 Text Match Semantics

Whether a value, term, or word query on text content is case-sensitive, diacritic-sensitive, whitespace-sensitive, or punctuation-sensitive depends on database configuration, query options, and the input criteria text. Whether stemming and wildcarding are active similar depends on options and database configuration.

The defaults for text matches are as follows:

- Case: If the criteria text is all lower-case, then the match is case-insenstive. If the criteria contains any upper-case letters, then the match is case sensitive.
- Diacritics: If the criteria text contains no diacritics, then the match is diacritic-insenstive. If the criteria contains any diacritics, then the match is diacritic-sensitive.
- Whitespace: Whitespace insenstive. (Whitespace is still used to tokenize words.)
- Punctuation: If the criteria text contains no punctuation, then the match is punctuation-insenstive. If the criteria contains any punctuation, then the match is punctuation sensitive.
- Stemming: Depends on the database configuration. Stemmed search is disabled on a database by default.
- Wildcarding: Depends on the database configuration and the criteria text. Wildcard searches are disabled on a database by default. If any wildcard search is enabled and the criteria text contains wildcard characters ('?' or '*'), then wildcarding is applied.

For example, a word query for "purple" matches both "purple elephants" and "Purple Elephants", but a word query for "Purple" only matches "Purple Elephants".

Similarly, a word query for "purple elephants" matches both "purple elephants" and "purple, elephants", but a word query for "purple,elephants" will only match "purple, elephants".

You can override some of these behaviors with query options and database configuration. For example, if wildcard searches are not enabled on the database, then a word query for "thom*" will not match "Thomas". Similarly, you can set term options local to a particular word-query or value-query, or more widely through the `term-options` query option.

For more details, see "Term Options" on page 950 and Understanding the Text Index Settings in the *Administrator's Guide*.

4.2.5 Structured Query Sub-Query Taxonomy

Structured query explicitly exposes all the query types described in "Major Query Categories" on page 76. This section is a quick reference for locating the kind of sub-query you need, based on this categorization.

You can use most kinds of sub-query in combination with each other to build up complex queries. For details, see "Syntax Reference" on page 87.

- Basic Content Queries
- Logical Expression Composers
- Location Qualifiers
- Document Selectors

4.2.5.1 Basic Content Queries

Basic content queries express search criteria about your content, such as "JSON property A contains value B" or "any document containing the phrase 'dog'". These queries function as "leaves" in the structure of a complex, compound query because they do not contain sub-queries.

- term-query
- word-query
- value-query
- range-query
- geo-elem-query
- geo-elem-pair-query
- geo-attr-pair-query
- geo-json-property-query
- geo-json-property-pair-query
- geo-path-query

4.2.5.2 Logical Expression Composers

Logical composers are queries that join one or more sub-queries into a logical expression. For example, "documents which match both query1 and query2" or "documents which match either query1 or query2 or query3".

- and-query
- and-not-query

- boost-query
- not-query
- not-in-query
- or-query

4.2.5.3 Location Qualifiers

Location qualifiers limit results based on where subquery matches occur, such as only in content, only in metadata, or only when contained by a specified JSON property or XML element. For example, “matches for this sub-query that occur in metadata” or “matches for this sub-query that are contained in JSON Property P”.

- document-fragment-query
- locks-fragment-query
- near-query
- properties-fragment-query
- container-query

4.2.5.4 Document Selectors

Document selectors are queries that match a group of documents by database attributes such as collection membership, directory, or URI, rather than by contents. For example, “all documents in collections A and B” or “all documents in directory D”.

- collection-query
- directory-query
- document-query

4.3 Constructing a Structured Query

You can construct a structured query in multiple ways:

- Manually, using the syntax described in “Syntax Reference” on page 87.
- 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`. or `com.marklogic.client.pojo.PojoQueryBuilder`, or an equivalent interface.

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>
```

4.4 Syntax Summary

This section gives a brief summary of structure of a structured query. For details, see "Syntax Reference" on page 87

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 property, 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 248.

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*.

4.5 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
- Example: Structured Search on Key-Value Metadata Fields

For additional examples, see "Syntax Reference" on page 87.

4.5.1 Example: Simple Structured Search

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

```
<search:query xmlns:search="http://marklogic.com/appservices/search">
  <search:term-query>
    <search:text>imagine 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": [ "imagine 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 $complex-search :=
  search:parse('"imagine a complex search"',
       (), "search:query")
return
search:resolve($complex-search)

=> a search:response element
```

The REST API and Java API include interfaces for searching directly with structured queries. For details, see Searching With Structured Queries in *REST Application Developer's Guide* and Search Documents Using Structured Query Definition in *Java Application Developer's Guide*.

4.5.2 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" ] } }
        ]
      }
    }]
}}
```

4.5.3 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 62 -->
<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"
            }
          }
        ]
      }
    }]
}}
```

4.5.4 Example: Structured Search on Key-Value Metadata Fields

A metadata field is a field over key-value document metadata. To make key-value metadata searchable, you must define a metadata field on the key, as described in Configuring a New Metadata Field in the *Administrator's Guide*. You might also need to enable field value searches on your database or configure a field range index, depending on the type of query you want to perform.

Once you define a field over a metadata key, you can include that key-value pair in searches using any of the field query capabilities.

For example, the following query matches the word “twain” when it occurs in the value of the metadata key “author”:

```
<word-query>
    <field name="author"/>
    <text>twain</text>
  </word-query>
</query>
```

The following is the equivalent query expressed as JSON:

```
{ "query": {
    "queries": [{
      "word-query": {
        "field": {"name": "mkey"},
        "text": "flubber"
      }
    }]
}}
```

4.6 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
- true-query
- false-query
- near-query
- boost-query
- properties-fragment-query
- directory-query
- collection-query
- container-query

- document-query
- document-fragment-query
- locks-fragment-query
- range-query
- value-query
- word-query
- geo-elem-query
- geo-elem-pair-query
- geo-attr-pair-query
- geo-path-query
- geo-json-property-query
- geo-json-property-pair-query
- geo-region-path-query
- range-constraint-query
- value-constraint-query
- word-constraint-query
- collection-constraint-query
- container-constraint-query
- element-constraint-query
- properties-constraint-query
- custom-constraint-query
- geospatial-constraint-query
- geo-region-constraint-query
- lsqt-query
- period-compare-query
- period-range-query
- operator-state

4.6.1 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 />`
`  <boost-query />`
`  <!-- cts:query scoping ctors -->`
`  <properties-fragment-query />`
`  <directory-query />`
`  <collection-query />`
`  <container-query />`
`  <document-query />`
`  <document-fragment-query />`
`  <locks-fragment-query />`
`  <range-query />`
`  <value-query />`
`  <word-query />`
`  <geo-elem-query />`
`  <geo-elem-pair-query />`
`  <geo-attr-pair-query />`
`  <geo-path-query />`
`  <geo-json-property-query />`
`  <geo-json-property-pair-query />`
`  <lsqt-query />`
`  <period-compare-query />`
`  <period-range-query />`
`  <!-- Search API abstractions -->`
`  <range-constraint-query />`
`  <value-constraint-query />`
`  <word-constraint-query />`
`  <collection-constraint-query />`
`  <container-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,`
	`  boost-query,`
	
	`  properties-fragment-query,`
	`  directory-query,`
	`  collection-query,`
	`  container-query,`
	`  document-query,`
	`  document-fragment-query,`
	`  locks-fragment-query,`
	`  range-query,`
	`  value-query,`
	`  word-query,`
	`  geo-elem-query,`
	`  geo-elem-pair-query,`
	`  geo-attr-pair-query,`
	`  geo-path-query,`
	`  geo-json-property-query,`
	`  geo-json-property-pair-query,`
	`  lsqt-query,`
	`  period-compare-query,`
	`  period-range-query,`
	
	`  range-constraint-query,`
	`  value-constraint-query,`
	`  word-constraint-query,`
	`  collection-constraint-query,`
	`  container-constraint-query,`
	`  element-constraint-query,`
	`  properties-constraint-query,`
	`  custom-constraint-query,`
	`  geospatial-constraint-query,`
	`  operator-state`
	`  ]`
	`  }`
	`}`

4.6.2 term-query

A query that matches one or more search terms or phrases. By default, a `term-query` is equivalent to `cts:word-query`. However, if you use the `term` query option to customize query term handling, this equivalence may not hold. For example, if your search includes a `term` option that specifies a field word constraint, then a `term-query` might be handled as a `cts:field-word-query`. For details, see "term" on page 931.

- Syntax Summary
- Component Description
- Examples

4.6.2.1 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-option>`*option*`</term-option>`
`</term-query>`
	`"term-query": {`
	`  "text": [ "`*term-or-phrase*`" ],`
	`  "weight": "`*value*`",`
	`  "term-option": [ `*option*` ]`
	`}`

4.6.2.2 Component Description

Element or JSON Property Name	Req'd?	Description
`text`	Y	The term or phrase to search for. The query can contain multiple `text` children. When there are multiple terms, the query matches if any of the terms match.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.
`term-option`	N	Term options to apply to the query. You can specify multiple term options. If the option has a value, the value of `term-option` is *`option=value`*. For example: `<term-option>min-occurs=1</term-option>` in XML, or `"term-option": ["min-occurs=1"]` in JSON. For details, see the `cts` query corresponding to the query constraint type: `cts:word-query`, `cts:element-word-query`, `cts:element-attribute-word-query`, `cts:field-word-query`, or `cts:json-property-word-query`; and "Term Options" on page 950.

4.6.2.3 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" ]`
	`  }`
	`  }]`
	`}}`

4.6.3 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

4.6.3.1 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*`
	`  ]`
	`}`

4.6.3.2 Component Description

Element or JSON Property Name	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.

4.6.3.3 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" }`
	`  ]`
	`  }`
	`  }]`
	`}}`

4.6.4 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

4.6.4.1 Syntax Summary

XML	JSON

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

4.6.4.2 Component Description

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

4.6.4.3 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" ]`
	`  }`
	`  }`
	`  ]}`
	`  }]`
	`}}`

4.6.5 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

4.6.5.1 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* }`
	`}`

4.6.5.2 Component Description

Element or JSON Property Name	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.

4.6.5.3 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" ]`
	`  }}`
	`  }`
	`  }]`
	`}}`

4.6.6 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

4.6.6.1 Syntax Summary

XML	JSON

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

4.6.6.2 Component Description

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

4.6.6.3 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" ]`
	`  }`
	`  }`
	`  }]`
	`}}`

4.6.7 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

4.6.7.1 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* }`
	`}`

4.6.7.2 Component Description

Element or JSON Property Name	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.

4.6.7.3 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"`
	`  ]`
	`  }}`
	`  }`
	`  }]`
	`}}`

4.6.8 true-query

A query that matches all documents (or fragments). For details, see `cts:true-query`.

- Syntax Summary
- Component Description
- Examples

4.6.8.1 Syntax Summary

XML	JSON
All elements are in the namespace `http://marklogic.com/appservices/search`. `<true-query/>`	`"true-query": null`

4.6.8.2 Component Description

This query has no sub-components.

4.6.8.3 Examples

The following example matches all documents.

XML	JSON

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

4.6.9 false-query

A query that matches no documents (or fragments). For details, see `cts:false-query`.

- Syntax Summary
- Component Description
- Examples

4.6.9.1 Syntax Summary

XML	JSON
All elements are in the namespace `http://marklogic.com/appservices/search`. `<false-query/>`	`"false-query": null`

4.6.9.2 Component Description

This query has no sub-components.

4.6.9.3 Examples

The following example matches all documents.

XML	JSON

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

4.6.10 near-query

A query that returns results matching all of the specified queries where the matches occur within a specified distance of each other. For details, see `cts:near-query`.

- Syntax Summary
- Component Description
- Examples

4.6.10.1 Syntax Summary

XML	JSON

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

4.6.10.2 Component Description

Note that `distance` and `minimum-distance` apply to each `near-query` match. Therefore, if `minimum-distance` is greater than `distance`, there can be no matches.

Element or JSON Property Name	Req'd?	Description
`anyQueryType`	Y	One or more queries that must match within the specified proxmity to each other.
`distance`	N	A maximum distance, in number of words, between any two matching queries. Default: 10. The results match if two queries match and the minimum distance between the two matches is less than or equal to the specified distance. A distance of 0 matches when the text is the same text or when there is overlapping text. A negative distance is treated as 0.
`minimum-distance`	N	The minimum distance, in words, between any two matching queries. Default: 0. The results match if the two queries match and the minimum distance between the two matches is greater than or equal to the specified minimum distance. A negative distance is treated as 0.
`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.

4.6.10.3 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" }`
	`  ]`
	`  }`
	`  }]`
	`}}`

4.6.11 boost-query

Find all matches to a query. Boost the search relevance score of results that also match the boosting query. For details, see `cts:boost-query` and "Boosting Relevance Score With a Secondary Query" on page 429.

- Syntax Summary
- Component Description
- Examples

4.6.11.1 Syntax Summary

XML	JSON

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

4.6.11.2 Component Description

Element or JSON Property Name	Req'd?	Description
`matching-query`	N	The query to match. All search results matching this query are returned (modulo limitations imposed by search options). This element can occur multiple times; multiple occurrences are AND'd together. If there are no occurrences, the boosting query implicitly matches all documents.
`boosting-query`	N	The query to use for relevance score boosting. Those results which match both `matching-query` and `boosting-query` have their relevance scores modified proportional to the weight of `boosting-query`. The `boosting-query` is not evaluated if there are no matches to `matching-query`. This element can occur multiple times; multiple occurrences are AND'd together. If there are no occurrences, the boosting query implicitly matches all documents.

4.6.11.3 Examples

The following example searches for documents containing the term “dog”. Documents that also contain the term “cat” will return a search:result with a relevance score boosted proportional to the weight 10.0, giving them higher scores than documents that do not contain “cat”.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <boost-query>`
`  <matching-query>`
`  <term-query>`
`  <text>dog</text>`
`  </term-query>`
`  </matching-query>`
`  <boosting-query>`
`  <term-query>`
`  <text>cat</text>`
`  <weight>10.0</weight>`
`  </term-query>`
`  </boosting-query>`
`  </boost-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "boost-query": {`
	`  "matching-query": {`
	`  "term-query": {`
	`  "text": [ "dog" ]`
	`  }},`
	`  "boosting-query": {`
	`  "term-query": {`
	`  "text": [ "cat" ],`
	`  "weight": "10.0"`
	`  }}`
	`  }`
	`  }]`
	`}}`

4.6.12 properties-fragment-query

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

- Syntax Summary
- Component Description
- Examples

4.6.12.1 Syntax Summary

XML	JSON

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

4.6.12.2 Component Description

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

4.6.12.3 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-fragment-query>
	  <range-constraint-query>
	  <constraint-name>modified</constraint-name>
	  <value>2012-12-31</value>
	  <range-operator>GT</range-operator>
	  </range-constraint-query>
	  </properties-fragment-query>
	</query>

Format	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-fragment-query": {
	  "range-constraint-query": {
	  "value": ["2012-12-31"],
	  "constraint-name": "modified",
	  "range-operator": "GT"
	  }
	  }
	  }]
	}}

4.6.13 directory-query

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

- Syntax Summary
- Component Description
- Examples

4.6.13.1 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`*
	`}`

4.6.13.2 Component Description

Element or JSON Property Name	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.

4.6.13.3 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`
	`  }`
	`  }]`
	`}}`

4.6.14 collection-query

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

- Syntax Summary
- Component Description
- Examples

4.6.14.1 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`*` ]`
	`}`

4.6.14.2 Component Description

Element or JSON Property Name	Req'd?	Description
`uri`	Y	One or more collection URIs. A document matches if it is in any one of the collections specified by `uri`.

4.6.14.3 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"]`
	`  }`
	`  }]`
	`}}`

4.6.15 container-query

A query matching documents containing a specified XML element or JSON property whose contents match a specified sub-query.

- Syntax Summary
- Component Description
- Examples

4.6.15.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<container-query>`
`  <element name=`*`string`* `ns=`*`string`* `/>`
`  <json-property>`*`name`*`</json-property>`
`  <fragment-scope>`*`scope`*`</fragment-scope>`
`  `*`anyQueryType`*
`</container-query>`
	`"container-query": {`
	`  "element": {`
	`  "name": `*`elem-name`*`,`
	`  "ns": `*`namespace`*
	`  },`
	`  "json-property": `*`prop-name`*`,`
	`  "fragment-scope": `*`scope`*`,`
	`  `*`anyQueryType`*
	`}`

4.6.15.2 Component Description

Your query must include exactly one of `element` or `json-property`.

Element or JSON Property Name	Req'd?	Description
`element`	Y	An XML element descriptor, identified by `name` and namespace (`ns`). Both `name` and `ns` are required. If you include `element`, you should not include a `json-property`. If you specify multiple elements, the query matches documents that satisfy any one of the element constraints.
`json-property`	Y	A JSON property name. If you include `json-property`, you should not include an `element`. If you specify multiple properties, the query matches documents that satisfy any one of the property constraints.
`fragment-scope`	N	Constrain matches to the specified fragment scope. Allowed values: `documents` (default) or `properties`. For more details, see the fragment-scope query option.
`anyQueryType`	Y	A sub-query to run against the contents of matching containers (XML elements or JSON properties). An `and-query` will only match array items in the container if all the `and-query` criteria are met in the same array value.

4.6.15.3 Examples

The following XML example matches all documents containing a `<pets/>` element with descendants whose contents match the term “dog”. The JSON example matches all documents containing the property named “pets” with descendants whose contents match the term “dog”.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <container-query>`
`  <element name="pet" ns="" />`
`  <term-query>`
`  <text>dog</text>`
`  </term-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "container-query": {`
	`  "json-property": "pet",`
	`  "term-query": {`
	`  "text": [ "dog" ]`
	`  }`
	`  }`
	`  }]`
	`}}`

4.6.16 document-query

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

- Syntax Summary
- Component Description
- Examples

4.6.16.1 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`*` ]`
	`}`

4.6.16.2 Component Description

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

4.6.16.3 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"`
	`  ]`
	`  }`
	`  }]`
	`}}`

4.6.17 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

4.6.17.1 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`
	`}`

4.6.17.2 Component Description

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

4.6.17.3 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" ]`
	`  }`
	`  }`
	`  }]`
	`}}`

4.6.18 locks-fragment-query

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

- Syntax Summary
- Component Description
- Examples

4.6.18.1 Syntax Summary

XML	JSON

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

4.6.18.2 Component Description

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

4.6.18.3 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-fragment-query>`
`  <container-constraint-query>`
`  <constraint-name>`
`  lock-type`
`  </constraint-name>`
`  <term-query>`
`  <text>write</text>`
`  </term-query>`
`  </container-constraint-query>`
`  </locks-fragment-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "locks-fragment-query": {`
	`  "container-constraint-query": {`
	`  "constraint-name": "lock-type",`
	`  "term-query": {`
	`  "text": [ "write" ]`
	`  }`
	`  }}`
	`  }]`
	`}}`

4.6.19 range-query

A query that applies a range constraint and compares the results to the specified value. For details, see “Constraint Options” on page 382 and the XQuery functions `cts:element-range-query`, `cts:element-attribute-range-query`, `cts:field-range-query`, and `cts:path-range-query`.

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

- Syntax Summary
- Component Description
- Examples

4.6.19.1 Syntax Summary

<table>
<tr><th>XML</th><th>JSON</th></tr>
<tr><td>All elements are in the namespace
http://marklogic.com/appservices/search.

<range-query type=<i>index-type</i> collation=<i>uri</i>>
  <element name=<i>elem-name</i> ns=<i>namespace</i> />
  <attribute name=<i>attr-name</i> ns=<i>namespace</i> />
  <json-property><i>name</i></json-property>
  <field name=<i>field-name</i> collation=<i>uri</i> />
  <path-index><i>path-expr</i></path-index>
  <fragment-scope><i>scope</i></fragment-scope>
  <value><i>value</i></value>
  <range-operator><i>operator</i></operator>
  <range-option><i>option</i></range-option>
  <weight><i>value</i></weight>
</range-query></td><td>"range-query": {
  "type": <i>index-type</i>,
  "collation": <i>index-collation-uri</i>,
  "element": {
  "name": <i>elem-name</i>,
  "ns": <i>namespace</i>
  },
  "attribute": {
  "name": <i>attr-name</i>,
  "ns": <i>namespace</i>
  },
  "json-property": <i>prop-name</i>,
  "field": {
  "name": <i>field-name</i>,
  "collation": <i>uri</i>
  },
  "path-index": {
  "text": <i>path-expr</i>,
  "namespaces": {
  <i>prefix</i>: <i>namespace-uri</i>
  }
  },
  "fragment-scope": <i>scope</i>,
  "value": <i>value-as-string</i>,
  "range-operator": <i>operator</i>,
  "range-option": <i>option</i>,
  "weight": <i>number</i>
}</td></tr>
</table>

4.6.19.2 Component Description

You must specify an `element`, `json-property`, `field`, or `path-index`, or an `element` and an `attribute` to define the range constraint to apply to the query. These components are mutually exclusive: Except for `element` and `attribute`, a query must include exactly one.

Element or JSON Property Name	Req'd?	Description
`element`	N	An XML element descriptor, identified by element `name` and namespace (`ns`). Both `name` and `ns` are required. If you include `element`, you should not include a `json-property`, `field`, or `path-index`.
`attribute`	N	An XML attribute descriptor, identifying the attribute by `name` and namespace (`ns`). Both `name` and `ns` are required.
`json-property`	N	A JSON property name. If you include `json-property`, you should not include an `element`, `field`, or `path-index`.
`field`	N	A field descriptor, identified by the field `name` (required) and optional `collation`. The database configuration must include a definition for this field. If you include `field`, you should not include an `element`, `json-property`, or `path-index`.
`path-index`	N	A path range expression. If the path expression includes namespace prefixes, you must define the namespace bindings on the `path-index`. If you include `path-index`, you should not include an `element`, `json-property`, or `field`. The database configuration must include a matching path range index. The path expression and namespace URIs must match the index configuration; namespace prefixes do not have to match. The path expression is limited to the subset of XPath that can be used to define a path range index. For details, see Path Field and Path-Based Range Index Configuration in the *XQuery and XSLT Reference Guide*.
`fragment-scope`	N	Constrain matches to the specified fragment scope. Allowed values: `documents` (default) or `properties`. For more details, see the `fragment-scope` query option.

Element or JSON Property Name	Req'd?	Description
`value`	N	The value against which to match XML elements, XML element attributes, JSON properties, 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 relationship that must be satisfied between constraint matches and `value`.
`range-option`	N	One or more range query options. Allowed values depend on the type of range query (element, path, field, etc.). For details, see Including a Range or Geospatial Query in Scoring in *Search Developer's Guide*. For a list of options, see "Range Options" on page 951.
`type`	N	The type of the range index. Required when you have multiple indexes over the same item with different datatypes.
`collation`	N	A collation URI to use if the index type is string.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.

4.6.19.3 Examples

The following example matches documents containing a `<body-color/>` element with a value of `black`, assuming an element range index exists on `<body-color/>`.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <range-query type="xs:string">`
`  <element ns="" name="body-color"/>`
`  <value>black</value>`
`  </range-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "range-query": {`
	`  "type": "xs:string",`
	`  "element": {`
	`  "ns": "",`
	`  "name": "body-color"`
	`  },`
	`  "value": [ "black" ]`
	`  }`
	`  }]`
	`}}`

4.6.20 value-query

A query that matches fragments where the value of an XML element, XML attribute, JSON property, or field matches value in the `text` XML element or JSON property of the query.

When searching JSON documents, the criteria value in the query `text` element or JSON property must be properly typed by setting the `type` element or property in the query. If you do not set `type` in the query, "string" is assumed, which will never match a number, boolean, or null value in a JSON document.

The match semantics depend on the `text` value, the database configuration, and the options in effect. For details, see"Text Match Semantics" on page 79 or `cts:element-value-query`, `cts:element-attribute-value-query`, `cts:json-property-value-query`, or `cts:field-value-query`.

- Syntax Summary
- Component Description
- Examples

4.6.20.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<value-query type=node-type>`
`  <element name=elem-name ns=namespace />`
`  <attribute name=attr-name ns=namespace />`
`  <json-property>name</json-property>`
`  <field name=field-name collation=uri />`
`  <fragment-scope>scope</fragment-scope>`
`  <text>name</text>`
`  <term-option>option/term-option>`
`  <weight>value</weight>`
`</value-query>`
	`"value-query": {`
	`  "type": node-type,`
	`  "element": {`
	`  "name": elem-name,`
	`  "ns": namespace`
	`  },`
	`  "attribute": {`
	`  "name": attr-name,`
	`  "ns": namespace`
	`  },`
	`  "json-property": name,`
	`  "field": {`
	`  "name": field-name,`
	`  "collation": uri`
	`  },`
	`  "fragment-scope": scope,`
	`  "text": [ name ],`
	`  "term-option": [ option ],`
	`  "weight": number`
	`}`

4.6.20.2 Component Description

You must specify an `element`, `json-property`, `field`, or an `element` and an `attribute` to define the range constraint to apply to the query. These components are mutually exclusive: Except for `element` and `attribute`, a query must include exactly one.

Element, Attribute, or JSON Property Name	Req'd?	Description
`type`	N	A JSON node type, one of `string` (default), `boolean`, `null`, `number`. Only meaningful for JSON content. Use `type` to constrain the matches to values in this node type. Non-JSON documents never contain boolean, null or number nodes.
`element`	N	An XML element descriptor, identified by element `name` and namespace (`ns`). Both `name` and `ns` are required. If you include `element`, you should not include a `json-property`.

Element, Attribute, or JSON Property Name	Req'd?	Description
`attribute`	N	An XML attribute descriptor, identifying the attribute by `name` and namespace (`ns`). Both `name` and `ns` are required.
`json-property`	N	A JSON property name. If you include `json-property`, you should not include an `element`.
`field`	N	A field descriptor, identified by the field `name` (required) and optional `collation`. The database configuration must include a definition for this field.
`fragment-scope`	N	Constrain matches to the specified fragment scope. Allowed values: `documents` (default) or `properties`. For more details, see the `fragment-scope` query option.
`text`	N	The value that must match in an XML element, XML element attribute, JSON property, or field value. Multiple values can be specified. If there is no `type` specifier, the values are treated as strings for matching purposes. The interpretation of the value is determined by the `type` setting in the query.
`term-option`	N	Term options to apply to the query when matching text. You can specify multiple term options. If the option has a value, the value of term-option is option=value. For example: `<term-option>min-occurs=1</term-option>`. For details, see the `cts` query corresponding to the query constraint type: `cts:element-value-query`, `cts:element-attribute-value-query`, `cts:json-property-value-query`, or `cts:field-value-query`; and "Term Options" on page 950.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.

4.6.20.3 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.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <value-query>`
`  <field name="myFieldName" />`
`  <text>Jane Doe</text>`
`  </value-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "value-query": {`
	`  "field": {`
	`  "name": "myFieldName",`
	`  },`
	`  "text": [ "Jane Doe" ]`
	`  }`
	`  }]`
	`}}`

The following example matches documents where the JSON property "num" contains the number value 42. Since `type` is set to "number", the query matches a document such as `{ "num": 42 }`, but it will not match a document such as `{ "num": "42" }`. If you did not set type, then the type would be string and the query would match a document such as `{ "num": "42" }` but would not match a document such as `{ "num": "42" }`.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <value-query>`
`  <json-property>num</json-property>`
`  <text>42</text>`
`  <type>number</type>`
`  </value-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "value-query": {`
	`  "json-property": "num",`
	`  "type": "number",`
	`  "text": [ "42" ]`
	`  }`
	`  }]`
	`}}`

Note that in the JSON version of the query, the value of `text` can be either a string ("42") or a number (42). The interpretation of the value depends on the `type` setting in the query.

4.6.21 word-query

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

- Syntax Summary
- Component Description
- Examples

4.6.21.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<word-query>`
`  <element name=elem-name ns=namespace />`
`  <attribute name=attr-name ns=namespace />`
`  <json-property>name</json-property>`
`  <field name=field-name collation=uri />`
`  <fragment-scope>scope</fragment-scope>`
`  <text>name</text>`
`  <term-option>option/term-option>`
`  <weight>value</weight>`
`</word-query>`
	`"word-query": {`
	`  "element": {`
	`  "name": elem-name,`
	`  "ns": namespace`
	`  },`
	`  "attribute": {`
	`  "name": attr-name,`
	`  "ns": namespace`
	`  },`
	`  "json-property": prop-name,`
	`  "field": {`
	`  "name": field-name,`
	`  "collation": uri`
	`  },`
	`  "fragment-scope": scope,`
	`  "text": [ name ],`
	`  "term-option": [ option ],`
	`  "weight": number`
	`}`

4.6.21.2 Component Description

You must specify at least one `element`, `json-property`, `field`, or an `element` and an `attribute` to define the range constraint to apply to the query. These components are mutually exclusive: Except for `element` and `attribute` pairs, a `word-query` must include exactly one type of constraint specifier.

Element or JSON Property Name	Req'd?	Description
`element`	N	An XML element descriptor, identified by element `name` and namespace (`ns`). Both `name` and `ns` are required. You can specify multiple elements, in which case the query matches if a match is found in any of the elements. If you include `element`, you should not include a `json-property` or `field`.
`attribute`	N	An XML attribute descriptor, identifying the attribute by `name` and namespace (`ns`). Both `name` and `ns` are required. You can specify multiple attributes, in which case the query matches if a match is found in any of the attributes. You cannot use this component in conjunction with `json-property` or `field`.
`json-property`	N	A JSON property name. You can specify multiple properties, in which case the query matches if a match is found in any of the properties. If you include `json-property`, you should not include an `element`, `attribute`, or `field`.
`field`	N	A field descriptor, identified by the field `name` (required) and optional `collation`. The database configuration must include a definition for the field. You can specify multiple fields, in which case the query matches if a match is found in any of the fields. If you include `field`, you should not include an `element`, `attribute`, or `json-property`.
`fragment-scope`	N	Constrain matches to the specified fragment scope. Allowed values: `documents` (default) or `properties`. For more details, see the `fragment-scope` query option.
`text`	N	Terms or phrases that must occur in documents matching the constraint defined by this query. Multiple values can be specified; if any one matches, the document matches the query.

Element or JSON Property Name	Req'd?	Description
`term-option`	N	Term options to apply to the query. You can specify multiple term options. If the option has a value, the value of term-option is option=value. For example: `<term-option>min-occurs=1</term-option>`. For details, see the `cts` query corresponding to the query constraint type, such as: `cts:element-word-query`, `cts:element-attribute-word-query`, `cts:json-property-word-query`, or `cts:field-word-query`; and "Term Options" on page 950.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.

4.6.21.3 Examples

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

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<query>`
`  <word-query>`
`  <element name="body-color" ns="" />`
`  <text>black</text>`
`  </word-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "word-query": {`
	`  "element": {`
	`  "name": "body-color",`
	`  "ns": ""`
	`  },`
	`  "text": [ "black" ]`
	`  }`
	`  }]`
	`}}`

4.6.22 geo-elem-query

A query that returns documents that match the geospatial element constraint defined in the query. For details, see `cts:element-geospatial-query`, `cts:element-child-geospatial-query`, or "Geospatial Search Applications" on page 476.

- Syntax Summary
- Component Description
- Examples

4.6.22.1 Syntax Summary

XML	JSON
All elements are in the namespace `http://marklogic.com/appservices/search`. `<geo-elem-query>` `  <parent name=`*elem-name* `ns=`*uri* `/>` `  <element name=`*elem-name* `ns=`*uri* `/>` `  <geo-option>`*option*`</geo-option>` `  <fragment-scope>`*scope*`</fragement-scope>` `  <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>` `  <weight>`*value*`</weight>` `</geo-elem-query>`	`"geo-elem-query": {` `  "parent": {` `  "name": `*elem-name* `  "ns": `*uri* `  },` `  "element": {` `  "name": `*elem-name* `  "ns": `*uri* `  },` `  "geo-option": [ `*option*` ],` `  "fragment-scope": `*scope*`,` `  "point": [` `  {` `  "latitude": `*number*`,` `  "longitude": `*number* `  }` `  ],` `  "box": [` `  {` `  "south": `*number*`,` `  "west": `*number*`,` `  "north": `*number*`,` `  "east": `*number* `  }` `  ],` `  "circle": [` `  {` `  "radius": `*number*`,` `  *point* ` }` `],` ` "polygon": [`*point*`],` ` "weight": `*number* `}`

4.6.22.2 Component Description

A geospatial query contains one or more points or regions, described by `point`, `box`, `circle`, and `polygon` XML child elements or JSON sub-objects. The element pair containing geospatial data are described by `parent`, `lat`, and `lon`. For details, see "Geospatial Search Applications" on page 476

Element or JSON Property Name	Req'd?	Description
`parent`	N	Optional. The parent element of the element containing geospatial data, identified by element `name` and namespace (`ns`). Both `name` and `ns` are required.
`element`	Y	The element containing geospatial data, identified by `name` and namespace (`ns`). Both `name` and `ns` are required.
`fragment-scope`	N	Constrain matches to the specified fragment scope. Allowed values: `documents` (default) or `properties`. For more details, see the `fragment-scope` query option.
`geo-option`	N	Geospatial options to apply to the query. You can specify multiple options. If an option has a value, the value of `geo-option` is of the form *`option=value`*. For example: `<geo-option>units=miles</geo-option>`. For details, see `cts:element-geospatial-query` or `cts:element-child-geospatial-query`.
`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.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.

4.6.22.3 Examples

The following example matches points contained in either of two polygons. The constraint defined in the query by `parent` and `element` defines how to construct the points to match against the regions defined in the query.

Format	Query

XML	`<query xmlns="http://marklogic.com/appservices/search">`
	`  <geo-elem-query>`
	`  <parent ns="ns1" name="elem1"/>`
	`  <element ns="ns1" name="elem2"/>`
	`  <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>`
	`  </geo-elem-query>`
	`</query>`

Format	Query

JSON	<pre>{
	  "query": {
	    "queries": [{
	      "geo-elem-query": {
	        "parent": { "ns": "ns1", "name": "elem1" },
	        "element": { "ns": "ns1", "name": "elem2" },
	        "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>

4.6.23 geo-elem-pair-query

A query that returns documents that match the geospatial XML element constraint defined in the query. For details, see `cts:element-pair-geospatial-query` and "Geospatial Search Applications" on page 476.

- Syntax Summary
- Component Description
- Examples

4.6.23.1 Syntax Summary

XML	JSON
All elements are in the namespace `http://marklogic.com/appservices/search`. `<geo-elem-pair-query>` `  <parent name=`*elem-name* `ns=`*uri* `/>` `  <lat name=`*elem-name* `ns=`*uri* `/>` `  <lon name=`*elem-name* `ns=`*uri* `/>` `  <geo-option>`*option*`</geo-option>` `  <fragment-scope>`*scope*`</fragement-scope>` `  <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>` `  <weight>`*value*`</weight>` `</geo-elem-pair-query>`	`"geo-elem-pair-query": {` `  "parent": {` `  "name": `*elem-name* `  "ns": `*uri* `  },` `  "lat": {` `  "name": `*elem-name* `  "ns": `*uri* `  },` `  "lon": {` `  "name": `*elem-name* `  "ns": `*uri* `  },` `  "geo-option": [ `*option*` ],` `  "fragment-scope": `*scope*`,` `  "point": [` `  {` `  "latitude": `*number*`,` `  "longitude": `*number* `  }` `  ],` `  "box": [` `  {` `  "south": `*number*`,` `  "west": `*number*`,` `  "north": `*number*`,` `  "east": `*number* `  }` `  ],` `  "circle": [` `  {` `  "radius": `*number*`,` `  *point* ` }` `],` ` "polygon": [`*point*`],` ` "weight": `*number* `}`

4.6.23.2 Component Description

A geospatial query contains one or more points or regions, described by `point`, `box`, `circle`, and `polygon` XML child elements or JSON sub-objects. The element pair containing geospatial data are described by `parent`, `lat`, and `lon`.

Element or JSON Property Name	Req'd?	Description
`parent`	N	The element containing geospatial data, identified by element `name` and namespace (`ns`). Both `name` and `ns` are required.
`lat`	Y	The XML element containing latitude data, identified by `name` and namespace (`ns`). Both `name` and `ns` are required.
`lon`	Y	The XML element containing longitude data, identified by `name` and namespace (`ns`). Both `name` and `ns` are required.
`fragment-scope`	N	Constrain matches to the specified fragment scope. Allowed values: `documents` (default) or `properties`. For more details, see the `fragment-scope` query option.
`geo-option`	N	Geospatial options to apply to the query. You can specify multiple options. If an option has a value, the value of `geo-option` is of the form *`option=value`*. For example: `<geo-option>units=miles</geo-option>`. For details, see `cts:element-pair-geospatial-query`.
`point`	N	Zero or more geographic points, each defined by a `latitude` and a `longitude`. The query can contain 0 or more points.

Element or JSON Property Name	Req'd?	Description
`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.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.

4.6.23.3 Examples

The following example matches points contained in either of two polygons. The constraint defined in the query by `parent`, `lat`, and `lon` defines how to construct the points to match against the regions defined in the query.

Format	Query
XML	<query xmlns="http://marklogic.com/appservices/search"> <geo-elem-pair-query> <parent ns="ns1" name="elem2"/> <lat ns="ns2" name="attr2"/> <lon ns="ns3" name="attr3"/> <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> </geo-elem-pair-query> </query>

Format	Query

JSON	{"query": {
	"queries": [{
	"geo-elem-pair-query": {
	"parent": { "ns": "ns1", "name": "elem2" },
	"lat": { "ns": "ns2", "name": "attr2" },
	"lon": { "ns": "ns3", "name": "attr3" }
	"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 }
	]
	}
	]
	}
	}]
	}}

4.6.24 geo-attr-pair-query

A query that returns documents that match the geospatial element constraint defined in the query. For details, see `cts:element-attribute-pair-geospatial-query` or “Geospatial Search Applications” on page 476.

- Syntax Summary
- Component Description
- Examples

4.6.24.1 Syntax Summary

XML	JSON
All elements are in the namespace `http://marklogic.com/appservices/search`. `<geo-attr-pair-query>` `  <parent name=`*elem-name* `ns=`*uri* `/>` `  <lat name=`*attr-name* `ns=`*uri* `/>` `  <lon name=`*attr-name* `ns=`*uri* `/>` `  <geo-option>`*option*`</geo-option>` `  <fragment-scope>`*scope*`</fragement-scope>` `  <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>` `  <weight>`*value*`</weight>` `</geo-attr-pair-query>`	`"geo-attr-pair-query": {` `  "parent": {` `  "name": `*elem-name* `  "ns": `*uri* `  },` `  "lat": {` `  "name": `*attr-name* `  "ns": `*uri* `  },` `  "lon": {` `  "name": `*attr-name* `  "ns": `*uri* `  },` `  "geo-option": [ `*option*` ],` `  "fragment-scope": `*scope*`,` `  "point": [` `  {` `  "latitude": `*number*`,` `  "longitude": `*number* `  }` `  ],` `  "box": [` `  {` `  "south": `*number*`,` `  "west": `*number*`,` `  "north": `*number*`,` `  "east": `*number* `  }` `  ],` `  "circle": [` `  {` `  "radius": `*number*`,` `  *point* ` }` `],` ` "polygon": [`*point*`],` ` "weight": `*number* `}`

4.6.24.2 Component Description

A geospatial query contains one or more points or regions, described by `point`, `box`, `circle`, and `polygon` XML child elements or JSON sub-objects. The element pair containing geospatial data are described by `parent`, `lat`, and `lon`. For details, see "Geospatial Search Applications" on page 476

Element or JSON Property Name	Req'd?	Description
`parent`	Y	The element containing the `lat` and `lon` attributes that hold geospatial data, identified by element `name` and namespace (`ns`). Both `name` and `ns` are required.
`lat`	Y	The name of the attribute that contains latitude data, identified by `name` and namespace (`ns`). Both `name` and `ns` are required.
`lon`	Y	The name of the attribute that contains longitude data, identified by `name` and namespace (`ns`). Both `name` and `ns` are required.
`fragment-scope`	N	Constrain matches to the specified fragment scope. Allowed values: `documents` (default) or `properties`. For more details, see the `fragment-scope` query option.
`geo-option`	N	Geospatial options to apply to the query. You can specify multiple options. If an option has a value, the value of `geo-option` is of the form *`option=value`*. For example: `<geo-option>units=miles</geo-option>`. For details, see `cts:element-attribute-pair-geospatial-query`.
`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.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.

4.6.24.3 Examples

The following example matches points contained in either of two polygons. The constraint defined in the query by `parent`, `lat`, and `lon` defines how to construct the points to match against the regions defined in the query.

Format	Query
XML	<query xmlns="http://marklogic.com/appservices/search"> <geo-attr-pair-query> <parent ns="ns1" name="elem"/> <lat ns="ns1" name="attr1"/> <lon ns="ns1" name="attr2" /> <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> </geo-attr-pair-query> </query>

Format	Query

JSON	<pre>{
	  "query": {
	    "queries": [{
	      "geo-elem-query": {
	        "parent": { "ns": "ns1", "name": "elem1" },
	        "element": { "ns": "ns1", "name": "elem2" },
	        "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>

4.6.25 geo-path-query

A query that returns documents that match the geospatial constraint defined in the query only when it occurs in an XML element, XML attribute, or JSON property that matches a specified XPath expression. For details, see `cts:path-geospatial-query` or “Geospatial Search Applications” on page 476.

- Syntax Summary
- Component Description
- Examples

4.6.25.1 Syntax Summary

<table>
<tr><th>XML</th><th>JSON</th></tr>
<tr><td>All elements are in the namespace
http://marklogic.com/appservices/search.

<pre>
<geo-path-query>
  <path-index>path-expr</path-index>
  <geo-option>option</geo-option>
  <fragment-scope>scope</fragement-scope>
  <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>
  <weight>value</weight>
</geo-path-query>
</pre>
</td><td>
<pre>
"geo-path-query": {
  "path-index": {
    "text": path-expr,
    "namespaces": [{
      prefix: namespace-uri
    }]
  },
  "geo-option": [option],
  "fragment-scope": scope,
  "point": [
    {
      "latitude": number,
      "longitude": number
    }
  ],
  "box": [
    {
      "south": number,
      "west": number,
      "north": number,
      "east": number
    }
  ],
  "circle": [
    {
      "radius": number,
      point
    }
  ],
  "polygon": [point],
  "weight": number
}
</pre>
</td></tr>
</table>

4.6.25.2 Component Description

A geospatial query contains one or more points or regions, described by `point`, `box`, `circle`, and `polygon` XML child elements or JSON sub-objects. The element or attribute containing geospatial data are described by `path-index`. For details, see “Geospatial Search Applications” on page 476

Element or JSON Property Name	Req'd?	Description
`path-index`	Y	A path range expression matching an element or attribute whose contents represent a point contained within the given geographic region(s). If the path expression includes namespace prefixes, you must define the namespace bindings on the `path-index`. The database configuration must include a matching path range index. The path expression and namespace URIs must match the index configuration; namespace prefixes do not have to match. The path expression is limited to the subset of XPath that can be used to define a path range index. For details, see Path Field and Path-Based Range Index Configuration in the *XQuery and XSLT Reference Guide*.
`fragment-scope`	N	Constrain matches to the specified fragment scope. Allowed values: `documents` (default) or `properties`. For more details, see the `fragment-scope` query option.
`geo-option`	N	Geospatial options to apply to the query. You can specify multiple options. If an option has a value, the value of `geo-option` is of the form *`option=value`*. For example: `<geo-option>units=miles</geo-option>`. For details, see `cts:element-attribute-pair-geospatial-query`.
`point`	N	Zero or more geographic points, each defined by a `latitude` and a `longitude`. The query can contain 0 or more points.

Element or JSON Property Name	Req'd?	Description
`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.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.

4.6.25.3 Examples

The following example matches points contained in either of two polygons. The constraint defined in the query by `path-index` defines how to construct the points to match against the regions defined in the query.

Format	Query

XML	`<query xmlns="http://marklogic.com/appservices/search">`
	`  <geo-path-query>`
	`  <path-index xmlns:ns1="/my/ns">/ns:a/ns:b</path-index>`
	`  <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>`
	`  </geo-path-query>`
	`</query>`

Format	Query

JSON	{"query":{
	"geo-path-query":{
	"path-index": {
	"text": "/ns1:a/ns2:b",
	"namespaces": [{"ns1": "/my/ns1"}, {"ns2": "my/ns2"}]
	},
	"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 }
	] }
	]
	}
	}}

4.6.26 geo-json-property-query

A query that returns documents that match the geospatial constraint defined in the query only when it occurs a specified JSON property. For details, see `cts:json-property-geospatial-query`, `cts:json-property-child-geospatial-query`, or “Geospatial Search Applications” on page 476.

- Syntax Summary
- Component Description
- Examples

4.6.26.1 Syntax Summary

XML	JSON
All elements are in the namespace `http://marklogic.com/appservices/search`. `<geo-json-property-query>` `  <parent-property>`*name*`</parent-property>` `  <json-property>`*name*`</json-property>` `  <geo-option>`*option*`</geo-option>` `  <fragment-scope>`*scope*`</fragement-scope>` `  <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>` `  <weight>`*value*`</weight>` `</geo-json-property-query>`	`"geo-json-property-query": {` `  "parent-property":` *name*`,` `  "json-property":` *name*`,` `  "geo-option": [` *option* `],` `  "fragment-scope":` *scope*`,` `  "point": [` `  {` `  "latitude":` *number*`,` `  "longitude":` *number* `  }` `  ],` `  "box": [` `  {` `  "south":` *number*`,` `  "west":` *number*`,` `  "north":` *number*`,` `  "east":` *number* `  }` `  ],` `  "circle": [` `  {` `  "radius":` *number*`,` `  *point* ` }` `],` ` "polygon": [` *point* `],` ` "weight":` *number* `}`

4.6.26.2 Component Description

A geospatial query contains one or more points or regions, described by `point`, `box`, `circle`, and `polygon` XML child elements or JSON sub-objects. The element pair containing geospatial data are described by `parent`, `lat`, and `lon`. For details, see “Geospatial Search Applications” on page 476

Element or JSON Property Name	Req'd?	Description
`parent-property`	N	Optional. The parent property of the property containing geospatial data, identified by name.
`json-property`	Y	The name of the property containing geospatial data.
`fragment-scope`	N	Constrain matches to the specified fragment scope. Allowed values: `documents` (default) or `properties`. For more details, see the `fragment-scope` query option.
`geo-option`	N	Geospatial options to apply to the query. You can specify multiple options. If an option has a value, the value of `geo-option` is of the form *`option=value`*. For example: `<geo-option>units=miles</geo-option>`. For details, see `cts:json-property-geospatial-query` or `cts:json-property-child-geospatial-query`.
`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.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.

4.6.26.3 Examples

The following example matches points contained in either of two polygons. The JSON property defined in the query by `parent-property` and `json-property` defines how to construct the points to match against the regions defined in the query.

Format	Query

XML	`<query xmlns="http://marklogic.com/appservices/search">`
	`  <geo-json-property-query>`
	`  <parent-property>myParent</parent>`
	`  <json-property>loc</json-property>`
	`  <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>`
	`  </geo-json-property-query>`
	`</query>`

Format	Query

JSON	<pre>{
	  "query": {
	    "queries": [{
	      "geo-json-property-query": {
	        "parent-property": "myParent",
	        "json-property": "loc",
	        "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>

4.6.27 geo-json-property-pair-query

A query that returns documents that match the geospatial constraint defined in the query only when it occurs a JSON property that matches a specified XPath expression. For details, see `cts:json-property-pair-geospatial-query` or “Geospatial Search Applications” on page 476.

- Syntax Summary
- Component Description
- Examples

4.6.27.1 Syntax Summary

XML	JSON
All elements are in the namespace `http://marklogic.com/appservices/search`. `<geo-json-property-pair-query>` `  <parent-property>`*name*`</parent-property>` `  <lat-property>`*name*`</lat-property>` `  <lon-property>`*name*`</lon-property>` `  <geo-option>`*option*`</geo-option>` `  <fragment-scope>`*scope*`</fragement-scope>` `  <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>` `  <weight>`*value*`</weight>` `</geo-json-property-pair-query>`	`"geo-json-property-pair-query": {` `  "parent-property":` *name*`,` `  "lat-property":` *name*`,` `  "lon-property":` *name*`,` `  "geo-option": [` *option* `],` `  "fragment-scope":` *scope*`,` `  "point": [` `  {` `  "latitude":` *number*`,` `  "longitude":` *number* `  }` `  ],` `  "box": [` `  {` `  "south":` *number*`,` `  "west":` *number*`,` `  "north":` *number*`,` `  "east":` *number* `  }` `  ],` `  "circle": [` `  {` `  "radius":` *number*`,` `  *point* ` }` `],` ` "polygon": [` *point* `],` ` "weight":` *number* `}`

4.6.27.2 Component Description

A geospatial query contains one or more points or regions, described by `point`, `box`, `circle`, and `polygon` XML child elements or JSON sub-objects. The element pair containing geospatial data are described by `parent-property`, `lat-property`, and `lon-property`. For details, see “Geospatial Search Applications” on page 476

Element or JSON Property Name	Req'd?	Description
`parent-property`	Y	The name of JSON property containing `lat-property` and `lon-property`.
`lat-property`	Y	The name of the JSON property that contains latitude data.
`lon-property`	Y	The name of the JSON property that contains longitude data.
`fragment-scope`	N	Constrain matches to the specified fragment scope. Allowed values: `documents` (default) or `properties`. For more details, see the `fragment-scope` query option.
`geo-option`	N	Geospatial options to apply to the query. You can specify multiple options. If an option has a value, the value of `geo-option` is of the form *`option=value`*. For example: `<geo-option>units=miles</geo-option>`. For details, see `cts:json-property-pair-geospatial-query`.
`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.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.

4.6.27.3 Examples

The following example matches points contained in either of two polygons. The constraint defined in the query by `parent`, `lat`, and `lon` defines how to construct the points to match against the regions defined in the query.

Format	Query
XML	<query xmlns="http://marklogic.com/appservices/search"> <geo-json-property-pair-query> <parent-property>myParent</parent> <lat-property>loc</lat-property> <lon-property>lon</lon-property> <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> </geo-json-property-pair-query> </query>

Format	Query

JSON	<pre>{
	  "query": {
	    "queries": [{
	      "geo-json-property-pair-query": {
	        "parent-property": "myParent",
	        "lat-property": "lat",
	        "lon-property": "lon",
	        "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>

4.6.28 geo-region-path-query

A query that returns documents containing at least one region *R1* that satisfies the requirement *R1 op R2*, for some topological operator *op* and criteria region *R2*. For example, "R1 contains R2" or "R1 intersects R2". The regions in the documents are identified by a reference to a geospatial region path index. The criteria regions are defined in the query. For details, see `cts:geospatial-region-query`, `cts.geospatialRegionQuery`, or "Searching for Matching Regions" on page 528.

If the query defines multiple criteria regions, a document matches the query if *R1 op R2* is true for any one (*R1*, *R2*) pair. That is, specifying multiple criteria regions is like an implicit OR query.

- Syntax Summary
- Component Description
- Examples

4.6.28.1 Syntax Summary

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<geo-region-path-query coord="coord-sys">`
`  <path-index>path-expr</path-index>`
`  <geospatial-operator>op</geospatial-operator>`
`  <geo-option>option</geo-option>`
`  <fragment-scope>scope</fragement-scope>`
`  <weight>value</weight>`
`  <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>`
`</geo-region-path-query>`
	`"geo-region path-query": {`
	`  "path-index": {`
	`  "text": path-expr,`
	`  "namespaces": [{`
	`  prefix: namespace-uri`
	`  }]`
	`  },`
	`  "geospatial-operator": op,`
	`  "coord": coord-sys-name,`
	`  "geo-option": [ option ],`
	`  "weight": number,`
	`  "fragment-scope": scope,`
	`  "point": [`
	`  {`
	`  "latitude": number,`
	`  "longitude": number`
	`  }`
	`  ],`
	`  "box": [`
	`  {`
	`  "south": number,`
	`  "west": number,`
	`  "north": number,`
	`  "east": number`
	`  }`
	`  ],`
	`  "circle": [`
	`  {`
	`  "radius": number,`
	`  point`
	`  }`
	`  ],`
	`  "polygon": [ point ]`
	`}`

4.6.28.2 Component Description

A geospatial region query contains one or more point or region criteria, described by `point`, `box`, `circle`, and `polygon` XML child elements or JSON sub-objects. The XML element or JSON property defining a region in your documents is identified by `path-index`. For details, see "Geospatial Region Queries and Indexes" on page 506.

Element or JSON Property Name	Req'd?	Description
`path-index`	Y	A path range expression matching an XML element or JSON property whose contents represent a region. If the path expression includes namespace prefixes, you must define the namespace bindings on the `path-index` in XML or using `path-index/namespaces` in JSON; for details, see the Examples, below. The database configuration must include a matching geospatial region path index. The path expression and namespace URIs must match the index configuration; namespace prefixes do not have to match. The path expression is limited to the subset of XPath that can be used to define a path range index. For details, see Path Field and Path-Based Range Index Configuration in the *XQuery and XSLT Reference Guide*.
`geospatial-operator`	N	A topological operator. One of `contains`, `covered-by`, `covers`, `disjoint`, `intersects`, `overlaps`, `within`. Default: `contains`. For a region *R1* in the specified region index and a search criteria region *R2*, a document matches if *R1 op R2* is true.
`coord`	N	Specify the coordinate system and precision of the region index associated with the path in `path-index`. Allowed values: `wgs84`, `wgs84/double`, `etrs89`, `etrs89/double`, `raw`, `raw/double`. You must specify this value if `path-index` is not sufficient to unambiguously identify the target region index.
`geo-option`	N	Geospatial options to apply to the query. You can specify multiple options. If an option has a value, the value of `geo-option` is of the form *`option=value`*. For example: `<geo-option>units=miles</geo-option>` in XML, or `"units=miles"` in JSON. For details, see `cts:geospatial-region-query` or `cts.geospatialRegionQuery`.

Element or JSON Property Name	Req'd?	Description
`fragment-scope`	N	Constrain matches to the specified fragment scope. Allowed values: `documents` (default) or `properties`. For more details, see the `fragment-scope` query option.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.
`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.

4.6.28.3 Examples

The following example matches regions in documents that intersect either of the two criteria polygons. A geospatial region index matching `path-index` and `coord` must exist.

Format	Query
XML	<query xmlns="http://marklogic.com/appservices/search"> <geo-region-path-query coord="wgs84/double"> <path-index xmlns:ns="/my/ns">/ns:a/ns:b</path-index> <geospatial-operator>intersects</geospatial-operator> <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> </geo-region-path-query> </query>

JSON	{"query":{
	  "geo-region-path-query":{
	  "path-index": {
	  "text": "/ns1:a/ns2:b",
	  "namespaces": [{"ns1": "/my/ns1"}, {"ns2": "my/ns2"}]
	  },
	  "coord": "wgs84/double",
	  "geospatial-operator": "intersects",
	  "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 }
	  ] }
	  ]
	  }
	}}

4.6.29 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 382 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

4.6.29.1 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-option>`*`option`*`</range-option>`
`</range-constraint-query>`
	`"range-constraint-query": {`
	`  "constraint-name": "`*`name`*`",`
	`  "value": [ `*`value-to-match`*` ],`
	`  "range-operator": "`*`operator`*`",`
	`  "range-option": [ `*`option`*` ]`
	`}`

4.6.29.2 Component Description

Element or JSON Property Name	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 XML elements, XML element attributes, JSON properties, 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`.
`range-option`	N	One or more range query options. Allowed values depend on the type of range query (element, path, field, etc.). For details, see Including a Range or Geospatial Query in Scoring in *Search Developer's Guide*. For a list of options, see "Range Options" on page 951.

4.6.29.3 Examples

The following example matches documents containing a `<body-color/>` element with a value of `black`, assuming an element range index exists on `<body-color/>`.

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">`
	`  <range-constraint-query>`
	`  <constraint-name>color</constraint-name>`
	`  <value>black</value>`
	`  </range-constraint-query>`
	`</query>`

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

4.6.30 value-constraint-query

A query that matches fragments where the value of the content of an XML element, XML attribute, JSON property, or field exactly matches the `text, number, boolean, or null` value in the query. The match semantics depend on the value, the database configuration, and the options in effect. The element, attribute, property, 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`, `cts:json-property-value-query`.

- Syntax Summary
- Component Description
- Examples

4.6.30.1 Syntax Summary

XML	JSON

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

4.6.30.2 Component Description

Element or JSON Property Name	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`, `number`, `boolean`, or `null` component, all values matching the constraint are returned.
`number`	N	A numeric value to match. Applicable to only to JSON documents. Multiple values can be specified. If there is no `text`, `number`, `boolean`, or `null` component, all values matching the constraint are returned.
`boolean`	N	A boolean value to match. Applicable to only to JSON documents. Multiple values can be specified. If there is no `text`, `number`, `boolean`, or `null` component, all values matching the constraint are returned.
`null`	N	Match a null value. Applicable to only to JSON documents. Multiple values can be specified. If there is no `text`, `number`, `boolean`, or `null` 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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.

4.6.30.3 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"`
	`  }`
	`  }]`
	`}}`

4.6.31 word-constraint-query

A query that matches fragments containing the specified terms or phrases in the XML element or attribute, JSON property, 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

4.6.31.1 Syntax Summary

XML	JSON

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

4.6.31.2 Component Description

Element or JSON Property Name	Req'd?	Description
`constraint-name`	Y	The name of a word constraint defined in the global or query-specific query options. If you include multiple constraint names, the query matches if any of the constraints are met.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.

4.6.31.3 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"`
	`  }`
	`  }]`
	`}}`

4.6.32 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

4.6.32.1 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*` ]`
	`}`

4.6.32.2 Component Description

Element or JSON Property Name	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.

4.6.32.3 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"`
	`  }`
	`  }]`
	`}}`

4.6.33 container-constraint-query

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

- Syntax Summary
- Component Description

- Examples

4.6.33.1 Syntax Summary

XML	JSON

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

4.6.33.2 Component Description

Element or JSON Property Name	Req'd?	Description
`constraint-name`	Y	The name of a constraint defined in the global or query-specific query options. If you specify multiple constraints, the query matches documents that satisfy any one of the constraints.
`anyQueryType`	Y	A query to run against containers matching the constraint identified by `constraint-name`.

4.6.33.3 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">`
	`  <container>`
	`  <element name="color" ns="" />`
	`  </container>`
	`  </constraint>`
	`</options>`
	
	`<query xmlns="http://marklogic.com/appservices/search">`
	`  <container-constraint-query>`
	`  <constraint-name>body-color</constraint-name>`
	`  <term-query>`
	`  <text>black</text>`
	`  </term-query>`
	`  </container-constraint-query>`
	`</query>`

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

4.6.34 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`.

> **Note:** Use of this query type is deprecated. Use container-constraint-query instead.

- Syntax Summary
- Component Description
- Examples

4.6.34.1 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`*
	`}`

4.6.34.2 Component Description

Element or JSON Property Name	Req'd?	Description
`constraint-name`	Y	The name of a container constraint defined in the global or query-specific query options. If you specify multiple constraints, the query matches documents that satisfy any one of the constraints.
`anyQueryType`	Y	A query to run against elements matching the constraint identified by `constraint-name`.

4.6.34.3 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-query 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" ]`
	
	`  }`
	`  }]`
	`}}`

4.6.35 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-fragment-query`.

- Syntax Summary
- Component Description
- Examples

4.6.35.1 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`*
	`}`

4.6.35.2 Component Description

Element or JSON Property Name	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`.

4.6.35.3 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" ]`
	`  }`
	`  }`
	`  }]`
	`}}`

4.6.36 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 42.

- Syntax Summary

- Component Description
- Examples

4.6.36.1 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*` ]`
	`}`

4.6.36.2 Component Description

Element or JSON Property Name	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`.

4.6.36.3 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"`
	`  }`
	`  }]`
	`}}`

4.6.37 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 476.

- Syntax Summary
- Component Description
- Examples

4.6.37.1 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*` ]`
	`}`

4.6.37.2 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 or JSON Property Name	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.

4.6.37.3 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	{
	  "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 }
	  ]
	  }
	  ]
	  }
	  }]
	}}

4.6.38 geo-region-constraint-query

A query that returns documents that match the specified geospatial region constraint and the matching fragments also match the geospatial region queries. For details, see “Geospatial Search Applications” on page 476.

A query that returns documents containing at least one region *R1* that satisfies the requirement *R1 op R2*, for some topological operator *op* and criteria region *R2*. For example, “R1 contains R2” or “R1 intersects R2”. The regions in the documents are identified by a reference to a geospatial region path index. The criteria regions are defined in the query. For details, see `cts:geospatial-region-query`, `cts.geospatialRegionQuery`, or “Searching for Matching Regions” on page 528.

If the query defines multiple criteria regions, a document matches the query if *R1 op R2* is true for any one (*R1*, *R2*) pair. That is, specifying multiple criteria regions is like an implicit OR query.

- Syntax Summary
- Component Description
- Examples

4.6.38.1 Syntax Summary

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<geo-region-path-query coord="coord-sys">`
`  <path-index>path-expr</path-index>`
`  <geospatial-operator>op</geospatial-operator>`
`  <geo-option>option</geo-option>`
`  <fragment-scope>scope</fragement-scope>`
`  <weight>value</weight>`
`  <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>`
`</geo-region-path-query>`
	`"geo-region path-query": {`
	`  "path-index": {`
	`  "text": path-expr,`
	`  "namespaces": [{`
	`  prefix: namespace-uri`
	`  }]`
	`  },`
	`  "geospatial-operator": op,`
	`  "coord": coord-sys-name,`
	`  "geo-option": [ option ],`
	`  "weight": number,`
	`  "fragment-scope": scope,`
	`  "point": [`
	`  {`
	`  "latitude": number,`
	`  "longitude": number`
	`  }`
	`  ],`
	`  "box": [`
	`  {`
	`  "south": number,`
	`  "west": number,`
	`  "north": number,`
	`  "east": number`
	`  }`
	`  ],`
	`  "circle": [`
	`  {`
	`  "radius": number,`
	`  point`
	`  }`
	`  ],`
	`  "polygon": [ point ]`
	`}`

4.6.38.2 Component Description

A geospatial region query contains one or more point or region criteria, described by `point`, `box`, `circle`, and `polygon` XML child elements or JSON sub-objects. The XML element or JSON property defining a region in your documents is identified by `path-index`. For details, see "Geospatial Region Queries and Indexes" on page 506.

Element or JSON Property Name	Req'd?	Description
`constraint-name`	Y	The name of a geospatial region constraint defined in the global or query-specific query options.
`geospatial-operator`	N	A topological operator. One of `contains`, `covered-by`, `covers`, `disjoint`, `intersects`, `overlaps`, `within`. Default: `contains`. For a region *R1* in the specified region index and a search criteria region *R2* specified in this query, a document matches if *R1 op R2* is true.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.
`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.

4.6.38.3 Examples

The following example matches regions in documents that intersect either of the two criteria polygons. A geospatial region index matching the constraint definition must exist.

Format	Query
XML	<options xmlns="http://marklogic.com/appservices/search"> <constraint name="my-geo-region"> <geo-region-path coord="wgs84"> <path-index>/a/b</path-index> <geo-option>units=feet</geo-option> </geo-region-path> </constraint> </options> <query xmlns="http://marklogic.com/appservices/search"> <geo-region-constraint-query> <constraint-name>my-geo-region</constraint-name> <geospatial-operator>intersects</geospatial-operator> <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> </geo-region-path-query> </query>

Format	Query

JSON	`{ "options": {`
	`  "constraint": [{`
	`  "name": "my-geo-region",`
	`  "geo-region-path": {`
	`  "path-index": "/a/b",`
	`  "coord": "wgs84",`
	`  "geo-option": ["units=feet"]`
	`  }`
	`  }]`
	`}}`
	
	`{ "query": {`
	`  "queries": [{`
	`  "geo-region-constraint-query":{`
	`  "constraint-name": "my-geo-region",`
	`  "geospatial-operator": "intersects",`
	`  "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 }`
	`  ] }`
	`  ]`
	`  }`
	`  }]`
	`}}`

4.6.39 lsqt-query

A query that returns documents before Last Stable Query Time (LSQT) or before a given timestamp that is before LSQT. For details, see `cts:lsqt-query` or Searching Temporal Documents in the *Temporal Developer's Guide*.

- Syntax Summary
- Component Description
- Examples

4.6.39.1 Syntax Summary

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<lsqt-query>`
`  <temporal-collection>`*name*`</temporal-collection>`
`  <timestamp>`*dateTime*`</timestamp>`
`  <weight>`*value*`</weight>`
`  <temporal-option>`*option*`</temporal-option>`
`</lsqt-query>`
	`"lsqt-query": {`
	`  "temporal-collection":` *name*`,`
	`  "timestamp":` *string*`,`
	`  "weight":` *number*`,`
	`  "temporal-option": [` *string* `]`
	`}`

4.6.39.2 Component Description

Element or JSON Property Name	Req'd?	Description
`temporal-collection`	Y	The name of a temporal collection.
`timestamp`	N	Return only temporal documents with a system start time less than or equal to this value. Timestamps greater than LSQT are rejected. Default: LSQT for the named temporal collection.
`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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.
`temporal-option`	N	Temporal options to apply to the query. You can specify multiple options. If the option has a value, the value of `temporal-option` is `option=value`. For example: `<temporal-option>score-function=linear</temporal-option>`. For available options, see `cts:lsqt-query`.

4.6.39.3 Examples

The following example returns documents in the temporal collection “myTemporalCollection” before LSQT.

Format	Query

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	
	`</options>`
	
	`<query xmlns:search="http://marklogic.com/appservices/search">`
	`  <lsqt-query>`
	`  <temporal-collection>myTemporalCollection</temporal-collection>`
	`  <temporal-option>cached-incremental</temporal-option>`
	`  </lsqt-query>`
	`</query>`

JSON	`{"query": {`
	`  "queries": [{`
	`  "lsqt-query": {`
	`  "temporal-collection": "myTemporalCollection",`
	`  "temporal-option": [ "cached-incremental" ]`
	`  }`
	`  }]`
	`}}`

4.6.40 period-compare-query

A query that matches documents for which the specified relationship holds between two temporal axes. For details, see `cts:period-compare-query` or Searching Temporal Documents in the *Temporal Developer's Guide*.

- Syntax Summary
- Component Description
- Examples

4.6.40.1 Syntax Summary

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<period-compare-query>`
`  <axis1>axis-`*`name`*`</axis1>`
`  <temporal-operator>`
`  `*`operator`*
`  </temporal-operator>`
`  <axis2>axis-`*`name`*`</axis2>`
`  <temporal-option>`*`option`*`</temporal-option>`
`</period-compare-query>`
	`"period-compare-query": {`
	`  "axis1": `*`name`*`,`
	`  "temporal-operator": `*`string`*`,`
	`  "axis2": `*`name`*`,`
	`  "temporal-option": [ `*`string`*` ]`
	`}`

4.6.40.2 Component Description

Element or JSON Property Name	Req'd?	Description
`axis1`	Y	The name of the first temporal axis.
`temporal-operator`	Y	The comparison operation to apply to the two axes. For a list of operator names, see `cts:period-compare-query` and Period Comparison Operators in the *Temporal Developer's Guide*.
`axis2`	Y	The name of the second temporal axis.
`temporal-option`	N	Temporal options to apply to the query. You can specify multiple options. If the option has a value, the value of `temporal-option` is `option=value`. For example: `<temporal-option>score-function=linear</temporal-option>`. For available options, see `cts:lsqt-query`.

4.6.40.3 Examples

The following example matches documents that were in the database when the time period defined by the axis named “valid” is within the time period defined by the axis “system”.

Format	Query

XML	`<query xmlns:search="http://marklogic.com/appservices/search">`
	`  <period-compare-query>`
	`  <axis1>system</axis1>`
	`  <temporal-operator>iso_contains</temporal-operator>`
	`  <axis2>valid</axis2>`
	`  </period-compare-query>`
	`</query>`

JSON	`{"query": {`
	`  "queries": [{`
	`  "period-compare-query": {`
	`  "axis1": "system",`
	`  "temporal-operator": "iso_contains",`
	`  "axis2": "valid"`
	`  }`
	`  }]`
	`}}`

4.6.41 period-range-query

- Syntax Summary
- Component Description
- Examples

4.6.41.1 Syntax Summary

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<period-range-query>`
`  <axis>axis-`*`name`*`</axis>`
`  <temporal-operator>`
`    `*`operator`*
`  </temporal-operator>`
`  <period>`
`    <period-start>dateTime</period-start>`
`    <period-end>period-end</period-end>`
`  </period>`
`  <temporal-option>`*`option`*`</temporal-option>`
`  <weight>`*`value`*`</weight>`
`</period-range-query>`
	`"period-range-query": {`
	`  "axis": [ `*`name`*` ],`
	`  "temporal-operator": `*`string`*`,`
	`  "period": [`
	`    {`
	`      "period-start": `*`string`*`,`
	`      "period-end": `*`string`*
	`    }`
	`  ],`
	`  "temporal-option": [ `*`string`*` ],`
	`  "weight": `*`number`*
	`}`

4.6.41.2 Component Description

Element or JSON Property Name	Req'd?	Description
`axis`	Y	The name of a temporal axis. You can specify multiple axis names.
`temporal-operator`	Y	The comparison operation to apply to the axis and period. For a list of operator names, see `cts:period-range-query` and Period Comparison Operators in the *Temporal Developer's Guide*.
`period`	Y	One or more periods to match. When multiple periods are specified, the query matches if any value matches.

Element or JSON Property Name	Req'd?	Description
`temporal-option`	N	Temporal options to apply to the query. You can specify multiple options. If the option has a value, the value of `temporal-option` is `option=value`. For example: `<temporal-option>score-function=linear</temporal-option>`. For available options, see `cts:lsqt-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 64 and greater than or equal to -16 (between -16 and 64, inclusive). Weights greater than 64 have the same effect as a weight of 64. 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.

4.6.41.3 Examples

The following example matches temporal documents a valid end time before. 14:00.

Format	Query

XML	`<query xmlns:search="http://marklogic.com/appservices/search">`
	`  <period-range-query>`
	`  <axis>valid</axis>`
	`  <temporal-operator>aln_before</temporal-operator>`
	`  <period>`
	`  <period-start>2014-04-03T14:00:00</period-start>`
	`  <period-end>9999-12-31T11:59:59Z</period-end>`
	`  </period>`
	`  </period-range-query>`
	`</query>`

Format	Query

JSON	{"query": {
	  "queries": [{
	  "period-range-query": {
	  "axis": ["valid"],
	  "temporal-operator": "aln_before",
	  "period": [
	  {
	  "period-start": "2014-04-03T14:00:00",
	  "period-end": "9999-12-31T11:59:59Z"
	  }
	  ]
	  }
	  }]
	}}

4.6.42 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 395.

- Syntax Summary
- Component Description
- Examples

4.6.42.1 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*"
	}

4.6.42.2 Component Description

Element or JSON Property Name	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.

4.6.42.3 Examples

The following examples illustrate use of a custom `sort` operator defined in the query options. The example structured queries is equivalent to the string query "`sort:date`". For details, see "Operator Options" on page 395.

Format	Code Example

XML options	`<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>`

XML query	`<query xmlns:search="http://marklogic.com/appservices/search">`
	`  <operator-state>`
	`  <operator-name>sort</operator-name>`
	`  <state-name>date</state-name>`
	`  </operator-state>`
	`</query>`

Format	Code Example

JSON options	<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 }
	          ]
	      }]
	    }]
	}}</pre>

JSON query	<pre>{
	  "query": {
	    "queries": [{
	      "operator-state": {
	        "operator-name": "sort",
	        "state-name": "date"
	      }
	    }]
	}}</pre>

5.0 Searching Using Query By Example

This chapter describes how to perform searches using Query By Example (QBE). A QBE is a query whose structure closely models the structure of the documents you want to match. You can use a QBE to search XML and JSON documents with the REST, Node.js and Java APIs.

This chapter includes the following sections:

- QBE Overview
- Example
- Understanding QBE Sub-Query Types
- Search Criteria Quick Reference
- QBE Structural Reference
- How Indexing Affects Your Query
- Adding Options to a QBE
- Customizing Search Results
- Scoping a Search by Document Type
- Converting a QBE to a Combined Query
- Validating a QBE

For details on supporting APIs, see *Java Application Developer's Guide* and *REST Application Developer's Guide*.

5.1 QBE Overview

The simple, intuitive syntax of a Query By Example (QBE) enables rapid prototyping of queries for "documents that look like this" because search criteria in a QBE resemble the structure of documents in your database. In its simplest form, a QBE models one or more XML elements, XML element attributes, or JSON properties in your documents.

For example, if your documents include an `author` XML element or JSON property, you can use the following QBE to find documents with an `author` value of "Mark Twain".

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <author>Mark Twain</author>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{`
	`  "$query": { "author": "Mark Twain" }`
	`}`

A QBE always contains a `query` component in which you define search criteria. A QBE can include an optional `response` component for customizing search results, and flags and options that control search behaviors. For details, see "QBE Structural Reference" on page 223.

QBE exposes many powerful features of the Search API, including the following:

- Search Criteria Based on Document Structure
- Logical Operators. Create complex composed queries using AND, OR, NOT and NEAR operators.
- Comparison Operators. Create range queries that test the value of XML elements, XML attributes, and JSON properties using comparison operators such as less than, greather than, and not equal.
- Query by Value or Word. Choose to match values exactly or as subset of the contained content.
- Search Result Customization. Control what to include in the results and snippets returned by your search.
- Options for Controlling Search Behavior. Use options and flags to control query behaviors such as case sensitivity, weights, and number of occurrences.

You can prototype queries using QBE without creating any database indexes, though doing so has implications for performance. For details, see "How Indexing Affects Your Query" on page 234.

This chapter covers the syntax and semantics of QBE. You can use a QBE to search XML and JSON documents with the following MarkLogic APIs:

API	More Information
Node.js Client API	Searching with Query By Example in the *Node.js Application Developer's Guide*.
Java Client API	Prototype a Query Using Query By Example in the *Java Application Developer's Guide*
REST Client API	Using Query By Example to Prototype a Query in the in *REST Application Developer's Guide*.

If you need access to more advanced search features, APIs are available for converting a QBE to a combined query, giving you a foundation on which to build. For details, refer to Client API documentation.

5.1.1 Search Criteria Based on Document Structure

A QBE uses search criteria expressed as XML elements, XML element attributes, or JSON properties that closely resemble portions of documents in the database.

For example, if the database contains documents of the following form:

Format	Example Document

XML	`<book>`
	`  <title>Tom Sawyer</title>`
	`  <author>Mark Twain</author>`
	`  <edition format="paperback"/>`
	`</book>`

JSON	`"book": {`
	`  "title": "Tom Sawyer",`
	`  "author" : "Mark Twain",`
	`  "edition": [`
	`  { "format": "paperback" }`
	`  ]`
	`} }`

Then you can construct a QBE to find all paperback books by a given author by creating criteria that model the `author`, edition `format`. The following QBE finds all paperback books by Mark Twain.

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <author>Mark Twain</author>`
	`  <edition format="paperback"/>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{"$query": {`
	`  "author": "Mark Twain",`
	`  "edition": {`
	`  "format": "paperback"`
	`  }`
	`} }`

By default, the literal values in criteria must exactly match document contents. That is, the above query matches if the `author` value is “Mark Twain”, but it will not match documents where the author is “M. Twain” or “mark twain”. You can change this behavior using word queries and options. For details, see “Understanding QBE Sub-Query Types” on page 208 and “Adding Options to a QBE” on page 235.

You can construct criteria that express value, word, and range queries. For example, you can construct a QBE that satisfies all of the following criteria. The Example Criteria column shows an XML and a JSON criteria that expresses each requirement.

Requirement	Query Type	Example Criteria

the author includes “twain”	word	`<author><q:word>twain</q:word></author>`
		
		`"author": {"$word": "shakespeare"}`

there is a paperback edition	value	`<edition format="paperback"/>`
		
		`"edition": {"format": "paperback"}`

the price of the paperback edition is less than 9.00	range	`<edition>`
		`  <price><q:lt>9.00</q:lt></price>`
		`</edition>`
		
		`"edition": {"price": {"$lt": 9.00} }`

When you combine the above criteria into a single query, you get the following QBE. Notice that the child elements of `query` are implicitly AND’d together.

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <author><q:word>twain</q:word></author>`
	`  <edition format="paperback">`
	`  <price><q:lt>9.00</q:lt></price>`
	`  </edition>`
	`  <q:filtered>true</q:filtered>`
	`  </q:query>`
	`</q:qbe>`

Format	Example

JSON	{
	  "$query": {
	  "author": {
	  "$word": "twain"
	  },
	  "edition": {
	  "format": "paperback",
	  "price": { "$lt": 9.00 }
	  },
	  "$filtered": true
	  }
	}

The above examples demonstrate searching for direct containment, such as "the author is Mark Twain." You can also search for matches anywhere within a containing XML element or JSON property. For example, suppose a `book` contains `author` and `editor` names, broken down into `first-name` and `last-name`:

Format	Example Document
XML	<book> <author> <first-name>Mark</first-name> <last-name>Twain</last-name> </author> <editor> <first-name>Mark</first-name> <last-name>Matthews</last-name> </editor> </book>

JSON	"book": {
	  "author" : {
	  "first-name": "Mark",
	  "last-name": "Twain"
	  },
	"editor" : {
	  "first-name": "Mark",
	  "last-name": "Matthews"
	  }
	} }

You can search for “any occurences of Mark as a first name contained by a book” using criteria such as the following:

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <book><first-name>Mark</last-name></book>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{"$query": {`
	`  "book": { "first-name": "Mark" }`
	`} }`

Such criteria represent container queries. For details, see “Container Query” on page 214.

5.1.2 Logical Operators

You can use logical “operators” to create powerful composed queries. The QBE grammar supports `and`, `or`, `not`, and `near` composers. The following example matches documents that contain “twain” or “shakespeare” in the `author` XML element or JSON property.

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <q:or>`
	`  <author><q:word>twain</q:word></author>`
	`  <author><q:word>shakespeare</q:word></author>`
	`  </q:or>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{`
	`  "$query" : {`
	`  "$or": [`
	`  { "author": {"$word": "twain" } },`
	`  { "author": {"$word": "shakespeare" } }`
	`  ]`
	`  }`
	`}`

Sub-queries that are immediate children of query represent an implicit `and` query.

For details, see “Composed Query” on page 213.

5.1.3 Comparison Operators

The QBE grammar supports the following comparison operators for constructing range queries on XML element, XML attribute, and JSON property values: `lt`, `le`, `eq`, `ne`, `ge`, `gt`. For example, the following query matches all documents where the price is greater than or equal to 10.00 and less than or equal to 20.00.

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <price><q:ge>10.00</q:ge></price>`
	`  <price><q:le>20.00</q:le></price>`
	`  <q:filtered>true</q:filtered>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{ "$query": {`
	`  "$and": [`
	`  {"price" : { "$ge": 10.00 } },`
	`  {"price" : { "$le": 20.00 } } ],`
	`  "$filtered": true`
	`}}`

The `filtered` flag is included in the above query because you must either use filtered search or back the range queries on “price” with a range index. For details, see “How Indexing Affects Your Query” on page 234.

For details, see “Range Query” on page 211.

Note: In MarkLogic 10, a user must have both rest-reader and eval-search-string privileges to execute a QBE that uses relational comparisons (in particular, EQ, NE, LT, LE, GT, and GE) for elements that don't have a range index. The eval-search-string privilege is available in MarkLogic 9 so adopters can add the privilege to the users's role prior to rolling upgrade.

5.1.4 Query by Value or Word

When you construct a criteria on a literal value, it is an implicit value query that matches an exact value. For example, the following criteria matches only when author is “Mark Twain”. It will not match “mark twain”or “M. Twain”:

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <author>Mark Twain</author>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{`
	`  "$query": { "author": "Mark Twain" }`
	`}`

When this is not the desired behavior, you can use a word query and/or options to modify the default behavior. A word query differs from a value query in two ways: It relaxes the default exact match semantics of a value query, and it matches a subset of the value in a document.

For example, the following query matches if the `author` contains “twain”, with any capitalization, so it matches values that are not matched by the original query, such as “Mark Twain”, “M. Twain” and “mark twain”.

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <author><q:word>twain</q:word></author>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{`
	`  "$query": { "author": { "$word": "twain" } }`
	`}`

For details, see “Value Query” on page 209 and “Word Query” on page 210.

5.1.5 Search Result Customization

You can include a `response` XML element or JSON property to customize the contents of returned search results. The default search results include a highlighted snippet of matching XML elements or JSON properties. Use the `response` section of a QBE to disable snippeting, extract additional elements, or return an entire document.

For details, see "Customizing Search Results" on page 240.

5.1.6 Options for Controlling Search Behavior

The QBE grammar includes several flags and options to control your search. Flags usually have a global effect on your search, such as how to score search results. Options affect a portion of your query, such as whether or not perform an exact match against a particular XML element or JSON property value.

The following example uses the `exact` option to disable exact matches on value queries.

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <author><q:value exact="false">mark twain</q:value></author>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{`
	`  "$query": {`
	`  "author": {`
	`  "$exact": false,`
	`  "$value" : "mark twain"`
	`  }`
	`  }`
	`}`

For more details, see "Adding Options to a QBE" on page 235.

5.2 Example

This section includes an example that uses most of the query features of a QBE.

- XML Example
- JSON Example

5.2.1 XML Example

This example assumes the database contains documents with the following structure:

```
<book>
  <title>Tom Sawyer</title>
  <author>Mark Twain</author>
  <edition format="paperback">
    <publisher>Clipper</publisher>
    <pub-date>2011-08-01</pub-date>
    <price>9.99</price>
    <isbn>1613800917</isbn>
  </edition>
</book>
```

The following query uses most of the features of QBE and matches the above document. The sub-queries that are immediate children of query are implicitly AND'd together, so all these conditions must be met by matching documents.

```
<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">
  <q:query>
    <title>
      <q:value exact="false">Tom Sawyer</q:value>
    </title>
    <q:near distance="2">
      <author><q:word>mark</q:word></author>
      <author><q:word>twain</q:word></author>
    </q:near>
    <edition format="paperback">
      <q:or>
        <publisher>Clipper</publisher>
        <publisher>Daw</publisher>
      </q:or>
    </edition>
    <q:and>
      <price><q:lt>10.00</q:lt></price>
      <price><q:ge>8.00</q:ge></price>
    </q:and>
    <q:filtered>true</q:filtered>
  </q:query>
</q:qbe>
```

The following table explains the requirement expressed by each component of the query. Each of the subquery types used in this example is explored in more detail in “Understanding QBE Sub-Query Types” on page 208.

Requirement	Example Criteria

The title is “Tom Sawyer”. Exact match is disabled, so the match is not sensitive to whitespace, punctuation, or diacritics. The match is case sensitive because the value (“Tom Sawyer”) is mixed case.	`<title>`
	`  <q:value exact="false">Tom Sawyer</q:value>`
	`</title>`

The author contains the word “mark” and the word “twain” within 2 words of each other.	`<q:near distance="2">`
	`  <author><q:word>mark</q:word></author>`
	`  <author><q:word>twain</q:word></author>`
	`</q:near>`

The edition format is “paperback” and the publisher is “Clipper” or “Daw”. All the atomic values in this sub-query use exact value match semantics.	`<edition format="paperback">`
	`  <q:or>`
	`  <publisher>Clipper</publisher>`
	`  <publisher>Daw</publisher>`
	`  </q:or>`
	`</edition>`

The price is less than 10.00 and greather than or equal to 8.00.	`<q:and>`
	`  <price><q:lt>10.00</q:lt></price>`
	`  <price><q:ge>8.00</q:ge></price>`
	`</q:and>`

Use unfiltered search. This flag can be omitted if there is a range index on `price`. For details, see “How Indexing Affects Your Query” on page 234.	`<q:filtered>true</q:filtered>`

5.2.2 JSON Example

This example assumes the database contains documents with the following structure:

```
{"book": {
  "title": "Tom Sawyer",
  "author" : "Mark Twain",
  "edition": [
    { "format": "paperback",
      "publisher": "Clipper",
      "pub-date": "2011-08-01",
      "price" : 9.99,
```

```
      "isbn": "1613800917",
    }
  ]
} }
```

The following query uses most of the features of QBE and matches the above document. The sub-queries that are immediate children of query are implicitly AND'd together, so all these conditions must be met by matching documents.

```
{"$query": {
    "title": {
      "$value": "Tom Sawyer",
      "$exact": false
    },
    "$near": [
      { "author": { "$word": "mark" } },
      { "author": { "$word": "twain" } }
    ], "$distance": 2,
    "edition": {
      "format": "paperback",
      "$or" : [
        { "publisher": "Clipper" },
        { "publisher": "Daw" }
      ]
    },
    "$and": [
      {"price": { "$lt": 10.00 }},
      {"price": { "$ge": 8.00 }}
    ],
    "$filtered": true
} }
```

The following table explains the requirement expressed by each component of the query. Each of the subquery types used in this example is explored in more detail in “Understanding QBE Sub-Query Types” on page 208.

Requirement	Example Criteria

The title is “Tom Sawyer”. Exact match is disabled, so the match is not sensitive to whitespace, punctuation, or diacritics. The match is case sensitive because the value (“Tom Sawyer”) is mixed case.	`"title": {`
	`  "$value": "Tom Sawyer",`
	`  "$exact": false`
	`}`

The author contains the word “mark” and the word “twain” within 2 words of each other.	`"$near": [`
	`  { "author": { "$word": "mark" } },`
	`  { "author": { "$word": "twain" } }`
	`],`
	`"$distance": 2`

The edition format is “paperback” and the publisher is “Clipper” or “daw”. All the atomic values in this sub-query use exact value match semantics.	`"edition": {`
	`  "format": "paperback",`
	`  "$or" : [`
	`  { "publisher": "Clipper" },`
	`  { "publisher": "Daw" }`
	`  ]`
	`}`

The price is less than 10.00 and greater than or equal to 8.00.	`"$and": [`
	`  {"price": { "$lt": 10.00 }},`
	`  {"price": { "$ge": 8.00 }}`
	`]`

Use unfiltered search. This flag can be omitted if there is range index on `price`. For details, see “How Indexing Affects Your Query” on page 234.	`"$filtered": true`

5.3 Understanding QBE Sub-Query Types

The `query` portion of a QBE is composed of sub-queries. While QBE enables you to express a sub-query using syntax that closely models your documents, you should understand the query types represented by this modeling. You can express the following query types in a QBE:

- Value Query
- Word Query
- Range Query

- Composed Query
- Container Query

5.3.1 Value Query

A value query matches an entire literal value, such as a string, date, or number.

By default, an XML element or JSON property criteria represents a value query with exact match semantics:

- The value in the criteria is matched with case, diacritic, punctuation, and whitespace sensitivity enabled.
- Stemming and wildcarding are not enabled.
- The specified value must be an immediate child of the containing XML element or JSON property.
- The value in the query will not match if it is a subset of the value in a document.

For example, the following criteria only matches documents where the `author` XML element or JSON property contains exactly and only the text "Twain". It will not match author values such as "Mark Twain" or "twain".

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <author>Twain</author>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{`
	`  "$query": { "author": "Twain" }`
	`}`

You can override some of the exact match semantics with options. For example, you can disable case-sensitive matches. For details, see "Adding Options to a QBE" on page 235.

A value query can be explicit or implicit. The example above is an implicit value query. You can make an explicit value query using the `value` QBE keyword. This is useful when you want to add options to a value query. The following example is an explicit value query that uses the `case-sensitive` option.

Format	Example
XML	<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample"> <q:query> <author> <q:value case-sensitive="false">Twain<q:value> </author> </q:query> </q:qbe>

JSON	{ "$query": {
	  "author": {
	  "$value": "Twain",
	  "$case-sensitive": false
	  }
	} }

5.3.2 Word Query

A word query matches a word or phrase appearing anywhere in a text value. A word query will match a subset of a text value. By default, word queries do not use exact match semantics.

- The value in the criteria is matched with case, diacritic, punctuation, and whitespace sensitivity disabled.
- Stemmed matches are included.
- Wildcard matching is performed if wildcarding is enabled for the database.
- The specified word or phrase can occur in the value of the immediately containing XML element or JSON property, or in the value of child components.

You can use options to override some of the match semantics. For details, see “Adding Options to a QBE” on page 235.

Word queries occurring within another container, such as an XML element or JSON property that describes content in your document, match occurrences within the container. Word queries that are not in a container, such as word queries that are immediate children of the top level QBE `query` wrapper, match occurrences anywhere in a document. For details, see “Container Query” on page 214 and “Searching Entire Documents” on page 221.

The following example QBE matches if the `author` contains "twain" with any capitalization, so it matches values such as "Mark Twain", "M. Twain" and "mark twain".

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <author><q:word>twain</q:word></author>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{`
	`  "$query": { "author": { "$word": "twain" } }`
	`}`

In JSON, the value in a word query can be either a string or an array of strings. An array of values is treated as an AND-related list of word queries. For example, the following query matches documents where `author` contains word matches for "mark" and "twain". The matched values need not be array item values.

```
{
  "$query": { "author": { "$word": [ "mark", "twain" ] } }
}
```

5.3.3 Range Query

A range query matches values that satisfy a relational expression applied to a string, number, date, time, or dateTime value, such as "less than 5" or "not equal to 10". This section includes the following topics:

- JSON Property Value Range Query
- XML Element Value Range Query
- XML Element Attribute Value Range Query
- Type Conversion in Range Expressions

Note: You must either back a range query by a range index or use the `filtered` flag. For details see "How Indexing Affects Your Query" on page 234.

5.3.3.1 JSON Property Value Range Query

To construct a range query for a JSON property value, construct a JSON property with the operator name prefixed with "$" as the name and the boundary value as the value:

```
{ "$operator" : boundary-value }
```

The following example criteria tests for `format` not equal to “paperback”:

```
"format": {"$ne": "paperback" }
```

You cannot construct a range query that is constrained to match an array item.

5.3.3.2 XML Element Value Range Query

To construct a range query on an XML element value, use the following syntax, where `q` is the namespace prefix for `http://marklogic.com/appservices/querybyexample`:

```
<container>
  <q:operator>boundary-value</q:operator>
</container>
```

The following example criteria tests for publication date greather than 2010-01-01:

```
<pub-date>
  <q:gt>2010-01-01</q:gt>
</pub-date>
```

5.3.3.3 XML Element Attribute Value Range Query

To construct a range query on an XML element attribute value, prefix the operator name with “$” and put the comparison expression in the string value of the attribute on the containing element criteria:

```
<container attr="$operator value" />
```

The following example criteria tests that `@format` of `edition` does not equal “paperback”:

```
<edition format="$ne paperback" />
```

5.3.3.4 Type Conversion in Range Expressions

By default, values in range queries are treated as `xs:boolean`, `xs:double`, `xs:dateTime`, `xs:date`, or `xs:time` if castable as such, and as strings otherwise.

You can use the `xsi:type` (XML) or `$datatype` (JSON) option to force a particular type conversion; for details, see “Adding Options to a QBE” on page 235.

5.3.4 Composed Query

A composed query is one composed of sub-queries joined by a logical "operator" such `and`, `or`, `not`, or `near`. The following example matches documents where the value of `author` is "Mark Twain" or "Robert Frost".

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <q:or>`
	`  <author>Mark Twain</author>`
	`  <author>Robert Frost</author>`
	`  </q:or>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{"$query": {`
	`  "$or": [`
	`  "author": "Mark Twain",`
	`  "author": "Robert Frost"`
	`  ]`
	`} }`

The `near` operator models a `cts:near-query` and accepts an optional `distance` XML attribute or JSON property to specify a maximum acceptable distance in words between matches for the operands queries. For example, the following near query specifies a maximum distance of 2 words. The default distance is 10.

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <q:near distance="2">`
	`  <author><q:word>mark</q:word></author>`
	`  <author><q:word>twain</q:word></author>`
	`  </q:near>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{"$query": {`
	`  "$near": [`
	`  { "author": { "$word": "mark" } },`
	`  { "author": { "$word": "twain" } }`
	`  ],`
	`  "$distance": 2`
	`} }`

5.3.5 Container Query

A container query matches when sub-query conditions are met within the scope of a specific XML element or JSON property. In a container query, the relationship between the named container and XML element or JSON property names used in the sub-queries is “contained by” not merely “child of”.

A container query is implicitly defined when you use search criteria that model your document and that contain a composed query or structural sub-queries (XML element, XML attribute, or JSON property).

For example, an XML criteria such as the following defines defines a container query on `edition` because it contains an implicit value query on another element, `price`.

```
<edition><price>8.99</price></edition>
```

By contrast, the following criteria is a value query, not a container query, on `author`:

```
<author>twain</author>
```

Similarly, the following JSON criteria is a container query on `edition` because it contains an implicit value query on another property, `price`.

```
"edition":{"price": 8.99}
```

By contrast, a criteria such as the following is a value query, not a container query, on `author`.

```
"author":"twain"
```

The examples below demonstrate how a container query for `price` contained by `book` matches at multiple levels.

Query	Example Matching Documents

`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
`  <q:query>`
`  <book><price>8.99</price></book>`
`  </q:query>`
`</q:qbe>`
	`<book>`
	`  <price>8.99</price>`
	`</book>`

	`<book>`
	`  <edition>`
	`  <price>8.99</price>`
	`  </edition>`
	`</book>`

`{"$query": {`
`  "book":{"price": 8.99}`
`} }`
	`{ "book": {`
	`  "price": 8.99`
	`} }`

	`{ "book": {`
	`  "edition": {"price": 8.99 }`
	`} }`

	`{ "book": {`
	`  "edition": [ {"price": 8.99} ]`
	`} }`

A query on an XML element attribute is a container query in that the element "contains" the attribute. However, only attributes on the containing element can match. The following criteria matches `@format` only when it appears as an attribute of `edition`. It does not match occurrences of `@format` on child elements of `edition`.

```
<edition format="paperback"/>
```

The following table contains XML examples of container queries.

Description	Container Query

The element `price` is contained by the element `edition` and has a value of exactly 8.99.
 	
`price` need not be an immediate child of edition.
	`<edition>`
	`  <price>8.99</price>`
	`</edition>`

The attribute `format` is contained by the element `edition` and has the exact value "paperback".	`<edition format="paperback"/>`

The element `price` is contained by the element `edition` and has the exact value 8.99, or the element `publisher` is contained by the element `edition` and has the exact value "Fawcett".	`<edition>`
	`  <q:or>`
	`  <price>8.99</price>`
	`  <publisher>Fawcett</publisher>`
	`  </q:or>`
	`</edition>`

The following table contains JSON examples of container queries.

Description	Container Query
The JSON property `price` is contained in a property named `edition` and has the exact value 8.99. `price` need not be an immediate child of edition.	`"edition": {"price": 8.99 }`

The JSON property `price` is contained in a property named `edition` and has the exact value 8.99, or the property named `publisher` is contained in JSON property named `edition` and has the exact value "Fawcett".	`"edition": {`
	`  "$or": [`
	`  "price": 8.99,`
	`  "publisher": "Fawcett"`
	`  ]`
	`}`

The property named `edition` contains a `price` property with the value 8.99 and a `publisher` property with the value "Fawcett" anywhere in its substructure. If the value of `edition` is an array, then `price` and `publisher` must match within the same array item. Otherwise, the matches need not be within the same object or array.	`"edition": {`
	`  "$and": [`
	`  "price": 8.99,`
	`  "publisher": "Fawcett"`
	`  ]`
	`}`

The property named `edition` contains a `price` property with the value 8.99 and a `publisher` property with the value "Fawcett" anywhere in its substructure. The matches need not be within the same object or array.	`"$and": [`
	`  "edition": {"price": 8.99},`
	`  "edition": {"publisher":"Fawcett"}`
	`]`

5.4 Search Criteria Quick Reference

This section provides templates for constructing composed queries and frequently used criteria that model your documents.

- XML Search Criteria Quick Reference
- JSON Search Criteria Quick Reference
- Searching Entire Documents

5.4.1 XML Search Criteria Quick Reference

The table below provides a quick reference for constructing QBE search criteria and composed queries in XML. Use these examples as templates for your own criteria. For more details, see "QBE Structural Reference" on page 223.

The examples below assume that the namespace prefix q is bound to `http://marklogic.com/appservices/querybyexample`.

Criteria Description	Example

element *e* has value *v*	`<e>v</e>`
	
	`<e><q:value>v</q:value></e>`

the value of attribute *a* of element *e* is *v*	`<e a="v"/>`
	
	`<e a="$value v"/>`

element *e* contains word *w* anywhere in the substructure of the element content	`<e><q:word>w</q:word></e>`
the value of attribute a of element *e* includes the word *w*	`<e a="$word w"/>`
element *e* has a value greater than 5	`<e><q:gt>5</q:gt></e>`
the value of attribute a of element *e* is greater than 5	`<e a="$gt 5"/>`
element *e* exists	`<e><q:exists/></e>`
attribute *a* of element *e* exists	*not supported*

element *e1* contains element *e2* with value v; *e2* can occur anywhere in the substructure of the element content	`<e1>`
	`  <e2>v</e2>`
	`</e1>`

element *e1* contains a descendant element *e2*, and *e2* contains word w anywhere in the substructure of the element content	`<e1>`
	`  <e2><q:word>w</q:word></e2>`
	`</e1>`

element *e1* contains a descendant element *e2*, that has an attribute *a* with value *v*	`<e1>`
	`  <e2 a="v"/>`
	`</e1>`

Criteria Description	Example

element *e1* contains a descendant element *e2* that has an attribute a with word *w* in its value	`<e1>`
	`  <e2 a="$word w"/>`
	`</e1>`

a descendant of element *e* has attribute *a*	*not supported*

element *e* has value *v1* or *v2*	`<q:or>`
	`  <e>v1</e>`
	`  <e>v2</e>`
	`</q:or>`

element *e* contains word *w1* or *w2* anywhere in the substructure of the element content	`<q:or>`
	`  <e><q:word>w1</q:word></e>`
	`  <e><q:word>w2</q:word></e>`
	`</q:or>`

element *e1* contains a descendant element *e2*, and *e2* has value *v1* or *v2*	`<e1>`
	`  <q:or>`
	`  <e2>v1</e2>`
	`  <e2>v2</e2>`
	`  </q:or>`
	`</e1>`

the value of attribute *a* of element *e* has is *v1* or *v2*	`<q:or>`
	`  <e a="v1"/>`
	`  <e a="v2"/>`
	`</q:or>`

the value of attribute *a* of element *e* includes word *w1* or *w2*	`<q:or>`
	`  <e a="$word w1"/>`
	`  <e a="$word w2"/>`
	`</q:or>`

the value of attribute *a* of element *e2* that is a descendant of element *e1* is *v1* or *v2*	`<e1>`
	`  <q:or>`
	`  <e2 a="v1"/>`
	`  <e2 a="v2"/>`
	`  </q:or>`
	`</e1>`

5.4.2 JSON Search Criteria Quick Reference

The table below provides a quick reference for constructing QBE search criteria and composed queries in JSON. Where the example property name begins with *c*, the criteria represents a container query. For more details, see "QBE Structural Reference" on page 223.

This list of example criteria is not exhaustive. Additional forms are supported. For example, not all variants of explicit and implicit value queries are shown for a given criteria.

Criteria Description	Example Criteria

Property *k* with value *v*	`{ "k": "v"}`
	
	`{ "k": {"$value": "v"}}`

Property *k* containing word *w* anywhere in the substructure of the property value	`{"k":{"$word":"w"}}`
Property *k* with a value greater than 5	`{"k":{"$gt":5}}`
Property *k* exists	`{"k":{"$exists":{}}}`

A property named *c* containing a property named *k* with value *v*, where `k` can be anywhere in the substructure of *c*'s value	`{"c": {"k":"v"} }`
	
	`{"c": {`
	`  "k": {"$value": "v"}`
	`} }`

A property named *c* containing a property named *k* with a value that includes word *w*, where *k* can be anywhere in the substructure of *c*'s value	`{"c":{`
	`  "k":{"$word":"w"}`
	`}}`

A property named *k* with value *v1* or value *v2*	`{"$or":[`
	`  {"k":"v1"},`
	`  {"k":"v2"}`
	`]}`
	
	`{"$or":[`
	`  {"k": {"$value": "v1"}},`
	`  {"k": {"$value": "v2"}}`
	`]}`
	
	`{"k": {"$or": [`
	`  {"$value": "v1"},`
	`  {"$value": "v2"}`
	`]}}`

Criteria Description	Example Criteria

A property named *k* that includes word *w1* or *w2* in its value.	`{"$or":[`
	`  {"k":{"$word":"w1"}},`
	`  {"k":{"$word":"w2"}}`
	`]}`
	
	`{"k": {"$or": [`
	`  {"$word": "v1"},`
	`  {"$word": "v2"}`
	`]}}`

A property named *c* that contains a property named *k* with value *v1* or *v2* anywhere within the substructure of *c*'s value	`{"c":{"$or":[`
	`  {"k":"v1"},`
	`  {"k":"v2"}`
	`]}}`

A property named *c* that contains a property named *k* with value *v1* and a property named *k* with value *v2* anywhere within the substructure of *c*'s value.	`{"c":{"$and":[`
	`  {"k":"v1"},`
	`  {"k":"v2"}`
	`]}}`
	
	`{"c":{"$and":[`
	`  {"k": {"$value": "v1"}},`
	`  {"k": {"$value": "v2"}}`
	`]}}`

A property named *k1* with value *v1* and a property named *k2* with value *v2*. *k1* and *k2* can be in different objects.	`{`
	`  "k1":"v1",`
	`  "k2":"v2"`
	`}`
	
	`{"$and":[`
	`  {"k1":"v1"},`
	`  {"k2":"v2"}`
	`]}`
	
	`{`
	`  "k1": {"$value": "v1"},`
	`  "k2": {"$value": "v2"}`
	`}`
	
	`{"$and":[`
	`  {"k1": {"$value": "v1"}},`
	`  {"k2": {"$value": "v2"}}`
	`]}`

5.4.3 Searching Entire Documents

This section describes how to construct a query that matches words or phrases anywhere in a document, rather than constraining the match to occurrences in a particular XML element, XML attribute, or JSON property.

A word query has document scope if it is not contained in an XML element or JSON property criteria. For example, a word query that is an immediate child of the top level query element, or one that is a child at any depth of a hierarchy of composed queries (`and`, `or`, `not`, `near`). This also applies to the implicit `and` query that joins the immediate children of `query`.

For example, the following query matches all documents containing the phrase "moonlight sonata":

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <q:word>moonlight sonata</q:word>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{"$query": {`
	`  "$word": "moonlight sonata"`
	`} }`

The following example matches all documents containing either the phrase "moonlight sonata" or the word "sunlight".

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <q:or>`
	`  <q:word>moonlight sonata</q:word>`
	`  <q:word>sunlight</q:word>`
	`  </q:or>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{"$query": {`
	`  "$or": [`
	`  {"$word": "moonlight sonata"},`
	`  {"$word": "sunlight"}`
	`  ]`
	`} }`

An AND relationship between words and phrases can be either explicit or implicit. The following example queries match all documents contains both the phrase “moonlight sonata” and the word “sunlight”:

Format	Example
XML	<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample"> <q:query> <q:word>moonlight sonata</q:word> <q:word>sunlight</q:word> </q:query> </q:qbe> <q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample"> <q:query> <q:and> <q:word>moonlight sonata</q:word> <q:word>sunlight</q:word> </q:and> </q:query> </q:qbe>

JSON	{"$query": [
	  {"$word": "moonlight"},
	  {"$word": "sunlight"}
	] }
	
	{"$query": {
	  "$and": [
	  {"$word": "moonlight"},
	  {"$word": "sunlight"}
	  ]
	} }

5.5 QBE Structural Reference

This section describes the syntax and semantics of a QBE. The following topics are covered:

- Top Level Structure
- Query Components
- Response Components
- XML-Specific Considerations
- JSON-Specific Considerations

5.5.1 Top Level Structure

At the top level, a QBE must contain a `query` and can optionally contain a `response` and/or a `format` flag. A QBE has the following top level parts:

- `query`: Define matching document requirements in the `query`.
- `response`: Customize your search results in the `response`; if there is no `response`, the default search response is returned.
- `format`: Use the `format` flag to override the interpretation of bare names as JSON property names or XML element names in no namespace, based on the query format. For details, see "Scoping a Search by Document Type" on page 245.
- `validate`: Use the validate flag to enable query validation before evaluating the search. The default is no validation, which can result in surprising search results if your QBE contains errors. However, validation has a performance cost, so it is best used only for debugging during development.

The following table outlines the top level of a QBE:

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  *search parameters*`
	`  </q:query>`
	`  <q:response>`
	`  *search result customizations*`
	`  </q:response>`
	`  <q:format>*xml-or-json*</q:format>`
	`  <q:validate>*true-or-false*</q:validate>`
	`</q:qbe>`

JSON	`{`
	`  "$query": {`
	`  *search parameters*`
	`  },`
	`  "$response": {`
	`  *search result customizations*`
	`  },`
	`  "$format": *xml-or-json*,`
	`  "$validate": *boolean*`
	`}`

A `query` contains one or more XML elements or JSON properties defining element or property criteria or composed queries. Use criteria to model document structure. Use a composed query to logically join sub-queries using operators such as `and`, `or`, `not`, and `near`.

In XML, a QBE has a `qbe` wrapper element. Element and attribute names pre-defined by the QBE grammar, such as `qbe`, `query`, and `word`, are in the namespace `http://marklogic.com/appservices/querybyexample`. All other element and attributes names represent element and attribute names in your documents. For details, see "Managing Namespaces" on page 228

In JSON, all property names pre-defined by the QBE grammar have a "$" prefix, such as `$query` or `$word`. Any property name without a "$" prefix represents a property in your documents. For details, see "Property Naming Convention" on page 230.

You will not usually need to set the `format` flag. You only need to set the `format` flag to use a JSON QBE to match XML documents, or vice versa. For details, see "Scoping a Search by Document Type" on page 245.

5.5.2 Query Components

The table below describes the components of the `query` portion of a QBE. Additional format-specific details are covered in "XML-Specific Considerations" on page 228 and "JSON-Specific Considerations" on page 230.

Component Type	XML Local Name	JSON Property Name	Description
query	`query`	`$query`	Defines the search criteria. Required.
criteria	*`your element name`*	*`your property name`*	Defines search criteria to apply within the scope of an XML element or JSON property in your documents. The name corresponds to an element or property in the content to be matched by the query. If the criteria wraps a composed query or another criteria, then it represents a container query. Otherwise, it represents a value, word, or range query.

composed query	`and`
	
	`or`
	
	`not`
	
	`near`
		`$and`
		
		`$or`
		
		`$not`
		
		`$near`
			Defines a composed query that joins sub-queries using logical "operators".
			
			The `near` operator accepts an optional `distance` XML attribute or JSON property:
			• `<q:near distance="5">...</q:near>`
			• `{ "$near": "$distance":5, [..] }`

Component Type	XML Local Name	JSON Property Name	Description

range query	`lt, le`
	`gt, ge`
	`eq, ne`
		`$lt, $le`
		`$gt, $ge`
		`$eq, $ne`
			Defines a relational "expressions" on a value in an XML element, XML attribute, or JSON property.

modifier	`value`
	`word`
	`exists`
		`$value`
		`$word`
		`$exists`
			A modifier on a value that defines how to match that value: with a value query (the default with no modifier), with a word query, or with an existence test.

flag	`filtered` `score`	`$filtered` `$score`	Flags are modifiers of search behavior. Use the boolean `filtered` flag to control whether the search is filtered or unfiltered (default). For more details, see "How Indexing Affects Your Query" on page 234. Use the score flag to override the search result scoring function. Allowed values: `logtf`, `logtfidf`, `random`, `simple`, `zero`. Default: `logtfidf`. For details, see "Relevance Scores: Understanding and Customizing" on page 422.
options			Use options to fine tune your search criteria and results. For details, see "Adding Options to a QBE" on page 235.

The following table summarizes where each component type can be used. Options are covered in "Adding Options to a QBE" on page 235.

Component Type	Contains	Contained By
query	One or more criteria, composed queries, and the `filtered` or `score` flags	`qbe` (XML) root object (JSON)
criteria	• Nothing (empty); or • One value; or • One word, (explicit) value, or range query; or • One or more criteria or composed queries	`query`, composed query, criteria
composed query (`and`, `or`, etc.)	One or more criteria, composed queries, or word queries. Word queries are only permitted when the composed query is an immediate child of `query`.	`query`, composed query, or criteria
range query (`lt`, `gt`, etc.)	a value	criteria
word or value query	a value (string, number, date, time, dateTime)	`word`: `query`, criteria, or composed query `value`: criteria; composed query contained by a criteria
`exists`		criteria
flag		`query`

5.5.3 Response Components

You can use the `response` portion of a QBE to customize the format of your search results. The following table describes the components of a `response`. A `response` is optional, and can only occur at the top level of a QBE, as a sibling of `query`.

A response can contain the following formatter components:

XML Local Name	JSON Property Name	Description
`snippet`	`$snippet`	A `snippet` element controls what is returned for search matches. You can specify elements to prefer if they have a match and/or set a policy (`default`, `document`, `none`) for what to show.
`extract`	`$extract`	An `extract` element supplements a `snippet` by listing XML elements or JSON properties to extract from matching documents, whether or not a match occurs in the listed elements or property.

For details, see “Customizing Search Results” on page 240.

5.5.4 XML-Specific Considerations

This section covers structural and semantic details you should know when constructing a QBE in XML.

- Managing Namespaces
- Querying Attributes

5.5.4.1 Managing Namespaces

Use the namespace `http://marklogic.com/appservices/querybyexample` for all pre-defined element names in the QBE grammar, such as `qbe`, `query`, and `word`. This namespace distinguishes the structural parts of the query from criteria elements that model your documents. You define this namespace at the top level of your QBE. For example:

```
<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">
   ...
</q:qbe>
```

Define namespaces required by your element criteria on the criteria or any enclosing element container. You cannot bind the same namespace prefix to different namespaces within a QBE.

The following example demonstrates declaring user-defined namespaces on the root `qbe` element, on a containing element, and on an element criteria.

```
<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample"
       xmlns:ns1="http://marklogic.com/example1">
```

```
  <q:query xmlns:ns2="http://marklogic.com/example2">
    <ns1:author xmlns="http://marklogic.com/example">
      Mark Twain
    </ns1:author>
    <ns2:edition format="paperback"/>
    <title xmlns="http://marklogic.com/example3">Tom Sawyer</title>
  </q:query>
</q:qbe>
```

5.5.4.2 Querying Attributes

To query an element attribute, create an element criteria that contains the attribute. The following example represents a value query for the attribute `edition/@format` with a value of "paperback".

```
<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">
  <q:query>
    <edition format="paperback"/>
  </q:query>
</q:qbe>
```

The value of the attribute can be an implicit value query, as in the example above, or an explicit value, word, or range query. To create a word, range, or explicit value query on an attribute, use the following template for the attribute value, where *keyword* is a modifier (`word` or `value`) or comparator (`lt`, `gt`, etc.).

```
$keyword value
```

For example, the following QBE represents a range query on the attribute `edition/@price`.

```
<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">
  <q:query>
    <edition price="$lt 9.00"/>
  </q:query>
</q:qbe>
```

You cannot use the `exists` modifier in an attribute value.

Multiple attributes on an element criteria are AND'd together. For example, the following QBE uses a range query on `edition/@price` and a word query on `edition/@format` paperback to find all paperback editions with a price less than 9.00.

```
<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">
  <q:query>
    <edition price="$lt 9.00" format="$word paperback" />
    <q:filtered>true</q:filtered>
  </q:query>
</q:qbe>
```

You cannot use range, word, or value query options such as `exact`, `min-occurs`, or `score-function` on attribute criteria. If you need this level of control over an attribute query, use a structured query instead of QBE. For details, see "Searching Using Structured Queries" on page 74.

5.5.5 JSON-Specific Considerations

This section covers structural and semantic details you should know when constructing a QBE in JSON. The following topics are covered:

- Property Naming Convention
- Matching Array Items
- Searching Array and Object Containers
- Constructing a QBE with the Node.js QueryBuilder

5.5.5.1 Property Naming Convention

In JSON, all pre-defined JSON property names in the QBE grammar have a "`$`" prefix to distinguish them from names that occur in your documents. For example, the property name for the `query` part of a JSON QBE is `$query`.

If your documents include property names that start with "$", the names in your content can conflict with the pre-defined property names. In such a case, you must use a structured query instead of QBE. For details, see "Searching Using Structured Queries" on page 74.

For a list of pre-defined property names, see "Query Components" on page 225 and "Response Components" on page 227.

5.5.5.2 Matching Array Items

QBE does not distinguish between values contained in an array and values not contained in an array. For example, the following query:

```
{ "$query": {"k": ["v"]} }
```

Matches both of the following documents:

```
{ "k": "v" }
```

```
{ "k": ["v"] }
```

Also, the query is exactly equivalent to the following query that does not use array syntax:

```
{ "$query": {"k": "v"} }
```

Consequently, you cannot use QBE to match a property whose value is exactly and only a specified array value.

When you use array syntax and include multiple values, an AND relationship is implied between the values. For example, the following two queries are equivalent:

```
{"$query":
  {"k": ["v1", "v2"]}
}

{"$query": {
  "$and": [
    {"k": "v1"},
    {"k": "v2"}
  ]
}}
```

Both queries will match all of the following documents:

```
{ "k": ["v1", "v2"] }

{ "k": ["v1", "v2", "v3"] }

{ "c": [{"k": "v1"}, {"k": "v2"}] }

{"c": {"k": "v1", "c2": {"k": "v2"}}
```

5.5.5.3 Searching Array and Object Containers

The type of query represented by a criteria property that names a JSON property in your content depends on the type of value in the property. If the value is an object or a composed query, then it represents a container query. Otherwise, it is a value, word, or range query. You should understand how container queries apply to searching JSON documents..

A criteria property expresses “Match a JSON property named *k* whose value meets these conditions” if the value is a literal value, or a word, value, or range query. Such a criteria is not a container query. The table below illustrates these forms.

Criteria Template	Example Criteria	Description
name : *value*	{ "price" : 8.99 }	Match a property named "price" whose value is 8.99

name : {
  word-or-value : *value*
}
	{ "title" : {
	  "$word" : "sawyer"
	} }
		Match a property named "title" whose value includes "sawyer"

name : {
  relational-op : *value*
}
	{ "price" : {
	  "$lt" : 9
	} }
		Match a property named "price" whose value is less than 9

A criteria property in which the value is an object or a composed query is a container query. Such a query says "Match a property named *c* that contains a value meeting these conditions *anywhere in its substructure*." The table below illustrtates these forms.

Criteria Template	Example Criteria	Description
name : *object*	{"edition": {"price" : 8.99 } }	Match a JSON property named "price" whose value is 8.99 and that is contained somewhere within a property named "edition". The value can occur as an array item.

name : {
  logical-op : [
  sub-query+
  ]
}
	{"edition" : {
	  "$or": [
	  {"format": "paperback"},
	  {"format": "hardback"}
	  ]
	} }
		Match a JSON property named "format" that is contained somewhere within a property named "edition" and whose value is "paperback" or "hardback". The values can occur as array items.

Since a container query always matches its sub-queries anywhere within the container substructure, you cannot construct a JSON QBE that matches "a container with property name *k* whose value is exactly and only this object".

The table below provides example documents matched by a value query and several kinds of container query. The matched document examples are not exhaustive. Each query is annotated with a textual description of what the criteria asserts about matching documents. For more examples, see "JSON Search Criteria Quick Reference" on page 220.

QBE	Matches
{"$query": { "k": "v" } } Property *k* has value "*v*"	{ "k": "v" } { "k": ["v"] }

QBE	Matches
{"$query": {"k": ["v1", "v2"]} } Property *k* has value "*v1*" and value "v2".	{ "k": ["v1", "v2"] } { "c": [{"k": "v1"}, {"k": "v2"}] } {"c": {"k": "v1", "c2": {"k": "v2"}}

{"$query":
  {"c": {"k": "v"}}
}
 	
Property *c* contains a property *k* that has value "*v*", where *k* can occur anywhere in *c*'s substructure.
	{ "c" : {"k" : "v" } }
	
	{ "c" :
	  {"c2": {"k": "v" }}
	}

5.5.5.4 Constructing a QBE with the Node.js QueryBuilder

This topic describes how to use the information in this chapter in conjunction with the Node.js Client API.

The Node.js Client API enables you to construct a QBE using the `QueryBuilder.byExample` function. The parameters of `byExample` correspond to the criteria within the `$query` portion of a raw QBE, expressed as a JavaScript object. For example, the table below shows a QBE example from elsewhere in this chapter and the equivalent `QueryBuilder.byExample` call.

Raw QBE	QueryBuilder.byExample

{"$query": {
  "author": {"$word": "twain"},
  "$filtered": true
}}
	qb.byExample({
	  author: {$word: 'twain'},
	  $filtered: true
	})

You can also supply the entire `$query` portion of a QBE to `byExample` as a JavaScript object. For example:

```
qb.byExample(
  { $query: {
    author: {$word: 'twain'},
    $filtered: true
  }
)
```

However, you cannot specify `$response` portions of a raw QBE through `QueryBuilder.byExample`. Response customization is still available through `QueryBuilder.extract` and `QueryBuilder.snippet`.

For details, see Querying Documents and Metadata in the *Node.js Application Developer's Guide*.

5.6 How Indexing Affects Your Query

You do not have to define any indexes to use QBE. This allows you to get started with QBE quickly. However, indexes can significantly improve the performance of your search.

Unless your database is small or your query produces only a small set of pre-filtering results, you should define an index over any XML element, XML attribute, or JSON property used in a range query. To configure an index, see Range Indexes and Lexicons in *Administrator's Guide*.

If your QBE includes a range query, you must either have an index configured on the XML element, XML attribute, or JSON property used in the range query, or you must use the `filtered` flag to force a filtered search.

A filtered search uses available indexes, if any, but then checks whether or not each candidate meets the query requirements. This makes a filtered search accurate, but much slower than an unfiltered search. An unfiltered search relies solely on indexes to identify matches, which is much faster, but can result in false positives. For details, see Fast Pagination and Unfiltered Searches in *Query Performance and Tuning Guide*.

In the absence of a backing index, a range query cannot be used with unfiltered search. To enable filtered search, set the `filtered` flag to true in the `query` portion of your QBE, as shown in the following example:

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <author><q:word>twain</q:word></author>`
	`  **<q:filtered>true</q:filtered>**`
	`  </q:query>`
	`</q:qbe>`

JSON	`{`
	`  "$query": {`
	`  "author": {"$word": "twain"},`
	`  **"$filtered": true**`
	`  }`
	`}`

5.7 Adding Options to a QBE

Options give you fine grained control over a QBE. Most options are associated with a value, word, or range query.

- Specifying Options in XML
- Specifying Options in JSON
- Option List
- Using Persistent Query Options

5.7.1 Specifying Options in XML

In an XML QBE, an option is an attributes of the predefined QBE element it modifies, such `<q:lt/>`, `<q:word/>` or `<q:value>`. The following query demonstrates use of the `exact` option on a value query.

```
<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">
  <q:query>
    <author><q:value exact="false">mark twain</q:value></author>
  </q:query>
</q:qbe>
```

You cannot apply options to queries on attributes because the range, word, or value query is embedded in the attribute value. For example, you cannot add a `case-sensitive` option to the following attribute word query:

```
<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">
  <q:query>
    <edition @format="$word paperback"/></edition>
  </q:query>
</q:qbe>
```

If you need such control over an element attribute query, you should use a structured or combined query.

5.7.2 Specifying Options in JSON

In a JSON QBE, an option is a sibling of the QBE object it modifies, such as a value, word, or range query. Option names always have a “$” prefix.

The following example query uses the `exact` option to modify a value query by including it as a JSON property at the same level as the `$value` object:

```
{
  "$query": {
    "author": {
      "$exact": false,
```

```
            "$value": "mark twain"
          }
        }
      }
```

5.7.3 Option List

The following table describes the options available for use in a QBE. The MarkLogic Server Search API supports additional options through other query formats, such as string or structured query, and through the use of persistent query options. For details, see "Search Customization Using Query Options" on page 381.

Option Attribute or Property Name	Description
`case-sensitive`	Whether or not to perform a case-sensitive match. Default: false if the text to match is all lower case, true otherwise. Value type: boolean. Usable with: word or value query. For details, see `cts:word-query` or `cts:value-query`.
`diacritic-sensitive`	Whether or not to perform a diacritic-sensitive match. Default: Depends on context: false if the text to match contains no diacritics, true otherwise. Value type: boolean. Usable with: word or value query. For details, see `cts:word-query` or `cts:value-query`.
`punctation-sensitive`	Whether or not to perform a punctuation-sensitive match. Default: depends on context: false if the text to match contains no punctuation, true otherwise. Value type: boolean. Usable with: word or value query. For details, see `cts:word-query` or `cts:value-query`.
`whitespace-sensitive`	Whether or not to perform a whitespace-sensitive match. Default: false. Value type: boolean. Usable with: word or value query. For details, see `cts:word-query` or `cts:value-query`.
`stemmed`	Whether or not to use stemming. Default: Depends on context and database configuration; for details, see `cts:word-query`. Value type: boolean. Usable with: word or value query. For details, see `cts:word-query` or `cts:value-query`.

Option Attribute or Property Name	Description
`exact`	Whether to perform an exact match or use the builtin context-sensitive default behaviors for the `*-sensitive` options. When true, `exact` is shorthand for case-sensitive, diacritic sensitive, punctuation-sensitive, whitespace-sensitive, unstemmed, and unwildcarded. Default: true for value and range query, false for word query. Value type: boolean. Usable with: word or value query.
`score-function`	Use the selected scoring function. Allowed values: `linear`, `reciprocal`. Usable with: range query. For details, see "Including a Range or Geospatial Query in Scoring" on page 430.
`slope-factor`	Apply the given number as a scaling factor to the slope of the scoring function. Default: 1.0. Value type: double. Usable with: range query. For details, see "Including a Range or Geospatial Query in Scoring" on page 430.
`min-occurs`	The minimum number of occurrences required. If there are fewer occurrences, the fragment does not match. Default: 1. Value type: integer. Usable with: range, word, or value query. For details, see `cts:word-query`.
`max-occurs`	The maximum number of occurrences required. If there are more occurrences, the fragment does not match. Default: Unbounded. Value type: integer. Usable with: range, word, or value query. For details, see `cts:word-query`.
`lang`	The language under which to interpret the content. The option value is case-insensitive. Allowed values: An ISO 639 language code. Default: The default language configured for the database. Usable with: `query`; range, word, or value query. In XML it can also appear on the `qbe` element. In JSON, it can appear as a top level property.
`weight`	A weight for this query. Higher weights move search results up in the relevance order. Allowed values: less than or equal to 64 and greater tha or equal to -16 (between -16 and 64). Default: 1.0. Usable with: a word or value query, or a range query that is backed by a range index. For details, see `cts:word-query`, `cts:value-query`, or `cts:element-range-query`.

Option Attribute or Property Name	Description
`constraint`	The name of a range, values, or word constraint specified for the same XML element or JSON property in persisted query options associated with the search. Usable with: range, word, or value query. For details, see "Using Persistent Query Options" on page 238.
`@xsi:type (XML)` `$datatype (JSON)`	The `xsi:type` to which to cast the value supplied in a range query. Default: Values are treated as `xs:boolean`, `xs:double`, `xs:date`, `xs:dateTime`, or `xs:time` if castable as such, and as string otherwise. Usable with: range query.

5.7.4 Using Persistent Query Options

The REST and Java APIs enable you to install persistent query options on your REST instance and apply them to subsequent searches. You can also use persistent query options with the Node.js Client API, but the API has no facility for creating and maintaining the persistent options.

Using persistent query options with a QBE allows you to use options not supported directly by the QBE grammar. Using persistent options with a QBE also allows you to define global options to apply throughout your query, such as making all word queries case-sensitive instead of specifying the `case-sensitive` option on each word query in your QBE.

Query options applied through through the `constraint` option override options specified inline on a QBE.

You can apply persistent query options to a QBE using the `constraint` option. To use this option:

1. Install named, persistent query options following the directions appropriate for the API you are using.

2. Specify the name of a constraint defined in the persistent options from Step 1 as the value of a `constraint` option on a word, value, or range query in your QBE. See the example, below.

3. When you execute a search with your QBE, associate the persistent query options from Step 1 with your search in the manner prescribed by the client API (REST, Java, or Node.js).

For details on defining, installing and using persistent query options, see Configuring Query Options in *REST Application Developer's Guide* or Query Options in *Java Application Developer's Guide*.

The pre-defined constraint named by the `constraint` option should match the type of query to which it is applied. That is, name a range constraint for a range query, a value constraint for a value query, and a word constraint for a word query.

The following example pre-defines a word constraint called "w-t" that gives weight 2.0 to matches in a `title` XML element or JSON property, and then applies it to a QBE that contains a word query on `title`. This enables word queries on `title` a default weight that can be overridden by omitting the `constraint` option.

If the folloiwng persistent query options are installed specified as a parameter to the search performed with the QBE:

XML Options	JSON Options

<search:options
xmlns:search="http://marklogic.com/appservices/search">
<search:constraint name="w-t">
<search:word>
<search:element name="title" ns=""/>
<search:weight>2.0</search:weight>
</search:word>
</search:constraint>
</search:options>
	{"options": {
	"constraint": [{
	"name": "w-t",
	"word": {
	"json-property": "title",
	"weight": 2
	}
	}]
	} }

Then the following QBE applies the "w-t" option to a word query on `title` to give weight 2.0 to matches in a `title` element.

XML	JSON

<q:qbe
xmlns:q="http://marklogic.com/appservices/querybyexample">
<q:query>
<title>
<q:word **constraint="w-t"**>sawyer</q:word>
</title>
</q:query>
</q:qbe>
	{ "$query": {
	"title": {
	"$word": "sawyer",
	"$constraint": "w-t"
	}
	} }

5.8 Customizing Search Results

You can include a `response` XML element or JSON property to customize the contents of returned search results. You can modify or supplement the default search results using the `snippet` and `extract` formatters in the `response` section of a QBE.

This section covers the following topics:

- When to Include a Response in Your Query
- Using the snippet Formatter
- Using the extract Formatter

5.8.1 When to Include a Response in Your Query

Add an optional `response` section to a QBE to do one or more of the following:

- Return matching documents instead of snippets. (`snippet`)
- Return only information about the document and the match, such as database URI, document format, and relevance score. (`snippet`)
- Specify XML elements or JSON properties to prefer when constructing snippets. (`snippet`)
- Specify XML elements or JSON properties to extract from matching documents, whether or not the match occurs within those elements or properties. (`extract`)

Advanced customization is available using result decorators, transforms, and persistent query options. For details, see Customizing Search Results in *REST Application Developer's Guide* or Transforming Search Results in *Java Application Developer's Guide*.

5.8.2 Using the snippet Formatter

Use `snippet` to control what, if anything, is included in the snippet portion of a search match and to identify preferred XML elements or JSON properties to include a snippet. The default snippet is a small text excerpt with the matching text tagged for highlighting. The following table contains an excerpt of the snippet section of a search response generated with the default policy.

Format	Default Snippet Example

XML	`<search:response ...>`
	`  <search:result ...>`
	`  **<search:snippet>**`
	`  **<search:match**`
	`  **path="fn:doc("/books/sawyer.xml")/book">**`
	`  **<search:highlight>Mark Twain</search:highlight>**`
	`  **</search:match>**`
	`  **</search:snippet>**`
	`  </search:result>`
	`  ...`
	`</searchresponse>`

JSON	`{`
	`  ...`
	`  "results": [ {`
	`  ...`
	`  "matches": [ {`
	`  "path": "fn:doc(\"/books/sawyer.json\")/*:json/*:book/*:author",`
	`  **"match-text": [ { "highlight": "Mark Twain" } ]**`
	`  } ]`
	`  } ],`
	`  ...`
	`}`

The snippet formatter has the following form:

XML	JSON

`<q:response>`
`  <q:snippet>`
`  <q:*policy*/>`
`  *preferred-element*`
`  </q:snippet>`
`</q:response>`
	`{ "$response": {`
	`  "$snippet": {`
	`  *policy*: {},`
	`  *preferred-property*: {}`
	`  },`
	`} }`

The *policy*, *preferred-element*, and *preferred-property* are optional.

The snippeting policy controls whether or not snippets are included in the output and whether to include a small text excerpt (default) or the entire document when snippets are enabled. Use one of the following element or property names for policy.

XML	JSON	Description
`default`	`$default`	Include a small excerpt of the text around the matching terms, with the matched text tagged for highlighting.
`document`	`$document`	Return the entire document.
`none`	`$none`	Do not include any snippets.

The following example disables snippet generation by setting the snippet policy to `none`. In JSON, specify an empty object value for the policy property.

XML	JSON

`<q:response>`
`  <q:snippet>`
`  <q:none/>`
`  </q:snippet>`
`</q:response>`
	`{ "$response": {`
	`  "$snippet": {`
	`  "$none": {}`
	`  }`
	`} }`

You can also specify one or more XML element or JSON property names to be preferred when generating snippets. For example, if you specify a preference for the `title` element or property, and both `title` and `author` contain a match, the snippet is generated from the match in `title`. In JSON, specify the preferred property with an empty object value.

XML	JSON

`<q:response>`
`  <q:snippet>`
`  <title/>`
`  </q:snippet>`
`</q:response>`
	`{ "$response": {`
	`  "$snippet": {`
	`  "title": {}`
	`  }`
	`} }`

5.8.3 Using the extract Formatter

Use the `extract` formatter to specify additional XML elements or JSON properties to include in the search output. If snippets are included, the extracted components supplement any snippet in a match, rather than replacing it.

XML	JSON

`<q:response>`
`  <q:extract>`
`  <your-element/>`
`  </q:extract>`
`</q:response>`
	`{ "$response": {`
	`  "$extract": {`
	`  "your-property-name": {}`
	`  }`
	`} }`

For example, the following response says to extract the `title` and `author` from a matching document. The `title` and `author` need not contain the matching terms or values.

XML	JSON

`<q:response>`
`  <q:extract>`
`  <title/>`
`  <author/>`
`  </q:extract>`
`</q:response>`
	`{ "$response": {`
	`  "$extract": {`
	`  "title": {},`
	`  "author": {}`
	`  }`
	`} }`

Extracted elements or properties go into the `metadata` section of the enclosing match. For an example, see “Example: Search Customization” on page 244.

5.8.4 Example: Search Customization

The following QBE modifies the search results to exclude snippets and to extract the `title` XML element or JSON property into the search result `metadata` section.

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <author>Mark Twain</author>`
	`  </q:query>`
	`  <q:response>`
	`  <q:extract><title/></q:extract>`
	`  <q:snippet><q:none/></q:snippet>`
	`  </q:response>`
	`</q:qbe>`

JSON	`{`
	`  "$query": {`
	`  "author": "Mark Twain"`
	`  },`
	`  "$response": {`
	`  "$snippet": { "$none": {} },`
	`  "$extract": { "title": {} }`
	`  }`
	`}`

The following table shows the default output and the modified output produced by the above query.

Format	Default Output	Customized Output
XML	<search:response **snippet-format="snippet"** total="1" start="1" page-length="10" ...> <search:result index="1" uri="/books/sawyer.xml" ...> **<search:snippet>** **<search:match ...>** **<search:highlight>** **Mark Twain** **</search:highlight>** **</search:match>** **</search:snippet>** </search:result> ... </search:response>	<search:response **snippet-format="empty-snippet"** total="1" start="1" page-length="10" ..> <search:result index="1" uri="/books/sawyer.xml" ...> **<search:snippet/>** **<search:metadata>** **<title>Tom Sawyer</title>** **</search:metadata>** </search:result> ... </search:response>

JSON	{
	"snippet-format": "snippet",
	"total": 1,
	"start": 1,
	"page-length": 10,
	"results": [{
	"index": 1,
	"uri": "/books/sawyer.json",
	"matches": [{
	"path": ...,
	"match-text": [{
	"highlight": "Mark Twain"
	}]
	}]
	}],
	...
	}
		{
		"snippet-format":
		"empty-snippet",
		"total": 1,
		"start": 1,
		"page-length": 10,
		"results": [{
		"index": 1,
		"uri": "/books/sawyer.json",
		...,
		"matches": [],
		"metadata": [{
		"title": "Tom Sawyer"
		}]
		}],
		...
		}

5.9 Scoping a Search by Document Type

This section describes how the treatment of bare names in a QBE affects the type of documents matched by the query.

A bare name in a JSON QBE is a JSON property name that does not include a "$" prefix. A bare name in an XML QBE is an element name in no namespace.

By default, the interpretation of bare names matches your query format. That is, bare names in a JSON QBE represent JSON property names in content, and bare names in an XML QBE represent element names in content that are in no namespace. The net effect is that an XML QBE only matches XML documents, and a JSON QBE only matches JSON documents by default.

Use the `format` option to override the default behavior, as shown in the following example:

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  **<q:format>json</q:format>**`
	`  <q:query>...</q:query>`
	`</q:qbe>`

JSON	`{`
	`  **"$format": "xml"**,`
	`  "$query": {...}`
	`}`

5.10 Converting a QBE to a Combined Query

The primary use case for QBE is rapid prototyping of queries during development. For best performance and access to the full set of Search API capabilities, you should eventually convert your QBE to a combined query. A combined query is a lower level representation that combines a structured query and query options.

The REST and Java APIs include an interface for generating a combined query from a QBE. For details, see the following:

- Convert a QBE to a Combined Query in *Java Application Developer's Guide*
- Generating a Combined Query from a QBE in *REST Application Developer's Guide*
- "Searching Using Structured Queries" on page 74

5.11 Validating a QBE

You can set the `validate` flag to true to perform query validation before evaluating a QBE. When validation is enabled, if you submit a QBE that contains errors, MarkLogic reports the errors and does not perform the search. If your query does not contain errors, the search proceeds as usual.

Performing query validation on every search can be expensive, so you should not enable validation in production. It is best used for debugging during development.

The following example is a QBE with validation enabled:

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <author>Mark Twain</author>`
	`  </q:query>`
	`  <q:validate>true</q:validate>`
	`</q:qbe>`

JSON	`{`
	`  "$query": {`
	`  "author": "Mark Twain"`
	`  },`
	`  "$validate": true`
	`}`

6.0 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
- Creating a Query From Search Text With cts:parse
- 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:
- Serializations of cts:query Constructors
- Example: Creating a cts:query Parser

6.1 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:

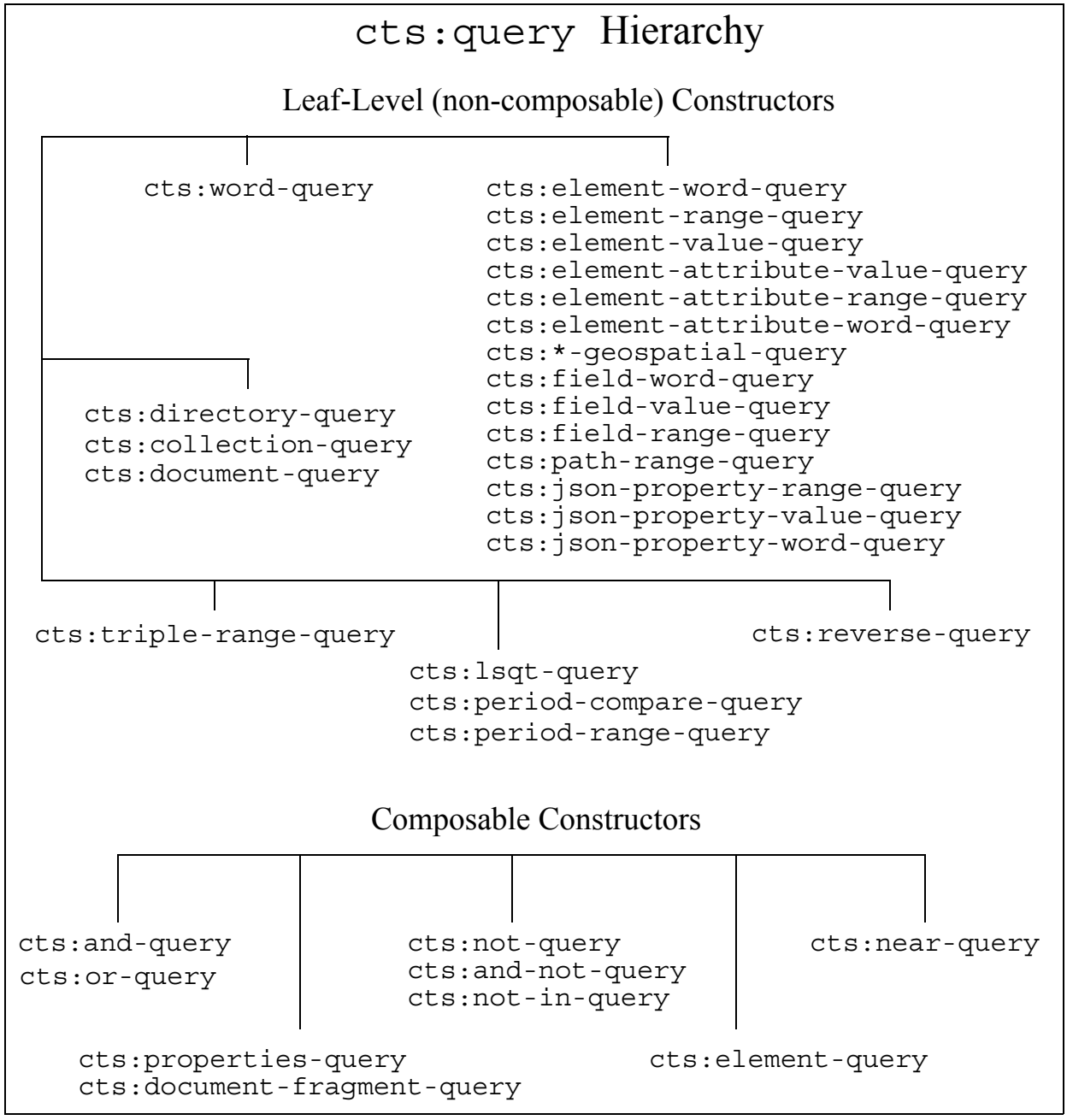
- cts:query Hierarchy
- Use to Narrow the Search
- Understanding cts:element-query
- Understanding cts:element-word-query
- Understanding Field Word and Value Query Constructors
- Understanding the Range Query Constructors
- Understanding the Reverse Query Constructor
- Understanding the Geospatial Query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

6.1.1 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. The diagram shows most of the available constructors, but not necessarily all of them.



Equivalent constructors exist for Server-Side JavaScript. For example, the JavaScript built-in `cts.andQuery` is equivalent to the XQuery built-in `cts:and-query` in the diagram above.

The remainder of this chapter goes into more detail on combining constructors.

6.1.2 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.

6.1.3 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.

6.1.4 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")
```

6.1.5 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 450.

6.1.6 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 459.

6.1.7 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 635.

6.1.8 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 476.

6.1.9 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`* option, 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 751.

6.2 Creating a Query From Search Text With cts:parse

This section describes how to create a cts:query from a simple search string using the `cts:parse`. XQuery function or the `cts.parse` Server-Side JavaScript function. The following topics are covered:

- String Query Overview
- Grammar Components and Operators
- Including Options and Weights in Query Text

- Binding a Tag to a Reference, Field, or Query Generator
- Customizing Naked Term Handling With Bindings

6.2.1 String Query Overview

A *string query* is a plain text search string ("query text") composed of terms, phrases, and operators that can be easily composed by end users typing into an application search box. For example, "cat AND dog" is a string query for finding documents that contain both the term "cat" and the term "dog".

You can use the `cts:parse` XQuery built-in function or the `cts.parse` Server-Side JavaScript built-in function to convert such a string query into a cts:query (XQuery) or cts.query (JavaScript). Use the resulting query in any interface that accepts a cts query, such as the `cts:search` XQuery function, the `cts.search` JavaScript function, and several JSearch API interfaces.

The following example uses `cts:parse` to match documents that contain the term "cat" and the term "dog".

Language	Example
XQuery	`cts:search(fn:doc(), cts:parse("cat AND dog"))`

JavaScript	`// with cts.search`
	`cts.search(cts.parse('cat AND dog'))`
	
	`// with JSearch`
	`import * as jsearch from '/MarkLogic/jsearch.mjs';`
	`jsearch.documents()`
	`  .where(cts.parse('cat AND dog'))`
	`  .result()`

The string query grammar supported by `cts:parse` and `cts.parse` enables users to compose complex queries. Adjacent terms, phrases and sub-expressions are implicitly AND'd together.

The following are some examples of queries that work with `cts:parse` and `cts.parse` "out of the box":

- `(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`

You can also bind a tag name to an index reference, lexicon reference, or field name. When such a tag name appears in a query string, it parses to a word, value, or range query that is scoped to the bound entity.

For example, binding the tag "color" to a `cts:reference` to a JSON property named `bodyColor` enables users to create query text like the following:

- `color:red`

  Match documents where the value of the `bodyColor` contains the word "red"

- `color NE blue`

  Match documents where the value of `bodyColor` is not "blue"

Without the binding, the above examples are just word queries that include the term "color". For example, without a binding, "color NE blue" becomes a query for documents containing the words "color", "NE", and "blue".

You can also bind a tag name to a reference to a function that generates a query, giving you more control over the interpretation. For example, you can use a query generator function to scope a query to documents in a particular collection or directory.

For details, see "Binding a Tag to a Reference, Field, or Query Generator" on page 264.

6.2.2 Grammar Components and Operators

This section describes the components and operators you can use in query text passed to cts.parse. Some operators are only available in search terms that involve tags bound to query generators using the parse binding feature.

- Basic Components and Operators
- Operators Usable With Bound Tags
- Query Text Parsing Examples

6.2.2.1 Basic Components and Operators

The table below describes the basic components and operators recognized by the `cts:parse` XQuery function and the `cts.parse` JavaScript function. If you define bindings, then additional operators become available for query expressions using a bound tag; for details, see "Operators Usable With Bound Tags" on page 258.

An empty query string (`cts:parse("")`) generates an empty `cts:and-query` that matches everything.

Query	Example	Description

any adjacent terms	`dog`
	`dog tail`
	`"dog tail" cat mouse`
	`dog (cat OR mouse)`
		Match one or more terms or query expressions, as with a `cts:and-query`. Adjacent terms and query expressions are implicitly joined with `AND`. For example, `dog tail` is the same as `dog AND tail`.

"*phrase*"	`"dog tail"`
	`"dog tail" "cat whisker"`
	`dog "cat whisker"`
		Terms in double quotes are treated as a phrase. Adjacent terms and phrases are implicitly joined with `AND`. For example, `dog "cat whisker"` matches documents containing both the term `dog` and the phrase `cat whisker`. NOTE: You cannot use single quotes in place of double quotes.

( )	`(cat OR dog) zebra`	Parentheses indicate grouping. The example matches documents containing at least one of the terms `cat` or `dog` as well as the term `zebra`.

-query	`-dog`
	`-(dog OR cat)`
	`cat -dog`
		A NOT operation, as with a `cts:not-query`. For example, `cat -dog` matches documents that contain the term `cat` but that do not contain the term `dog`.

Query	Example	Description
query1 `AND` *query2*	`dog AND cat` `(cat OR dog) AND zebra`	Match two query expressions, as with a `cts:and-query`. For example, `dog AND cat` matches documents containing both the term `dog` and the term `cat`. `AND` is the default way to combine terms and phrases, so the previous example is equivalent to `dog cat`.
query1 `OR` *query2*	`dog OR cat`	Match either of two queries, as with a `cts:or-query`. The example matches documents containing at least one of either of terms `cat` or `dog`.
query1 `NOT_IN` *query2*	`dog NOT_IN "dog house"`	Match one query when the match does not overlap with another, as with `cts:not-in-query`. The example matches occurrences of `dog` when it is not in the phrase `dog house`.
query1 `NEAR` *query2*	`dog NEAR cat` `(cat food) NEAR mouse`	Find documents containing matches to the queries on either side of the `NEAR` operator when the matches occur within 10 terms of each other, as with a `cts:near-query`. For example, `dog NEAR cat` matches documents containing `dog` within 10 terms of `cat`.
query1 `NEAR/`*N* *query2*	`dog NEAR/2 cat`	Find documents containing matches to the queries on either side of the NEAR operator when the matches occur within *N* terms of each other, as with a `cts:near-query`. The example matches documents where the term `dog` occurs within 2 terms of the term `cat`.

Query	Example	Description
query1 `BOOST` *query2*	`george BOOST washington`	Find documents that match *query1*. Boost the relevance score of documents that also match *query2*. The example returns all matches for the term "george", with matches in documents that also contain "washington" having a higher relevance score. For more details, see `cts:boost-query`.
[*opt*,*opt*,...]	`cat[min-occurs=5]` `cat AND[ordered] dog`	Pass options or a weight to the cts query generated for *query*. Options after a word or phrase apply to the word query on that word or phrase. Options after the operator apply to the query associated with the operator, such as `cts:and-query` for AND. For details, see "Including Options and Weights in Query Text" on page 262.

6.2.2.2 Operators Usable With Bound Tags

When you bind a tag to an index, lexicon, field, or query generator, then you can use the tag name in the ways shown in the following table. If you use these operators in a context in which the left operand is not a tag name, then the "operator" is simply interpreted as another query term. That is, "unbound LT value" is a `cts:and-query` of word queries on the words "unbound", "LT", and "value".

For more information on defining a binding, see "Binding a Tag to a Reference, Field, or Query Generator" on page 264. For tags bound to geospatial indexes, see "Operators Usable with Geospatial Queries" on page 260.

The sub-expressions enabled by these operators can be used in combination with the grammar features described in "Basic Components and Operators" on page 256. You can also associate options with sub-expressions that use tags; for details, see "Including Options and Weights in Query Text" on page 262.

If you bind a tag to a geospatial index reference, the value you compare to the tag can be geospatial point or region. Not all the operators listed below are sensible in a geospatial context. For details, see "Binding to a Geospatial Index Reference" on page 269.

Query	Example	Description

tag:*value*	`color:red`
	`decade:1980s`
	`birthday:1999-12-31`
		Matches documents where *value* satisfies a word query against the reference bound to *tag*. For example, as with a `cts:element-word-query`.

tag:(*valueList*)	`color:(red blue)` `decade:(1980s 1990s)`	Matches documents where at least one value in *valueList* satisfies a word query against the reference bound to *tag*. For example, as with a `cts:element-word-query`.

tag = *value*	`color = red`
	`decade = 1980s`
	`birthday = 1999-12-31`
		Matches documents where *value* satisfies a value query against the reference bound to *tag*. For example, as with a `cts:element-value-query`.

tag = (*valueList*)	`color = (red blue)` `decade = (1980s 1990s)`	Matches documents where at least one value in *valueList* satisfies a value query against the reference bound to *tag*. For example, as with a `cts:element-value-query`.

tag `EQ` *value*	`color EQ red`
	
	`decade EQ 1980s`
	`birthday EQ 1999-12-31`
		Matches documents where *value* satisfies a range query with the "=" operator against the reference bound to *tag*. For example, as with a `cts:element-range-query`.

tag EQ (*valueList*)	`color EQ (red blue)` `decade EQ (1980s 1990s)`	Matches documents where at least one value in *valueList* satisfies a range query with the "=" operator against the reference bound to *tag*. For example, as with a `cts:element-word-query`.
tag `NE` *value*	`color NE red` `birthday NE 1999-12-31`	Matches documents where *value* satisfies a range query with the "!=" operator against the reference bound to *tag*. For example, as with a `cts:element-range-query`.

Query	Example	Description
tag `LT` *value*	`color LT red` `birthday LT 1999-12-31`	Matches documents where *value* satisfies a range query with the "<" operator against the reference bound to *tag*. For example, as with a `cts:element-range-query`.
tag `LE` *value*	`color LE red` `birthday LE 1999-12-31`	Matches documents where *value* satisfies a range query with the "<=" operator against the reference bound to *tag*. For example, as with a `cts:element-range-query`.
tag `GT` *value*	`color GT red` `birthday GT 1999-12-31`	Matches documents where *value* satisfies a range query with the ">" operator against the reference bound to *tag*. For example, as with a `cts:element-range-query`.
tag `GE` *value*	`color GE red` `birthday GE 1999-12-31`	Matches documents where *value* satisfies a range query with the ">=" operator against the reference bound to *tag*. For example, as with a `cts:element-range-query`.
query[*opt*,*opt*,...]	`color:(red,blue)[unstemmed]` `price GT 5[min-occurs=2]`	Pass options or a weight to the cts query generated for *query*. For details, see "Including Options and Weights in Query Text" on page 262

6.2.2.3 Operators Usable with Geospatial Queries

When you bind a tag to a geospatial point or region index, then you can use the tag name with the operators listed in this section. If you use these operators in a context in which the left operand is not a tag name, then the "operator" is simply interpreted as another query term. That is, "unbound EQ value" is a `cts:and-query` of word queries on the words "unbound", "EQ", and "value".

For more information on defining a binding, see "Binding a Tag to a Reference, Field, or Query Generator" on page 264.

The sub-expressions enabled by these operators can be used in combination with the grammar features described in "Basic Components and Operators" on page 256. You can include options with geospatial sub-expressions; for details, see "Including Options and Weights in Query Text" on page 262.

The value operand must be a geospatial point or region literal. For details, see "Binding to a Geospatial Index Reference" on page 269.

You can use the following operators with tags bound to a geospatial point index, such as a geospatial element child index or geospatial path index.

Query	Example	Description
tag`:`*value*	`pt:"37.5128,-122.2581"`	Matches documents where *value* satisfies a point query against the geospatial point reference bound to *tag*. For example, as with a `cts:element-geospatial-query`.
tag `=` *value*	`pt = "37.5128,-122.2581"`	
tag `EQ` *value*	`pt EQ "37.5128,-122.2581"`	
[*opt*,*opt*,...]	`pt EQ "37,-122"[precision=float]`	Pass options or a weight to the generated point query. For details, see "Including Options and Weights in Query Text" on page 262.

Tags bound to a geospatial region index can only be used with the `DE9IM_*` operators listed below. These operators implement the DE9-IM semantics described in http://en.wikipedia.org/wiki/DE-9IM. Expressions using these operators produce a `cts:geospatial-region-query` (XQuery) or `cts.geospatialRegionQuery` (JavaScript).

Query	Description
tag `DE9IM_CONTAINS` *value*	Matches regions in the bound index that contain the region *value*. That is, regions where `geo:region-contains(`*`indexedRegion`*`,`*`value`*`)` returns true.
tag `DE9IM_COVERED_BY` *value*	If R1 is a region in the bound index and R2 is *value*, then R1 is covered by R2 if every point of R1 is a point of R2, and the interiors of R1 and R2 have at least one point in common.
tag `DE9IM_COVERS` *value*	If R1 is a region in the bound index and R2 is *value*, R1 covers R2 if R2 lies in R1. That is, no points of R2 lie in the exterior of R1, or every point of R2 is a point of the interior or boundary of R1.
tag `DE9IM_CROSSES` *value*	If R1 is a region in the bound index and R2 is *value*, R1 crosses R2 if their interiors intersect and the dimension of the intersection is less than that of at least one of the regions.
tag `DE9IM_DISJOINT` *value*	If R1 is a region in the bound index and R2 is *value*, R1 is disjoint from R2 if the intersection of the two regions is empty.

Query	Description
tag `DE9IM_EQUALS` *value*	If R1 is a region in the bound index and R2 is *value*, R1 equals R2 if every point of R1 is a point of R2, and every point of R2 is a point of R1. That is, the regions are topologically equal.
tag `DE9IM_INTERSECTS` *value*	If R1 is a region in the bound index and R2 is *value*, R1 intersect R2 if `geo:region-intersects(R1,R2)` returns true.
tag `DE9IM_OVERLAPS` *value*	If R1 is a region in the bound index and R2 is *value*, then R1 overlaps R2 if R1 intersects R2, exclusive of boundaries, and neither region contains the other.
tag `DE9IM_TOUCHES` *value*	If R1 is a region in the bound index and R2 is *value*, R1 touches R2 if they have a boundary point in common but no interior points in common.
tag `DE9IM_WITHIN` *value*	If R1 is a region in the bound index and R2 is *value*, then R1 is within R2 if R2 contains R1.

As with point queries, pass options to a region query by putting the option list after the query. For example, if the tag "region" is bound to a geospatial region index, then you can specify the units option as follows:

```
region DE9IM_CONTAINS "@1 32,-122" [units=km]
```

6.2.3 Including Options and Weights in Query Text

Your query text can include query options or a weight that is passed through to the query generated by `cts:parse`. This is an advanced feature that you would not typically expose directly to end users. To use this feature, put the options or weight in brackets after query term or operator. The position depends on the type of query.

Place the option list adjacent to a word or phrase sub-expression or a sub-expression that uses a bound tag. For example:

```
cat[min-occurs=2]
tag LT value [min-occurs=2]
tag DE9IM_OVERLAPS [1, 10, 5, 20] [units=km]
```

Place the options adjacent to the operator when the operator is one of the operators listed in "Basic Components and Operators" on page 256 (AND, OR, NEAR, etc.). For example:

```
cat AND[ordered] dog
```

To specify a weight, use "weight=*N*". For example:

```
tag LT value [weight=2.0]
```

The following table provides additional examples of passing options and weights in query text. Assume that the query terms “cat” and “dogs” are simple words, and the query terms “price”, “pt”, and “region” are tags bound to an index, field, or lexicon reference.

Query Text	Generated Query
`cat`	`cts:word-query("cat", ("lang=en"), 1)`
`cat[case-sensitive]`	`cts:word-query(` `  "cat", ("case-sensitive","lang=en"), 1)`
`chat[stemmed,lang=fr]`	`cts:word-query("chat", ("stemmed","lang=fr"), 1)`

`cat AND dog`	`cts:and-query(`
	`  (cts:word-query("cat", ("lang=en"), 1),`
	`  cts:word-query("dog", ("lang=en"), 1)`
	`  ),("unordered"))`

`cat[min-occurs=3] AND`
`  dog[weight=2]`
	`cts:and-query(`
	`  (cts:word-query("cat",`
	`  ("lang=en","min-occurs=3"), 1),`
	`  cts:word-query("dog", ("lang=en"), 2)`
	`  ),("unordered"))`

`cat[min-occurs=3]`
`  AND[ordered]`
`perro[lang=es]`
	`cts:and-query((`
	`  cts:word-query("cat",`
	`  ("min-occurs=3","lang=en"), 1),`
	`  cts:word-query("perro", ("lang=es"), 1)`
	`  ), ("ordered"))`

`price GT 5 [min-occurs=2]`	`cts:json-property-range-query(`
	`  "price", ">", xs:int("5"), (`
	`  "min-occurs=2"), 1)`

`price EQ 5 [min-occurs=2]`
`  AND[ordered]`
`perro[lang=es]`
	`cts:and-query((`
	`  cts:json-property-range-query(`
	`  "price", ">", xs:int("5"),`
	`  ("min-occurs=2"), 1),`
	`  cts:word-query("perro", ("lang=es"), 1)`
	`  ), ("ordered"))`

`pt:"@1 37,-122"[units=km]`	`cts:element-child-geospatial-query(`
	`  fn:QName(...), fn:QName(...),`
	`  cts:point("37,-122"),`
	`  ("coordinate-system=wgs84","units=km"), 1)`

`region`
`  DE9IM_CONTAINS`
`"@1 32,-122" [units=km]`
	`cts:geospatial-region-query((`
	`  cts:geospatial-region-path-reference(`
	`  "/envelope/cts-region",(`
	`  "coordinate-system=wgs84"))),`
	`  "contains", cts:circle("@1 32,-122"),`
	`  ("units=km"), 1)`

6.2.4 Binding a Tag to a Reference, Field, or Query Generator

This topic describes how to define parse bindings that enable the use of specially scoped relational and comparison operators in query text passed to the cts:parse XQuery function or cts.parse Server-Side JavaScript function. You can create bindings to XML elements, XML element attributes, JSON properties, fields, and paths, as well as to custom parsing functions.

The following topics are covered:

- Binding Overview
- Binding to a cts:reference
- Binding to a Field by Simple Name
- Binding to a Geospatial Index Reference
- Binding to an XQuery Query Generator Function
- Binding to a JavaScript Query Generator Function

6.2.4.1 Binding Overview

The `cts:parse` XQuery function and the `cts.parse` JavaScript function accept an optional 2nd parameter that is a set of bindings between a tag and a content reference, field name, or a query generator function. When you use the tag in query text, `cts:parse` (`cts.parse`) uses the binding to generate a query based on the bound reference, field, or function.

In XQuery, bindings are represented by a map with the tag names as the keys. In JavaScript, the bindings are represented by a JavaScript object with the tag names as the object property names. For example, the following code snippet binds the tag “by” to an XML element/JSON property named “author”:

Language	Example

XQuery	`let $bindings := map:map()`
	`let $_ := map:put(`
	`  $bindings,`
	`  "by", cts:element-reference(xs:QName("author"))`

JavaScript	`const bindings =` `  { by: cts.jsonPropertyReference('author') };`

Given the above binding, you can use "by" in query text to represent the value of the "author" element or property. For example, the following query text parses to a `cts:element-word-query` (or `cts.jsonPropertyWordQuery`) for the phrase "mark twain" in the "author" XML element or JSON property.

```
by:"mark twain"
```

Note: The example above uses an element reference in XQuery and a JSON property reference in JavaScript, but your choice of query language does not limit you to a particular reference type. For example, you can create a binding with `cts:json-property-reference` in XQuery and with `cts.elementReference` in JavaScript.

You can examine the serialized output produced by the parse in Query Console to observe the results of using a bound tag in query text. For example, passing the above query text and bindings to `cts:parse` yields the results shown below:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`let $bindings := map:map()`
	`let $_ := map:put(`
	`  $bindings,`
	`  "by", cts:element-reference(xs:QName("author")))`
	`return cts:parse('by:"mark twain"', $bindings)`
	
	`(: emits`
	`: cts:element-word-query(fn:QName("","author"), "mark twain")`
	`:)`

JavaScript	`const bindings =`
	`  { by: cts.jsonPropertyReference('author') };`
	`cts.parse('by:"mark twain"', bindings)`
	
	`// emits`
	`// cts.jsonPropertyWordQuery("author", "mark twain")`

You get this result because the ":" operator signifies comparison as per a word query, and the binding dictates the word query is scoped to a specific JSON property. Thus, the combination of the operator and the bound reference determines the generated query. For details, see "Binding to a cts:reference" on page 266.

The ":", "=", and "EQ" operators also accept a grouping of values, which is handled like an OR. For example, the following query matches documents where the "author" JSON property contains either the word "twain" or the word "frost":

```
by:(twain frost)
```

If you define a binding with an empty string as the tag, the binding applies to unqualified terms like "cat". For details, see "Customizing Naked Term Handling With Bindings" on page 274.

Binding to a simple string is similar, but the bound entity in that case is a field. For details, see "Binding to a Field by Simple Name" on page 268.

For a complete mapping of reference type and operator to query type, refer to the reference documentation for `cts:parse` in the *MarkLogic XQuery and XSLT Function Reference* or `cts.parse` in the *MarkLogic Server-Side JavaScript Function Reference*.

If the default query mapping does not satisfy the requirements of your application, you can bind a tag to a query generator function instead. Binding a tag to a function that generates a cts query gives you more control over the interpretation of a query sub-expression and enables using the following operators in query text: ":", "=", "LT", "LE", "GT", "GE", "EQ", "NE".

The bound function is expected to generate a cts:query (or cts.query) from the operator and operands. For example, you could cause the query text 'by:"mark twain"' to match "mark twain" in the `author` property only when the phrase occurs in documents in a specific collection. For details, see "Binding to an XQuery Query Generator Function" on page 271 or "Binding to a JavaScript Query Generator Function" on page 272.

> **Note:** Function binding is designed to enable you to override the default query selection when a tag is bound to a reference or simple string. It is not a general purpose grammar extender. For example, you cannot define a new operators or change the number of operands expected by an operator.

6.2.4.2 Binding to a cts:reference

You can bind a tag to a cts:reference by using any cts:reference constructor. This enables you to bind a tag to an XML element or element attribute, JSON property, field, or path. Query expressions using the tag can parse to a word query, value query, or range query, depending on the operator context.

For example, the following code binds the tag “cost” to an XML element or JSON property named “price”, then uses the “cost” tag in the query expression “cost LT 15”. The use of the tag with the “LT” operator causes the expression to parse to a range query, so the database configuration should include a range index on “price” with type “float”.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`let $bindings := map:map()`
	`let $_ := map:put(`
	`  $bindings,`
	`  "cost", cts:element-reference(xs:QName("price")))`
	`return cts:parse('cost LT 15', $bindings)`
	
	`(: cts:element-range-query(`
	`  fn:QName("","price"), "<", xs:float("15"), (), 1)`
	`:)`

JavaScript	`const bindings =`
	`  { by: cts.jsonPropertyReference('price') };`
	`cts.parse('cost LT 15', bindings)`
	
	`// cts.jsonPropertyRangeQuery(`
	`//  "price", "<", xs.float("15"), [], 1)`

If you use the binding in a different operator context, the parser generates a different kind of query. For example, the “:” operator generates a word query in most cases, so the query text “cost:15” parses to a `cts:element-word-query` or `cts.jsonPropertyWordQuery`, similar to the following:

```
cts:element-word-query(fn:QName("","price"), "15", ("lang=en"), 1)

cts.jsonPropertyWordQuery("price", "15", ["lang=en"], 1)
```

If you bind a tag to a geospatial index reference, the “:” operator generates a geospatial query. For details, see “Binding to a Geospatial Index Reference” on page 269.

For a complete list of the types of query generated by each operator, refer to `cts:parse` in the *MarkLogic XQuery and XSLT Function Reference* or `cts.parse` in the *MarkLogic Server-Side JavaScript Function Reference*.

By default, the parser checks for the existence of a backing index or lexicon for each cts reference when it processes your bindings. Though it is usually beneficial to have a backing index for a binding, you can suppress the check if you want to defer index creation or know you will never use the binding in a search context that actually requires an index. For example, range queries always require an index, but a word query does not necessarily require one. If you use an unchecked binding to create a query that requires an index, you will still get an error when you use the query in a search.

To suppress the parse time index check, add the "unchecked" and "type" options when creating the reference. The "type" option is required because the parser can no longer derive this information from the index definition. The following example illustrates the parse time check vs. the search time check:

Language	Example

XQuery	`(: parse time XDMP-ELEMRIDXNOTFOUND if no range index exists:)`
	`xquery version "1.0-ml";`
	`let $bindings := map:map()`
	`let $_ := map:put(`
	`  $bindings,`
	`  "cost", cts:element-reference(xs:QName("price")))`
	`return cts:parse('cost LT 15', $bindings);`
	
	`(: search time XDMP-ELEMRIDXNOTFOUND :)`
	`xquery version "1.0-ml";`
	`let $bindings := map:map()`
	`let $_ := map:put(`
	`  $bindings, "cost", cts:element-reference(xs:QName("price"),`
	`  **("type=float","unchecked")**))`
	`return cts:search(cts:parse('cost LT 15', $bindings))`

JavaScript	`// parse time XDMP-ELEMRIDXNOTFOUND if no range index exists`
	`cts.parse('cost LT 15', {p: cts.jsonPropertyReference('price')})`
	
	`// Suppress the parse time index check`
	`const query = cts.parse('cost LT 15',`
	`  {cost: cts.jsonPropertyReference(`
	`  'price',**['type=float','unchecked']**)})`
	`// But will still get search time error if no range index found`
	`cts.search(query)  // XDMP-ELEMRIDXNOTFOUND`

6.2.4.3 Binding to a Field by Simple Name

You can bind to a field by name or by cts:reference. This section describes how to bind to field by name. To use a reference constructor, instead, see "Binding to a cts:reference" on page 266.

When you bind a tag to a simple string, the string is interpreted as the name of a field. The database configuration should include a corresponding field definition. You can bind to any type of field, including metadata fields.

For example, the following binds the tag “name” to a field named “person”:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`let $bindings := map:map()`
	`let $_ := map:put(`
	`  $bindings,`
	`  "name", "person")`
	`return cts:parse('name:"jane doe"', $bindings)`
	
	`(: cts:field-word-query("name", "jane doe", ("lang=en"), 1) :)`

JavaScript	`const bindings = { name: 'person' };`
	`cts.parse('name:"jane doe"', bindings)`
	
	`// cts.fieldWordQuery("name", "jane doe", ["lang=en"], 1)`

When you use the bound tag, it will parse to a `cts:field-word-query`, `cts:field-value-query`, or `cts:field-range-query`, depending on the operator context. If you use the tag name in a context that parses to a range query, you will get an error if the database configuration does not include a corresponding field range index.

To learn more about fields, see Fields Database Settings in the *Administrator's Guide*.

For a complete list of the kinds of query generated by the supported (cts:reference, operator) pairs, refer to cts:parse in the *MarkLogic XQuery and XSLT Function Reference* or cts.parse in the *MarkLogic Server-Side JavaScript Function Reference*.

6.2.4.4 Binding to a Geospatial Index Reference

If you bind a tag (or naked terms) to a `cts:reference` to a geospatial index, you can construct query terms that represent a geospatial point or region query. For example you can match documents containing a point within a region defined in the query text, or documents containing a region that intersects a region defined in the query text.

For example, if you bind the tag “loc” to a geospatial point index, then the following query text matches documents containing a point within a circle defined by a radius and a center point, using the syntax “*@radius lon,lat*”:

```
loc:"@5 37.5,-122.4"
```

The following code demonstrates how to define the binding and parse the above query text. In this example, the tag "loc" is bound to a geospatial point index on an XML element or JSON property named "incidents". The resulting query matches documents containing points in the "incidents" element or property contained within the circle with center (37.5,-122.4) and a radius of 5 miles.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`let $bindings := map:map()`
	`let $_ := map:put(`
	`  $bindings,`
	`  "loc", cts:geospatial-element-reference(xs:QName("incidents")))`
	`return cts:parse('loc:"@5 37.5,-122.4"', $bindings)`
	
	`(: cts:element-geospatial-query(`
	` :  fn:QName("","incidents"),`
	` :  cts:circle("@5 37.5,-122.4"),`
	` :  ("coordinate-system=wgs84"), 1)`
	` :)`

JavaScript	`cts.parse('loc:"@5 37.5,-122.4"',`
	`  { loc: cts.geospatialJsonPropertyReference('incidents') })`
	
	`// cts.jsonPropertyGeospatialQuery(`
	`//  "incidents",`
	`//  cts.circle("@5 37.5,-122.4"),`
	`//  ["coordinate-system=wgs84"], 1)`

You can bind a tag to any of the index types described in "Understanding Geospatial Query and Index Types" on page 493. Parsing an expression that uses such a tag creates a query of the corresponding type. For example, a tag bound to a geospatial element reference produces an element geospatial query, and a tag bound to a geospatial region path reference produces a geospatial region query.

Use the ":", "EQ", and "=" operators with tags bound to a geospatial point index. Use the `DE9IM_*` operators with tags bound to a geospatial region index. For details, see "Operators Usable with Geospatial Queries" on page 260. For example:

```
mypoint:"@1 -122.2465038,37.5073428"

myregion DE9IM_OVERLAPS "@1 -122.2465038,37.5073428"
```

The right operand of a geospatial query expression must be a geospatial literal. You can specify a point, circle, box, or polygon using the shorthand shown below, or you can specify any supported region type using WKT. The shorthand is equivalent to the serialization of `cts:point`, `cts:circle`, `cts:box`, and `cts:polygon` in XQuery; and of `cts.point`, `cts.circle`, `cts.box`, and `cts.polygon` in JavaScript. For details, see the corresponding region constructors and "Constructing Geospatial Point and Region Values" on page 567.

Geospatial Entity	Literal Syntax	Example
point	`lat,lon`	*tag*:"37.5, -122.4"
circle	`@radius lat,lon`	*tag*:"@5 37.5,-122.4"
box	`[sbound, wbound, nbound, ebound]`	*tag*:"[45, -122, 78, 30]"
polygon	`lat1,lon1 lat2,lon2 ...latN,lonN`	*tag*:"100,0 101,0 101,1 100,1 100,0"

Note: Geospatial point and region literals such as the point "37,-122" must be enclosed in double quotes. You cannot substitute single quotes for the double quotes.

For more details, see "Constructing Geospatial Point and Region Values" on page 567 and "Converting To and From Common Geospatial Representations" on page 562.

6.2.4.5 Binding to an XQuery Query Generator Function

A query generator function should implement the following interface:

```
function (
  $operator as xs:string,
  $values as xs:string*,
  $options as xs:string*
) as cts:query?
```

If your function does not return a value, the query sub-expression is interpreted as text.

The following example adds a `cts:collection-query` to the search, corresponding to each term in the query text that is qualified by the tag name "cat" (as in "category"). If an unsupported category name is supplied, an error is thrown. If the operator is not ":" or "EQ", no value is returned.

```
xquery version "1.0-ml";

(: The query generator :)
declare function local:scope-to-coll(
  $operator as xs:string,
  $values as xs:string*,
```

```
  $options as xs:string*)
as cts:query?
{
  if ($operator = (":", "EQ")) then
    let $known := ("classics", "fiction", "poetry")
    return cts:collection-query(
      for $c in ($values)
      return
        if ($c = $known)
        then $c
        else fn:error(
          xs:QName("ERROR"),
          fn:concat("Unrecognized category: ", $c))
    )
  else ()       (: unsupported operator :)
};

(: how to use it :)
let $bindings := map:map()
let $_ := map:put($bindings, "cat", local:scope-to-coll#3)
return cts:parse('cat EQ classics california', $bindings)
(: matchs docs in the "classics" collection that contain califorina :)
```

This query generator function produces the following results:

Query Text	Result

`cat:classics`
 	
`cat EQ classics`
	`cts:collection-query("classics")`

`cat:unrecognized`	`None - function reports an error`

`cat LT anything`	`(: interpreted as text :)`
	`cts:and-query((`
	`  (cts:word-query("cat", ("lang=en"), 1),`
	`  cts:word-query("LT", ("lang=en"), 1),`
	`  cts:word-query("anything", ("lang=en"), 1)`
	`  ),("unordered"))`

6.2.4.6 Binding to a JavaScript Query Generator Function

A query generator function should implement the following interface:

```
function (operator, values, options)
```

Where `operator` is a string containing the operator token, and `values` and `options` are either a single value or a (possibly empty) `Sequence`.

Your function can return a cts.query, return nothing, or throw an error by calling `fn.error`. If you return nothing, the sub-expression is interpreted as text.

The following example adds a `cts.collectionQuery` to the search, corresponding to each term in the query text that is qualified by the tag name "cat" (as in "category"). If an unsupported category name is supplied, an error is thrown. If the operator is not ":" or "EQ", no value is returned.

```
function scopeToColl(operator, category, options) {
  if (operator === ':' || operator === 'EQ') {
    // normalize input, which can be one val or an iterator
    const categories =
        (category instanceof Sequence)
        ? category.toArray() : [category];
    const known = ['classics', 'fiction', 'poetry']
    const collections = [];
    categories.forEach(function (c) {
      if (known.indexOf(c) != -1) {
        collections.push(c);
      } else {
        fn.error('ERROR', 'Unrecognized category: ' + c);
      }
    });
    return cts.collectionQuery(collections);
  }
  // else, unsupported operator, so return nothing
};

const bindings = { cat: scopeToColl };
cts.parse('cat:(classics poetry) california', bindings)
```

This query generator function produces the following results:

Query Text	Result

`cat:classics`
 	
`cat EQ classics`
	`cts.collectionQuery('classics')`

`cat:unrecognized`	`None - function reports an error`

`cat LT anything`	`// Function returns nothing, phrase interpreted as text`
	`// by cts.parse`
	`cts.andQuery(`
	`  [cts.wordQuery("cat", ["lang=en"], 1),`
	`   cts.wordQuery("LT", ["lang=en"], 1),`
	`   cts.wordQuery("anything", ["lang=en"], 1)],`
	`  ["unordered"])`

The values in the second parameter may be strings or numbers. If a term in the query text can be represented as a number, then your function receives it as a number. Otherwise, the term is a string.

The following table illustrates how several variations on query text are interpreted and passed as input to your query generator:

Query Text	Function Parameter Values

`tag LT value`	`operator: 'LT'`
	`values: value`
	`options: an empty Sequence`

`tag = (val1 val2)`	`operator: '='`
	`values: Sequence over val1 and val2`
	`options: an empty Sequence`

`tag:42`	`operator: ':'`
	`values: 42 as a number`
	`options: an empty Sequence`

`tag:true`	`operator: ':'`
	`values: 'true' (string, not boolean)`
	`options: an empty Sequence`

`tag:value[opt]`	`operator: ':'`
	`values: value`
	`options: 'opt'`

`tag LT value[opt1,opt2=42]`	`operator: 'NE'`
	`values: value`
	`options: a Sequence over 'opt1' and 'opt2=42'`

6.2.5 Customizing Naked Term Handling With Bindings

You can use bindings to control the interpretation of terms in query text that are not qualified by a tag (naked terms). For example, in query text such as “cat AND dog”, “cat” and “dog” are naked terms. The default interpretation of this query text is a query that matches the terms “cat” and “dog” anywhere they appear, similar to the following

```
cts:and-query((cts:word-query('cat'), cts:word-query('dog')))
```

If you create a binding with the empty string as the tag, you can customize the handling of terms that have no tag qualifier in the same way you can customize the interpretation of a defined tag. For example, you can configure the parser to scope the terms “cat” and “dog” to a particular XML element or JSON property.

You can bind naked terms to a content reference, field name, or a query generator function, just as when using a tag.

The following examples constrain naked terms to occurrences in an XML element/JSON property named “title”.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`let $bindings := map:map()`
	`let $_ := map:put($bindings, "",`
	`  cts:element-reference(xs:QName("title")))`
	`return cts:parse('cat AND dog', $bindings)`
	
	`(:`
	`  cts:and-query((`
	`  cts:element-word-query(fn:QName("","title"),"cat",("lang=en"),1),`
	`  cts:element-word-query(fn:QName("","title"),"dog",("lang=en"), 1)),`
	`  ("unordered"))`
	`:)`

JavaScript	`cts.parse(`
	`  'cat AND dog',`
	`  {'': cts.jsonPropertyReference('title')}`
	`)`
	
	`// cts.andQuery([`
	`//  cts.jsonPropertyWordQuery("title", "cat", ["lang=en"], 1),`
	`//  cts.jsonPropertyWordQuery("title", "dog", ["lang=en"], 1)`
	`//  ],`
	`//  ["unordered"])`

For more details on using bindings, see “Binding a Tag to a Reference, Field, or Query Generator” on page 264.

6.2.6 Query Text Parsing Examples

This section illustrates the output from the `cts:parse` XQuery function or `cts.parse` JavaScript function various inputs. For examples of queries that include option values, see “Including Options and Weights in Query Text” on page 262.

You can use a query similar to the following in Query Console to explore the parser output on your own. The bindings are only needed for the examples that use the "color" or "loc" tag. To parse some of the query text that uses the bound tags, you need to define an element range index on the "body-color" XML element or "bodyColor" JSON property, and a geospatial element ranage index on an XML element or JSON property named "incidents".

Query Language	Query Template

XML	`xquery version "1.0-ml";`
	`let $bindings := map:map()`
	`let $_ :=`
	`  map:put($bindings,`
	`  "color", cts:element-reference(xs:QName("body-color")))`
	`return cts:parse(`***`queryText`***`, $bindings)`

JavaScript	`cts.parse(`***`queryText`***`,` `  { color: cts.jsonPropertyReference('bodyColor') })`

The following table contains examples of input query text and the result returned by the parser.

Query Text	cts:parse Output (XQuery)	cts.parse Output (JavaScript)
`cat`	`cts:word-query(` `  "cat", ("lang=en"), 1)`	`cts.wordQuery(` `  "cat", ["lang=en"], 1)`

`cat dog`
 	
`cat AND dog`
	`cts:and-query((`
	`  cts:word-query(`
	`  "cat", ("lang=en"), 1),`
	`  cts:word-query(`
	`  "dog", ("lang=en"), 1)`
	`  ), ("unordered"))`
		`cts.andQuery([`
		`  cts.wordQuery(`
		`  "cat", ["lang=en"], 1),`
		`  cts.wordQuery(`
		`  "dog", ["lang=en"], 1)`
		`  ], ["unordered"])`

`cat dog OR mouse`	`cts:or-query((`
	`  cts:and-query((`
	`  cts:word-query(`
	`  "cat",("lang=en"),1),`
	`  cts:word-query(`
	`  "dog",("lang=en"), 1)`
	`  ), ("unordered")),`
	`  cts:word-query(`
	`  "mouse",("lang=en"),1)`
	`  ), ())`
		`cts.orQuery([`
		`  cts.andQuery([`
		`  cts.wordQuery(`
		`  "cat",["lang=en"],1),`
		`  cts.wordQuery(`
		`  "dog", ["lang=en"],1)`
		`  ], ["unordered"]),`
		`  cts.wordQuery(`
		`  "mouse", ["lang=en"],1)`
		`  ], [])`

Query Text	cts:parse Output (XQuery)	cts.parse Output (JavaScript)

`cat (dog OR mouse)`	`cts:and-query((`
	`  cts:word-query(`
	`  "cat", ("lang=en"), 1),`
	`  cts:or-query((`
	`  cts:word-query(`
	`  "dog",("lang=en"),1),`
	`  cts:word-query(`
	`  "mouse",("lang=en"),`
	`  1)`
	`  ), ())`
	`  ), ("unordered"))`
		`cts.andQuery([`
		`  cts.wordQuery(`
		`  "cat", ["lang=en"], 1),`
		`  cts.orQuery([`
		`  cts.wordQuery(`
		`  "dog",["lang=en"],1),`
		`  cts.wordQuery(`
		`  "mouse",["lang=en"],1)`
		`  ], [])`
		`  ], ["unordered"])`

`cat -dog`	`cts:and-query((`
	`  cts:word-query(`
	`  "cat", ("lang=en"), 1),`
	`  cts:not-query(`
	`  cts:word-query(`
	`  "dog", ("lang=en"),`
	`  1),`
	`  1)`
	`  ), ("unordered"))`
		`cts.andQuery([`
		`  cts.wordQuery(`
		`  "cat", ["lang=en"], 1),`
		`  cts.notQuery(`
		`  cts.wordQuery(`
		`  "dog", ["lang=en"],1),`
		`  1)`
		`  ], ["unordered"])`

`color:red`	`cts:element-word-query(`
	`  fn:QName("","body-color"),`
	`  "red", ("lang=en"), 1)`
		`cts.jsonPropertyWordQuery(`
		`  "bodyColor", "red",`
		`  ["lang=en"], 1)`

`color = red`	`cts:element-value-query(`
	`  fn:QName("","body-color"),`
	`  "red", ("lang=en"), 1)`
		`cts.jsonPropertyValueQuery(`
		`  "bodyColor", "red",`
		`  ["lang=en"], 1)`

`color EQ red`	`cts:element-range-query(`
	`  fn:QName("","body-color"),`
	`  "=", "red",`
	`  ("collation=..."), 1)`
		`cts.jsonPropertyRangeQuery(`
		`  "bodyColor", "=", "red",`
		`  ["collation=..."], 1)`

`color:(red blue)`	`cts:element-word-query(`
	`  fn:QName("","body-color"),`
	`  ("red", "blue"),`
	`  ("lang=en"), 1)`
	
	Matches if body-color contains either red or blue.
		`cts.jsonPropertyWordQuery(`
		`  "color", ["red", "blue"],`
		`  ["lang=en"], 1)`
		
		Matches if bodyColor contains either red or blue.

Query Text	cts:parse Output (XQuery)	cts.parse Output (JavaScript)
`loc:"100.0,1.0"`	`cts:element-geospatial-query( fn:QName("","incidents"), cts:point("100,1"), ("coordinate-system=wgs84"), 1)`	`cts.jsonPropertyGeospatialQuery( "incidents", cts.point("100,1"), ["coordinate-system=wgs84"], 1)`
`loc:"[10,20,30,40]"`	`cts:element-geospatial-query( fn:QName("","incidents"), cts:box("[10, 20, 30, 40]"), ("coordinate-system=wgs84", 1)`	`cts.jsonPropertyGeospatialQuery( "incidents", cts.box("[10,20,30,40]"), ["coordinate-system=wgs84", 1)`

6.3 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

6.3.1 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.

6.3.2 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. You can specify both a maximum and a minimum distance.

For more details, see the *MarkLogic XQuery and XSLT Function Reference* for `cts:near-query`.

6.3.3 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")`.

6.3.4 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
```

You can also use cts:true-query and cts:false-query to match everything or nothing. For example:

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

cts:search(fn:doc(), cts:true-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` or a `cts-true-query` that matches everything is sometimes useful when you need a `cts:query` to match everything.

6.4 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.

6.5 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

6.5.1 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*.

6.5.2 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*.

6.5.3 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 layer.

6.5.4 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.

6.5.5 Registered Queries and Relevance Calculations

Searches that use registered queries will generate results having different scores from the equivalent searches using 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 422.

6.5.6 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") )
```

6.6 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 422.

6.7 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 635) 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 as XML
- Serializing a cts.query as JSON
- Add Arbitrary Annotations With cts:annotation
- Constructing a cts:query From XML
- Constructing a cts.query From a JavaScript Object or JSON String

6.7.1 Serializing a cts:query as 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 635.

6.7.2 Serializing a cts.query as JSON

You can construct the JSON representation of a cts query manually, or by applying `xdmp.toJsonString` to the result of any `cts.query` constructor call. Consider the following example:

```
xdmp.toJsonString(cts.wordQuery("hello"))
```

If you evaluate the above expression in Query Console, you get the following output:

```
{"wordQuery":{"text":["hello"], "options":["lang=en"]}}
```

You can also turn a cts query into a JavaScript object in Server-Side JavaScript using the toObject method on the object turned by one of the cts.query constructors. For example, the following expression returns a JavaScript object equivalent to the above JSON.

```
cts.wordQuery('hello').toObject()
```

6.7.3 Add Arbitrary Annotations With cts:annotation

You can annotate your `cts:query` XML with `cts:annotation` elements. A `cts:annotation` 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:annotation` 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 "Serializations of cts:query Constructors" on page 284.

6.7.4 Constructing 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) :)
```

6.7.5 Constructing a cts.query From a JavaScript Object or JSON String

Before you can use a serialized cts.query in a context such as `cts.search`, you must "de-serialize" it and turn it back into an in-memory cts.query. When working with a serialized `cts.query` in Server-Side JavaScript, you will likely have the serialized query in memory as either a JavaScript object or as a JSON string.

To convert a JavaScript object into a cts.query node, pass the object to the cts.query constructor function. The following example artificially constructs a JavaScript object equivalent to the JSON serialization of a cts.query, for purposes of illustration.

```
const aQueryObject =
  {wordQuery: {text : ['hello'], options: ['lang=en']}}
cts.query(aQueryObject)
```

To convert a JSON string `cts.query` serialization back into a `cts.query` node, first pass the JSON string through `xdmp.fromJsonString`, and then to the `cts.query` constructor function. Note that `xdmp.fromJsonString` returns a `Sequence`, so you must use the `fn.head` function to access the underlying node value. For example:

```
cts.query(fn.head(
  xdmp.fromJsonString(
    '{"wordQuery":{"text":["hello"], "options":["lang=en"]}}')
))
```

6.8 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>
```

7.0 Creating JavaScript Search Applications

This chapter describes how to add search operations and lexicon analysis to your Server-Side JavaScript modules and extensions using the JSearch library module. This chapter includes the following sections:

- JSearch Introduction
- Searching Documents
- Scoping Operations by Collection
- Creating a cts.query
- Including Facets in Search Results
- Controlling the Ordering of Results
- Returning a Result Subset
- Including Snippets of Matching Content in Search Results
- Extracting Portions of Each Matched Document
- Using Options to Control a Query
- Transforming Results with Map and Reduce
- Querying Lexicons and Range Indexes
- Grouping Values and Facets Into Buckets
- Preparing to Run the Examples

This chapter provides background, design patterns, and examples of the JSearch library module. For the function signatures and descriptions, see the JSearch documentation under JavaScript Library Modules in the *MarkLogic Server-Side JavaScript Function Reference*.

You can also use the Node.js Client API to integrate search operations and lexicon analysis into your client-side code. For details, see the *Node.js Application Developer's Guide*.

7.1 JSearch Introduction

This section provides a high level overview of the features and design patterns of the JSearch library. This section covers the following topics:

- JSearch Feature Summary
- Top Level Function Summary
- Query Design Pattern
- How JSearch Relates to Other MarkLogic Search APIs

- Running the Examples in This Chapter

7.1.1 JSearch Feature Summary

You can use the JSearch library to perform most of the query operations available through the cts built-in functions and the Search API, including the following:

- Search document contents and document properties using Query By Example (QBE), query text parsable by `cts:parse`, and cts queries.
    - Include documents, snippets, and/or facets in your search results.
    - Apply content transformations to search results.
    - Return results in configurable slices.
- Generate facets for an arbitrary set of documents in the database.
- Query lexicons and range indexes.
    - Find lexicon and range index values and tuples (value co-occurrences).
    - Compute aggregates over lexicon and range index values and tuples.

7.1.2 Top Level Function Summary

Libraries can be imported as JavaScript MJS modules. This is the preferred import method.

The following table provides an overview of the key top level JSearch methods. All these methods are effectively query builders. You can chain additional methods to them to refine and produce results. For details, see “Query Design Pattern” on page 291.

The API also includes helper functions, not listed here, for constructing complex inputs such as lexicon references, facet definitions, and heatmap definitions.

For a complete list of functions, see the *MarkLogic Server-Side JavaScript Function Reference*.

JSearch Method	Description
`collections`	Creates a `jsearch` object that implicitly scopes all operations to one or more collections. For details, see “Scoping Operations by Collection” on page 295.
`documents`	Search documents and document properties. You can tailor the results to include data such as matching documents, document projections, and snippets, as well as search metadata such as relevance score. For details, see “Document Search Basics” on page 296.

JSearch Method	Description
`values`	Query the values in a lexicon or range index, optionally computing one or more aggregates over the values. For details, see "Querying the Values in a Lexicon or Index" on page 354.
`tuples`	Find n-way value co-occurrences in lexicons and range indexes, optionally computing one or more aggregates over the tuples. For details, see "Finding Value Co-Occurrences in Lexicons and Indexes" on page 357.
`words`	Query the values in a word lexicon. For details, see "Querying Values in a Word Lexicon" on page 359.
`facets`	Generate facets from a value lexicon. The results can optionally include documents as well as facets. For details, see "Including Facets in Search Results" on page 308.
`documentSelect`	Generate snippets, sparse document projections, and/or a set of similar documents from an arbitrary set of documents, such as the result of calling `cts.search` or `fn.doc`.

7.1.3 Query Design Pattern

The top level JSearch operations, such as document search, lexicon value queries, and lexicon tuple queries use a pipeline pattern for defining the query and customizing results. The pipeline mirrors steps MarkLogic performs when evaluating a query. The pipeline stages vary by operation, but can include steps such as query criteria definition, result ordering, and result transformations.

Building and evaluating a query consists of the following steps:

1. Select the resource you want to work with, such as documents, lexicon values, or tuples.

2. Add the pipeline stages that define your query and desired result set, such as query criteria, sort order, and transformations. All pipeline stages are optional.

3. Optionally, specify advanced options, such as a quality weight. The available options depend on the resource selected in Step 1.

4. Perform the operation and get results.

If you omit all the pipeline stages in Step 2, then you retrieve the default slice from all selected resources. For example, all the documents in the database or all values or tuples in the selected lexicon(s).

Consider the case of a document search. The following example (1) selects documents as the resource; (2) defines the query and customizes the result using the `where`, `orderBy`, `slice`, and `map` pipeline stages; (3) specifies the `returnQueryPlan` option using the `withOptions` method; and then (4) evaluates the assembled query and gets results.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()                       // 1. resource selection
  .where(cts.parse('title:california',    // 2. query defn pipeline
          {title: cts.jsonPropertyReference('title')}))
  .orderBy('price')                       //     .
  .slice(0,5)                             //     .
  .map({snippet: true})                   //     .
  .withOptions({returnQueryPlan: true})   // 3. additional options
  .result()                               // 4. query evaluation
```

The query definition pipeline in this example uses the following stages:

Stage	Description
`where(...)`	Define the query criteria: Match documents with “california” in the `title` JSON property (or XML element).
`orderBy('price')`	Define the ordering of results: Order the results by the values in the `price` property.
`slice(0,5)`	Define a result subset: Limit the results to the first 5 matches.
`map({snippet: true})`	Define a mapping operation to apply to each result: Use the built-in mapper to generate snippets.

For comparsion, below is a JSearch values query. Observe that it follows the same pattern. In this case, the selected resource is the values in a range index on the `price` JSON property or XML element.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.values('price')                   // 1. resource selection
  .where(cts.parse('by:"mark twain"',     // 2. query defn pipeline
           {by: cts.jsonPropertyReference('author')}))
  .orderBy('item','descending')           //     .
  .slice(0,20)                            //     .
  .withOptions({qualityWeight: 2})        // 3. additional options
  .result()                               // 4. query evaluation
```

The query definition pipeline in this values query example uses the following stages:

Stage	Description
`where(...)`	Define the query criteria: Limit the results to the values in documents where the `author` property or element value is “mark twain”.
`orderBy( 'item', 'descending')`	Define the ordering of results: Return the values in descending item order.
`slice(0,20)`	Define a result subset: Return the first 20 values.

The query definition pipeline is realized through a call chain, as shown in the examples. All pipeline stages are optional, but the order is fixed. The table below summarizes the pipeline stages available for querying each resource type. The stage names are also JSearch method names. Note that two pipelines are available for values and tuples queries: one for retrieving values or tuples from lexicons and another for computing aggregates over the values or tuples.

Selected Resource	Query Definition Pipeline
Documents	`where > orderBy > filter > slice > (map or reduce)`
Values	`where > (match or groupInto) > orderBy > slice > (map or reduce)` `where > aggregate`
Tuples	`where > orderBy > slice > (map or reduce)` `where > aggregate`

Results can be returned as values (typically, an array) or as an `Iterable`. The default is values. For example, the default output from a document search has the following form:

```
{ results: [resultItems], estimate: totalEstimatedMatches }
```

However, if you request an `Iterable` object by passing `'iterator'` to the result method, then you get the following:

```
{ results: iterableOverResultItems, estimate: totalEstimatedMatches }
```

When you request iterable results by calling `results('iterator')` on the various JSearch APIs, you receive a `Sequence` in some contexts and a `Generator` in others. For more information on these constructs, see `Sequence` in the *JavaScript Reference Guide* and the definition of Generator in the JavaScript standard:

http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator

7.1.4 How JSearch Relates to Other MarkLogic Search APIs

The JSearch library module is primarily designed for JavaScript developers writing MarkLogic applications that initiate document searches and lexicon queries on the server. The same capabilities are available through other server-side interfaces, such as the `cts` built-in functions and the Search API, but JSearch offers the following advantages for a JavaScript developer:

- All input and output is in the form of JavaScript objects.
- A fluent call chain pattern that is natural for JavaScript.
- Powerful convenience methods for operations such as snippet generation and faceting.

In addition, the design patterns, query styles, and configuration options are similar to those used by the Node.js Client API. Thus, developers creating multi-tier JavaScript applications will find it easy to move between client (or middle) and server tiers when using JSearch. To learn more about the Node.js Client API, see the *Node.js Application Developer's Guide*.

You can use the JSearch API in conjunction with the `cts` built-in functions, in many contexts. For example:

- You can use the `cts` query constructors to create input queries usable with a JSearch-based document search. For details, see "Using cts.query Constructors" on page 308.
- You can construct index references for a JSearch values query using the `cts.reference` constructors.
- You can use the `jsearch.documentSelect` method to generate snippets or sparse document projections from the results returned by `cts.search`.
- Many JSearch operations enable you to pass advanced options to the underlying cts layer through the `withOptions` method. For details, see "Using Options to Control a Query" on page 341.

7.1.5 Running the Examples in This Chapter

All the examples in this chapter can be run using Query Console. To configure the sample database and load the sample documents, see the instructions in "Preparing to Run the Examples" on page 375.

For more information about Query Console, see the *Query Console User Guide* or the Query Console help.

7.2 Scoping Operations by Collection

If your application primarily works with documents in one or more collections, you can use the `collections` method to create a top level `jsearch` object that implicitly limits operations by collection.

For example, suppose your application is operating on documents in a collection with the URI "classics". Including a `cts.collectionQuery('classics')` in all your query operations can be inconvenient. Instead, use the `collections` method to create a scoped search object through which you can perform all JSearch operations, as shown below:

```
import jsearch from '/MarkLogic/jsearch.mjs';
const classics = jsearch.collections('classics');

// implicitly limit results to matches in the 'classics' collection
classics.documents()
  .where(cts.parse('california'))
  .result()
```

You can use the resulting object everywhere you can use the object returned by the `require` that brings the JSearch library into scope.

You can scope to one or many collections. When you specify multiple collections, the implicit collection query matches documents in any of the collections. For example:

```
import jsearch from '/MarkLogic/jsearch.mjs';

// Work with documents in either the "novels" or "poems" collection
const books = jsearch.collections(['novels','poems']);
```

The collection scope is ignored on operations for which it makes no sense, such as when constructing a lexicon reference using a helper function like `jsearch.elementLexicon`. On operations where scope matters, such as `documents`, `values`, and `words`, the implicit `cts.collectionQuery` is added to a top-level `cts.andQuery` on every `where` clause.

For more details, see `jsearch.collections` .

7.3 Searching Documents

To perform a document search, use the `jsearch.documents` method and the design pattern described in "Query Design Pattern" on page 291.

- Document Search Basics
- Example: Basic Document Search

7.3.1 Document Search Basics

This section outlines how to perform a document search. The search features touched on here are discussed in more detail in the remainder of this chapter.

Bring the JSearch library module functions into scope by including a `import` statement similar to the following in your code.

```
import jsearch from '/MarkLogic/jsearch.mjs';
```

A document search begins by selecting documents as the resource you want to work with by calling the top level `documents` method. You can invoke this method either on the object created by the require statement, or on a collection-scoped instantiation.

```
// Work with all documents
jsearch.documents().where(cts.parse('cat')).result() ...

// Work with documents in collections 'coll1' and 'coll2'
const myColls = jsearch.collections([coll1,coll2]);
myColls.documents().where(cts.parse('cat')).result() ...
```

To learn more about working with collections, see “Scoping Operations by Collection” on page 295

Build and execute your search following the pattern described in “Query Design Pattern” on page 291. The following table maps the applicable JSearch methods to the steps in the design pattern. Note that all the pipeline stages in Step 2 are optional, but you must use them in the order shown. For an example, see “Example: Basic Document Search” on page 298.

Pattern Step		Method(s)	Notes
1	Select resource	`documents`	Required. Select documents as the resource to work with. For details, see `jsearch.documents` in the *MarkLogic Server-Side JavaScript Function Reference*.

Pattern Step		Method(s)	Notes
2	Add a query definition and result set pipeline	`where`	Optional. Define your query. Accepts one or more cts.query objects as input. If you pass in more than one `cts.query` object, the queries are implicitly AND'd together. You can create a cts.query from a QBE, query text, cts.query constructors, or any other technique that creates a `cts.query`. For details, see "Creating a cts.query" on page 300 and `DocumentsSearch.where` in the *MarkLogic Server-Side JavaScript Function Reference*.
		`orderBy`	Optional. Specify sort keys and/or sorting direction. For details, see "Controlling the Ordering of Results" on page 328 and `DocumentsSearch.orderBy` in the *MarkLogic Server-Side JavaScript Function Reference*.
		`filter`	Optional. Specify whether or not to filter the search. By default, the search is unfiltered. Filtered search is always accurate, but can take longer. For details, see `DocumentsSearch.filter` and Fast Pagination and Unfiltered Searches in the *Query Performance and Tuning Guide*.
		`slice`	Optional. Select a subset of documents from the result set. The default slice is the first 10 documents. Retrieving results incrementally is best practice for most applications.For details, see "Returning a Result Subset" on page 331 and `DocumentsSearch.slice` in the *MarkLogic Server-Side JavaScript Function Reference*.
		`map	reduce`	Optional. Configure snippeting, extraction of specific pieces of matched documents, or custom transformations. You cannot use `map` and `reduce` together. For details, see "Transforming Results with Map and Reduce" on page 343, `DocumentsSearch.map`, and `DocumentsSearch.reduce`.

Pattern Step		Method(s)	Notes
3	Add advanced options	`withOptions`	Optional. Specify additional, advanced search options that customize the search behavior. For details, see "Using Options to Control a Query" on page 341 and `DocumentsSearch.withOptions` in the *MarkLogic Server-Side JavaScript Function Reference*.
4	Evaluate the query and get results	`result`	Required. Execute the search and receive your results, optionally specifying whether to receive the results as a value or an Iterable. The default is a value (typically an array).

7.3.2 Example: Basic Document Search

The following is the most minimal JSearch document search, but it has the broadest scope in that it returns the default slice of all documents in the database.

```
jsearch.documents().result()
```

More typically, your search will include at least a `where` "clause" that defines the desired set of results. The `where` method accepts one or more cts.query objects as input and defines your search criteria. For example, the following query matches documents where the `author` property has the value "Mark Twain":

```
jsearch.documents()
  .where(jsearch.byExample({author: 'Mark Twain'}))
  .result()
```

You can customize the results by adding `orderBy`, `slice`, `map`, and `reduce` stages to the operation. For example, you can suppress the search metadata, include snippets instead of (or in addition to) the full documents, extract just a portion of each matching document, or apply a custom content transformation. These and other features are covered elsewhere in this chapter.

The following example matches documents that contain an `author` JSON property with the value "Mark Twain", `price` property with a value less than10, and that are in the `/books/` directory. Notice that the search criteria are expressed in several ways; for details, see "Creating a cts.query" on page 300. The search results contain at most the first 3 matching documents (`slice`), ordered by the value of the `title` property (`orderBy`).

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where([
    jsearch.byExample({author: 'Mark Twain'}),
    cts.parse('price LT 10',
              {price: cts.jsonPropertyReference('price')}),
```

```
    cts.directoryQuery('/books/')])
  .orderBy('title')
  .slice(0,3)
  .result()
```

This query produces output similar to the following when run against the documents and database configuration described in “Preparing to Run the Examples” on page 375.

```
{ "results": [
    { "index": 0,
      "uri": "/books/twain3.json",
      "score": 16384,
      "confidence": 0.43934014439583,
      "fitness": 0.69645345211029,
      "document": {
        "title": "Adventures of Huckleberry Finn",
        "author": "Mark Twain",
        "edition": {
          "format": "paperback",
          "price": 8
        },
        "synopsis": "The adventures of Huck, a boy ..."
      }
    },
    { "index": 1,
      "uri": "/books/twain1.json",
      "score": 16384,
      "confidence": 0.43934014439583,
      "fitness": 0.69645345211029,
      "document": {
        "title": "Adventures of Tom Sawyer",
        "author": "Mark Twain",
        "edition": {
          "format": "paperback",
          "price": 9
        },
        "synopsis": "Tales of mischief and adventure ..."
      }
    }
  ],
  "estimate": 2
}
```

By default, the results include search metadata (`uri`, `score`, `confidence`, `fitness`, etc.) and the full content of each matched document.

You can also choose whether to work with the results embedded in the return value as a value or an `Iterable`. For example, by default the results are returned in an array:

```
import jsearch from '/MarkLogic/jsearch.mjs';
const response =
  jsearch.documents()
```

```
    .where(jsearch.byExample({author: 'Mark Twain'}))
    .result();                          // or .result('value')
response.results.forEach(function (result) {
  // work with the result object
});
```

By passing "iterator" as the input to the result method, you can work with the results as an `Iterable` instead:

```
import jsearch from '/MarkLogic/jsearch.mjs';
const response =
  jsearch.documents()
    .where(jsearch.byExample({author: 'Mark Twain'}))
    .result('iterator');
for (const result of response.results) {
  // work with the result object
}
```

For more details, see the following topics:

- "Creating a cts.query" on page 300
- "Controlling the Ordering of Results" on page 328
- "Returning a Result Subset" on page 331
- "Including Snippets of Matching Content in Search Results" on page 332
- "Including Facets in Search Results" on page 308
- "Transforming Results with Map and Reduce" on page 343
- `DocumentsSearch` in the *MarkLogic Server-Side JavaScript Function Reference*

7.4 Creating a cts.query

This section describes the most common ways of creating a cts.query for defining query criteria. Most JSearch operations include a `where` clause that accepts one or more cts.query objects as input. For example, the `documents`, `values`, and `tuples` methods all return an object with a `where` method for defining query criteria.

This section covers the following topics:

- Using byExample to Create a Query
- Using Query Text to Create a cts.query
- Using cts.query Constructors

7.4.1 Using byExample to Create a Query

The `jsearch.byExample` method enables you to build queries by modeling the structure of the content you want to match. It enables you to express your search in terms of "documents that look like this".

This section covers the following topics:

- Introduction to byExample
- Example: Building a Query With byExample
- Differences Between byExample and QBE

7.4.1.1 Introduction to byExample

JSearch.byExample() and search:by-example() take a query represented as an XML element for XML or as a JSON node or map for JSON and return a cts:query that can be used in any API that takes a cts:query including cts:search(), cts:uris() and the Optic where clause:

```
jsearch.byExample({author: 'Mark Twain'})
```

Search criteria like the one immediately above are implicitly value queries with exact match semantics in QBE.

The XQuery equivalent to the preceding JavaScript call is:

```
import module namespace q =
"http://marklogic.com/appservices/querybyexample"
at "/MarkLogic/appservices/search/qbe.xqy";
q:by-example(<author>Mark Twain</author>)
```

which yields:

```
cts:element-value-query(fn:QName("","author"), "Mark Twain", ...)
```

Search criteria like the `jsearch.byExample()` above are implicitly value queries with exact match semantics in QBE, so the query constructed with `byExample` above is equivalent to the following `cts.query` constructor call:

```
// equivalent cts.query constructor call:
cts.jsonPropertyValueQuery(
  'author', 'Mark Twain',
  ['case-sensitive','diacritic-sensitive',
   'punctuation-sensitive','whitespace-sensitive',
   'unstemmed','unwildcarded','lang=en'],
  1)
```

QBE provides much of the expressive power of `cts.query` constructors. For example, you can use QBE keywords in your criteria to construct value, word, and range queries, as well as compose compound queries with logical "operators. For a more complete example see "Example: Building a Query With byExample" on page 303. For details, see "Searching Using Query By Example" on page 195.

> **Note:** The JSearch `byExample` method does not use the `$response` portion of a QBE. This and other QBE features, such as result customization, are provided through other JSearch interfaces. For details, see "Differences Between byExample and QBE" on page 305.

The input to `jsearch.byExample` can be a JavaScript object, XML node, or JSON node. In all cases, the object or node can express either a complete QBE, as described in "Searching Using Query By Example" on page 195, or just the contents of the query portion of a QBE (the search criteria). For convenience, you can also pass in a document that encapsulates an XML or JSON node that meets the preceding requirements. You must use the complete QBE form of input if you need to specify the `format` or `validate` QBE flags.

For example, all the following are valid inputs to `jsearch.byExample`:

Input	Example
JavaScript Object	`// Criteria only` `{author: 'Mark Twain'}`
	`// Fully formed QBE` `{ $query: {author: 'Mark Twain'}, $validate: true}`

Input	Example

JSON node	`// Criteria only`
	`fn.head(xdmp.unquote(`
	`  '{"author": "Mark Twain"}'`
	`)).root`

	`// Fully formed QBE`
	`fn.head(xdmp.unquote(`
	`  '{"$query": {"author": "Mark Twain"}, "$validate": true}}'`
	`)).root`

XML node	`// Criteria only`
	`fn.head(xdmp.unquote(`
	`  '<my:author xmlns:my="http://marklogic.com/example">' +`
	`  'Mark Twain</my:author>'`
	`)).root`

	`// Fully formed QBE`
	`fn.head(xdmp.unquote(`
	`  '<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">' +`
	`  '<q:query>' +`
	`  '<my:author xmlns:my="http://marklogic.com/example">' +`
	`  'Mark Twain</my:author>'+`
	`  '</q:query>' +`
	`  '<q:validate>true</q:validate>' +`
	`  '</q:qbe>'`
	`)).root`

Document node	`// (xdmp.unquote returns a Sequence of document nodes)`
	`fn.head(xdmp.unquote(`
	`  '{"$query": {"author": "Mark Twain"}, "$validate": true}}'`
	`))`

By default, a query expressed as JavaScript object or JSON node will match JSON documents and a query expressed as an XML node will match XML documents. You can use the `format` QBE flag to override this behavior; for details, see “Scoping a Search by Document Type” on page 245.

You must use the XML node (or a corresponding document node wrapper) form to search XML documents that use namespaces as there is no way to define namespaces in the JavaScript/JSON QBE format.

7.4.1.2 Example: Building a Query With byExample

This example assumes the database contains documents with the following structure:

```
{ "title": "Tom Sawyer",
  "author" : "Mark Twain",
  "edition": {
    "format": "paperback",
    "price" : 9.99
```

```
    }
}
```

To add similar data to your database, see "Preparing to Run the Examples" on page 375.

The following query uses most of the expressive power of QBE and matches the above document. The top level properties in the query object passed to `byExample` are implicitly AND'd together, so all these conditions must be met by matching documents. Since the query includes range queries on a "price" property, the database configuration must include an element range index with local name "price" and type float.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(jsearch.byExample({
    "title": {
      "$value": "adventures of tom sawyer",
      "$exact": false
    },
    "$near": [
      { "author": { "$word": "mark" } },
      { "author": { "$word": "twain" } }
    ], "$distance": 2,
    "edition": {
      "$or" : [
        { "format": "paperback" },
        { "format": "hardback" }
      ]
    },
    "$and": [
      {"price": { "$lt": 10.00 }},
      {"price": { "$ge": 8.00 }}
    ]
   }))
  .result()
```

If you run this query using the documents created by "Preparing to Run the Examples" on page 375, the above query should match one document.

The following table explains the requirements expressed by each component of the query. Each of the subquery types used in this example is explored in more detail in "Understanding QBE Sub-Query Types" on page 208.

Requirement	Example Criteria

The title is "adventures of tom sawyer". Exact match is disabled, so the match is not sensitive to case, whitespace, punctuation, or diacritics.	`"title": {`
	`  "$value": "adventures of tom sawyer",`
	`  "$exact": false`
	`}`

The author contains the word "mark" and the word "twain" within 2 words of each other.	`"$near": [`
	`  { "author": { "$word": "mark" } },`
	`  { "author": { "$word": "twain" } }`
	`],`
	`"$distance": 2`

The edition format is "paperback" or "hardback". All the atomic values in this sub-query use exact value match semantics.	`"edition": {`
	`  "$or" : [`
	`  { "format": "paperback" },`
	`  { "format": "hardback" }`
	`  ]`
	`}`

The price is less than 10.00 and greater than or equal to 8.00.	`"$and": [`
	`  {"price": { "$lt": 10.00 }},`
	`  {"price": { "$ge": 8.00 }}`
	`]`

If you examine the output from `byExample`, you can see that the generated cts.query is complicated and much more difficult to express than the QBE syntax.

For more details, see "Searching Using Query By Example" on page 195.

7.4.1.3 Differences Between byExample and QBE

The `byExample` method of JSearch does not use all parts of a QBE. A full QBE encapsulates search criteria, results refinement, and other options. However, JSearch supports some QBE features through other interfaces like `filter` and `map`. If you pass a full QBE to `byExample`, only the `$query`, `$format`, and `$validate` properties are used. Similarly, if you use an XML QBE, only the `query`, `format`, and `validate` elements are used.

When reviewing the QBE documentation or converting QBE queries from client-side code, keep the following differences and restrictions in mind:

- Use the JSearch `filter` method instead of the QBE `$filtered` flag to enable filtered search.

- Your database configuration must include a range index definition for any range queries. There is no equivalent to using `$filtered` to avoid or defer index creation.
- Use the JSearch `withOptions` method instead of the QBE `$score` flag to select a scoring algorithm.
- You cannot use the QBE options `$constraint` or `$datatype` in your queries.
- Use the JSearch `map` method instead of the QBE `$response` property to customize results.

The following table contains a QBE on the left that uses several features affected by the differences listed above, including `$filtered`, `$score`, and `$response`. The JSearch example on the right illustrates how to achieve the same result by combining `byExample` with other JSearch features.

Standalone QBE	Equivalent JSearch byExample

`{"$query": {`
`  "author": "Mark Twain"`
`  "$filtered": true,`
`  "$score": "logtf"`
`},`
`"$response": {`
`  "$snippet": { "$none": {} },`
`  "$extract": { "title": {} }`
`}`
`}`
	`import jsearch from '/MarkLogic/jsearch.mjs';`
	`jsearch.documents()`
	`  .where(jsearch.byExample({author: 'Mark Twain'}))`
	`  .filter()`
	`  .map({snippet:false,`
	`  extract: {paths: ['/title']}`
	`  })`
	`  .result()`

7.4.2 Using Query Text to Create a cts.query

Use `cts.parse` to create a `cts.query` from query text such as "cat AND dog" that a user might enter in a search text box. The `cts.parse` grammar is similar to the Search API default string query grammar. For grammar details, see "Creating a Query From Search Text With cts:parse" on page 253.

For example, the following code matches documents that contain the word "steinbeck" and the word "california", anywhere.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(cts.parse('steinbeck AND california'))
  .result()
```

You can use the `cts.parse` grammar to generate complex queries. The following table illustrates some simple query text strings with their equivalent `cts.query` constructor calls.

Query Text	Equivalent cts.query	Explanation
`(tom or huck) NEAR becky`	`cts.nearQuery( [ cts.orQuery([ cts.wordQuery("tom"), cts.wordQuery("huck")]), cts.wordQuery("becky") ])`	at least one of the terms `tom` or `huck` within 10 terms (the default distance for `cts.nearQuery`) of the term `becky`
`tom NEAR/30 huck`	`cts.nearQuery([ cts.wordQuery("tom"), cts.wordQuery("huck")], 30)`	the term `tom` within 30 terms of the term `huck`
`huck -tom`	`cts.andQuery([ cts.wordQuery("huck"), cts.notQuery( cts.wordQuery("tom")) ])`	the term `huck` where there is no occurrence of `tom`

You can also bind a keyword to a query-generating function that the parser uses to generate a sub-query when the keyword appears in a query expression. This feature is similar to using pre-defined constraint names in Search API string queries. You can use a built-in function, such as `cts.jsonPropertyReference`, or supply a custom function that returns a `cts.query`.

For example, you can use a binding to cause the query text "by:twain" to generate a query that matches the word "twain" only when it appears in the value of the `author` JSON property. (In the `cts.parse` grammar, the colon (":") operator signifies a word query by default.)

```
import jsearch from '/MarkLogic/jsearch.mjs';

// bind 'by' to the JSON property 'author'
const queryBinding = {
  by: cts.jsonPropertyReference('author')
};

// Perform a search using the bound name in a word query expression
jsearch.documents()
  .where(cts.parse('by:twain', queryBinding))
  .result();
```

You can also define a custom binding function rather than using a pre-defined function such as `cts.jsonPropertyReference`. For more details and examples, see "Creating a Query From Search Text With cts:parse" on page 253.

7.4.3 Using cts.query Constructors

You can build a `cts.query` by calling one or more `cts.query` constructor built-in functions such as `cts.andQuery` or `cts.jsonPropertyRangeQuery`. The constructors enable you to compose complex and powerful queries.

For example, the following code uses a `cts.query` constructor built-in function to create a word query that matches documents containing the phrase “mark twain” in the value of the “author” JSON property.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(
    cts.jsonPropertyWordQuery('author', 'mark twain'))
  .result();
```

Query constructor built-in functions can be either leaf constructors, such as the one in the above example, or composable constructors. A leaf constructor does not accept cts.query’s as input, while a composable constructor does. You can use composable constructors to build up powerful, complex queries.

For example, the following call creates a query that matches documents in the database directory `/books` that contain the phrase “huck” or the phrase “tom” in the “title” property and either have a “format” property with the value “paperback” or a “price” property with a value that is less than 10.

```
cts.andQuery([
  cts.directoryQuery('/books/', 'infinity'),
  cts.jsonPropertyWordQuery('title', ['huck','tom']),
  cts.orQuery([
    cts.jsonPropertyValueQuery('format', 'paperback'),
    cts.jsonPropertyRangeQuery('price', '<', 10)])
])
```

You can pass options to most cts.query constructor built-ins for fine-grained control of each portion of your search. For example, you can specify whether or not a particular word query should be case and diacritic insensitive. For details on available options, see the API reference documentation for each constructor.

For more details on constructing cts.query objects, see “Composing cts:query Expressions” on page 248.

7.5 Including Facets in Search Results

Search facets provide a summary of the values of a given characteristic across a set of search results. For example, you could query an inventory of appliances and facet on the manufacturer names. Facets can also include counts. The `jsearch.facets` method enables you to generate search result facets quickly and easily.

This section includes the following topics:

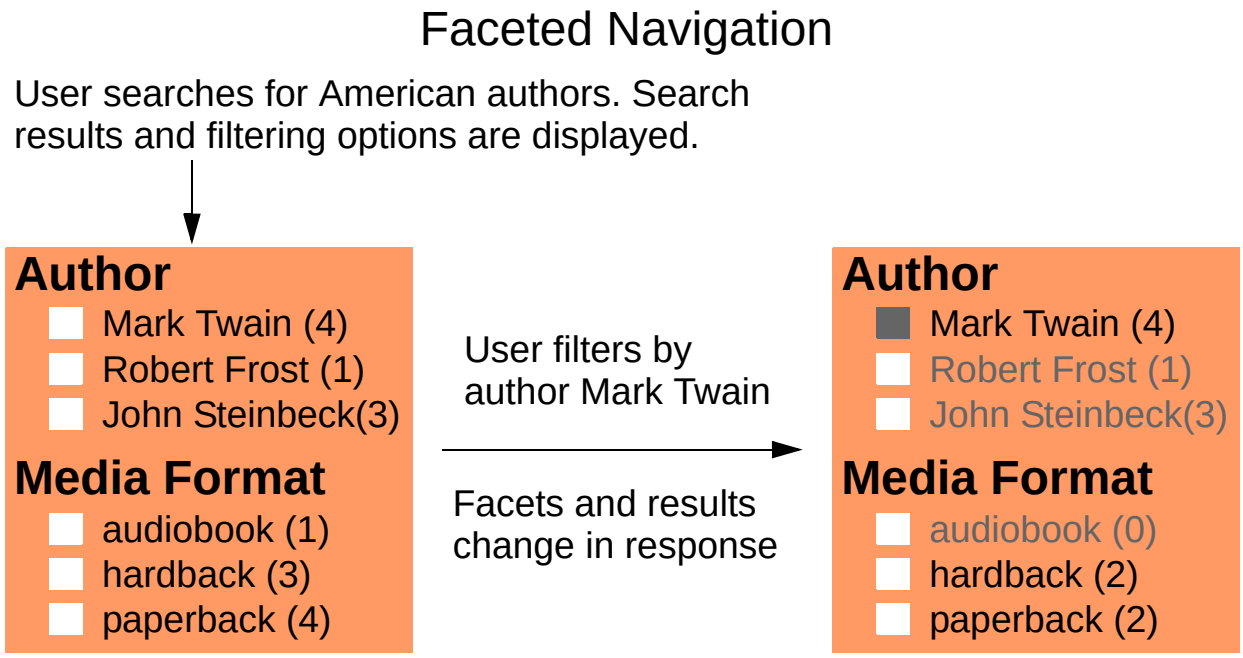
- Introduction to Facets
- Basic Steps for Generating Facets
- Example: Generating Facets From JSON Properties
- Creating a Facet Definition
- Understanding the Output of Facets
- Sorting Facet Values with OrderBy
- Retrieving Facets and Content in a Single Operation
- Multi-Facet Interactions Using othersWhere
- Example: Multi-Facet Interactions Using othersWhere

7.5.1 Introduction to Facets

Search facets can enable your application users to narrow a search by "drilling down" with search criteria presented by the application.

For example, suppose you have an application that enables users to search bibliographic data on books. If the user searches for American authors, the application displays the search results, plus filtering controls that enable the user to narrow the results by author and/or media format. The filtering controls may include both a list of values, such as author names, and the number of items matching each selection.

The following diagram depicts such an interaction. Search results are not shown; only the filtering controls are included due to space constraints. The greyed out items are just representative of how an application might choose to display unselected facet values.



The filtering categories “Author” and “Media Format” represent facets. The author names and formats are values from the author and format facets, respectively. The numbers after each value represent the number of items containing that value.

MarkLogic generates facet values and counts from range indexes and lexicons. Therefore, your database configuration must include a lexicon or index for any content feature you want to use as a facet source, such as a JSON property or XML element.

Use the JSearch `facet` method to identify an index from which to source facet data; for details, see “Creating a Facet Definition” on page 313. Use the Jsearch `facets` method to generate facets from such definitions. Only facet data is returned by default, but you can optionally request matching documents as well; for details, see “Retrieving Facets and Content in a Single Operation” on page 319.

The remainder of this section describes how to generate and customize facets in more detail.

7.5.2 Basic Steps for Generating Facets

The primary interfaces for generating facets are the `jsearch.facets` and `jsearch.facet` methods. Use the `facet` method to create a `FacetDefinition`, then pass your facet definitions to the `facets` method to create a facet generation operation. As with other JSearch operations, facets are not generated until you call the `result` method.

The following procedure outlines the steps for building a faceting operation. For a complete example, see “Example: Generating Facets From JSON Properties” on page 312.

1. Define one or more facets using the `jsearch.facet` method. For each, provide a label and an index, lexicon, or JSON property reference that identifies the facet source. The label becomes the property name for the facet data in the results.

   For example, the following call defines a facet labeled “Author” derived from a range index on the JSON property named “author”. The database must include a range index on “author”.

   ```
   jsearch.facet('Author', 'author')
   ```

   A facet definition can include additional configuration. For details, see “Creating a Facet Definition” on page 313.

2. Pass your facet definitions to the `jsearch.facets` method. For example:

   ```
   jsearch.facets([
       jsearch.facet('Author', 'author'),
       jsearch.facet('MediaFormat', 'format')])
   ```

3. Optionally, add a `documents` “clause” to return document search results and contents along with the facets. By default, only the facet data is returned. For example:

   ```
   jsearch.facets([
       jsearch.facet('Author', 'author'),
       jsearch.facet('MediaFormat', 'format')],
       jsearch.documents())
   ```

4. Optionally, use `FacetsSearch.where` method to select the documents over which to facet. You can pass one or more `cts.query` objects, just as for a document search. For example:

   ```
   jsearch.facets([
       jsearch.facet('Author', 'author'),
       jsearch.facet('MediaFormat', 'format')])
     .where(jsearch.byExample({price: {$lt: 15}}))
   ```

5. Optionally, use the `FacetsSearch.withOptions` method to specify advanced options. For example:

   ```
   jsearch.facets([
       jsearch.facet('Author', 'author'),
       jsearch.facet('MediaFormat', 'format')])
     .where(jsearch.byExample({price: {$lt: 15}}))
     .withOptions({maxThreads: 15})
   ```

6. Generate facets (and documents, if requested in Step 3) by calling the `result` method. For example:

```
jsearch.facets([
    jsearch.facet('Author', 'author'),
    jsearch.facet('MediaFormat', 'format')])
  .where(jsearch.byExample({price: {$lt: 15}}))
  .result()
```

For a complete example, see "Example: Generating Facets From JSON Properties" on page 312.

For more details, see the following topics in the *MarkLogic Server-Side JavaScript Function Reference*:

- `jsearch.facet` and `FacetDefinition`
- `jsearch.facets` and `FacetsSearch`

7.5.3 Example: Generating Facets From JSON Properties

This example is a simple demonstration of generating facets. The example uses the sample documents and database configuration described in "Preparing to Run the Examples" on page 375.

The example generates facets for documents that contain a "price" property with value less than 15 (`jsearch.byExample({price: {$lt: 15}})`). Since the search criteria is a range query, the database configuration must include a range index on "price".

Facets are generated for the matched documents from two content features:

- The "author" JSON property values. The database configuration must include a range index on this property.
- The "format" JSON property values. The database configuration must include a range index on this property.

If your database is configured according to the instructions in "Preparing to Run the Examples" on page 375, then it already includes the indexes needed to run this example.

The following query builds up and then evaluates a facet request. Facets are not generated until the `result` method is evaluated.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets([
    jsearch.facet('Author', 'author'),
    jsearch.facet('MediaFormat', 'format')])
  .where(jsearch.byExample({price: {$lt: 15}}))
  .result()
```

Running this query in Query Console produces the following output:

```
{"facets":{
  "Author": {
    "Mark Twain": 2,
    "John Steinbeck": 1
  },
  "MediaFormat": {
    "paperback": 3
}}}
```

Notice that the "facets" property of the results contains a child property corresponding to each facet definition created by `jsearch.facet`. In this case, the documents that met the "price < 15" criteria include two documents with an "author" value of "Mark Twain" and one document with an author value of "John Steinbeck". Similarly, based on the "format" property, a total of 3 paperbacks meet the price criteria.

If you add a documents query, you can retrieve facets and matched documents together. For details, see "Retrieving Facets and Content in a Single Operation" on page 319.

7.5.4 Creating a Facet Definition

The `facets` method accepts one or more facet definitions as input. Use the `jsearch.facet` method to create each facet definition.

The simplest form of facet definition just associates a facet name with a reference to a JSON property, XML element, field or other index or lexicon. For example, the following facet definition associates the name "Author" with a JSON property named "author".

```
jsearch.facet('Author', 'author')
```

However, you can further customize the facet using a pipeline pattern similar to the one described in “Query Design Pattern” on page 291. The table below describes the pipeline stages availble for building a facet definition.All pipeline stages are optional, can appear at most once, and must be used in the order shown. Most stages behave as they do when used with a values query; for details, see `ValuesSearch` in the *MarkLogic Server-Side JavaScript Function Reference*.

Method	Stage Description
`othersWhere`	Control how facets interact with each other and with any queries that are part of the facets call, such as a documents query. For details, see “Multi-Facet Interactions Using othersWhere” on page 322 and `FacetDefinition.othersWhere` in the *MarkLogic Server-Side JavaScript Function Reference*.
`thisWhere`	Control how facets interact with each other and with any queries that are part of the facets call, such as a documents query. For details, see “Multi-Facet Interactions Using othersWhere” on page 322 and `FacetDefinition.thisWhere` in the *MarkLogic Server-Side JavaScript Function Reference*.
`groupInto`	Group facet values into buckets based on a range of values. For example you can facet on price and group facet values into price range buckets such as “Less than $10” and “$10 or more”, rather than simply retrieving a set of individual prices and counts. For details, see “Grouping Values and Facets Into Buckets” on page 367 and `FacetDefinition.groupInto` in the *MarkLogic Server-Side JavaScript Function Reference*.
`orderBy`	Control whether the results from this facet are ordered by frequency or value and whether they’re listed in ascending or descending order. For details, see “Sorting Values or Tuples Query Results” on page 330 and `FacetDefinition.orderBy` in the *MarkLogic Server-Side JavaScript Function Reference*.
`slice`	Define a subset of the results to return. Slicing enables you to “page” through a large set of results. For details, see “Returning a Result Subset” on page 331 and `FacetDefinition.slice` in the *MarkLogic Server-Side JavaScript Function Reference*.

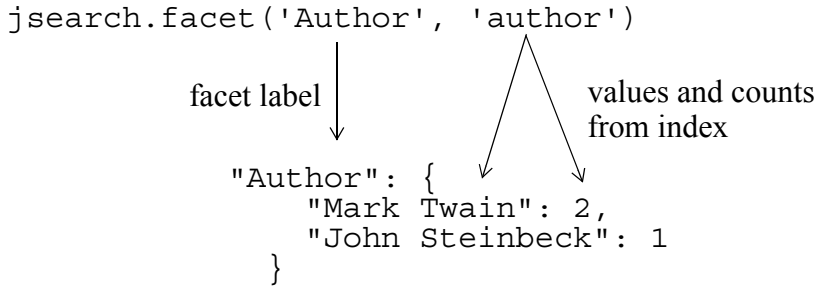
Method	Stage Description
`map | reduce`	Use `map` or `reduce` to apply transformations to the results. You can only use `map` or `reduce`, never both together. For details, see "Transforming Results with Map and Reduce" on page 343, `FacetDefinition.map`, and `FacetDefinition.reduce`.
`withOptions`	Specify advanced faceting options, such as an option accepted by `cts.values` or a quality weight. A facet definition accepts the same options configuration as a values query. For details, see `FacetDefinition.withOptions` in the *MarkLogic Server-Side JavaScript Function Reference*.

7.5.5 Understanding the Output of Facets

By default, only facet data is returned from a facets request, and the data for each facet is an object containing *`facetValue:count`* properties. That is, the default output has the following form:

```
{"facets": {
  "facetName1": {
    "facetValue1": count,
    ...
    "facetValueN": count,
  },
  "facetNameN": { ... },
}}
```

The facet names come from the facet definition. The facet values and counts come from the index or lexicon referenced in the facet definition. The following diagram shows the relationship between a facet definition and the facet data generated from it:



For example, the following output was produced by a facets request that included two facet definitions, name "Author" and "MediaFormat". For details on the input facet definitions, see "Example: Generating Facets From JSON Properties" on page 312.

```
{"facets":{
  "Author": {
```

```
    "Mark Twain": 2,
    "John Steinbeck": 1
  },
  "MediaFormat": {
    "paperback": 3
}}}
```

The built-in reducer generates the per facet objects, with counts. If you do not require counts, you can use the `map` method to bypass the reducer and configure the built-in mapper to omit the counts. For example:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets(
    jsearch.facet('Author', 'author').map({frequency: 'none'})
  .where(cts.directoryQuery('/books/'))
  .result()
```

Running this query on a database configured according to the instructions in “Preparing to Run the Examples” on page 375 produces the following output:

```
{"facets": {
  "Author": ["Mark Twain", "Robert Frost", "John Steinbeck"]
}}
```

If you include a `documents` call in your `facets` operation, then the output includes both facet data and the results of the document search. The output has the following form:

```
{ "facets": {
    property for each facet
  },
  "documents": [
    descriptor for each matched document
  ]
}
```

The documents array items are search result descriptors exactly as returned by a document search. They can include the document contents and search match snippets. For an example, see “Example: Generating Facets From JSON Properties” on page 312.

You can pass 'iterator' to your `result` call to return a `Sequence` as the value of each facet instead of an object. For example:

```
import jsearch from '/MarkLogic/jsearch.mjs';
const results =
  jsearch.facets(jsearch.facet('Author', 'author'))
    .where(cts.directoryQuery('/books/'))
    .result('iterator')
const authors = [];
for (const author of results.facets.Author) {
  authors.push(author)
}
```

```
authors

==> [{"Mark Twain":4, "Robert Frost":1, "John Steinbeck":3}]
```

In this case, the returned `Iterable` contains only a single item: The object containing the *`value:count`* properties for the facet that is produced by the built-in reducer. However, if you use a mapper or a custom reducer, you can have more items to iterate over.

For example, the following call chain configures the built-in mapper to return only the facet values, without counts, so returning an iterator results in a `Sequence` over each facet value (author name, here):

```
import jsearch from '/MarkLogic/jsearch.mjs';
const results =
jsearch.facets(
    jsearch.facet('Author', 'author').map({frequency: 'none'})
  .where(cts.directoryQuery('/books/'))
  .result('iterator')
const authors = [];
for (const author of results.facets.Author) {
  authors.push(author)
}
authors

==> ["Mark Twain", "Robert Frost", "John Steinbeck"]
```

If you use `groupInto` to group the values for a facet into "buckets" representing value ranges, then the value of the facet is either an object or an `Iterable` over the bucket descriptors. For example, suppose you generate facets on a `price` property and get the following values:

```
{"facets":{
  "Price": "8":1, "9":1, "10":1, "16":1, "18":2, "20":1, "30":1}
}}
```

You could add a `groupInto` specification to group the facet values into 3 price range buckets instead, as shown in the following query:

```
jsearch.facets(
  jsearch.facet('Price','price')
    .groupInto([
      jsearch.bucketName('under $10'), 10,
      jsearch.bucketName('$10 to $19.99'), 20,
      jsearch.bucketName('over $20')
    ]))
  .where(cts.directoryQuery('/books/'))
  .result();
```

Now, the generated facets are similar to the following:

```
{"facets": {
  "Price": {
```

```
      "under $10": {
        "value": {
          "minimum": 8,
          "maximum": 9,
          "upperBound": 10
        },
        "frequency": 2
      },
      "$10 to $19.99": {
        "value": {
          "minimum": 10,
          "maximum": 18,
          "lowerBound": 10,
          "upperBound": 20
        },
        "frequency": 4
      },
      "over $20": {
        "value": {
          "minimum": 20,
          "maximum": 30,
          "lowerBound": 20
        },
        "frequency": 2
      }
    }
} }
```

For details, see “Grouping Values and Facets Into Buckets” on page 367.

7.5.6 Sorting Facet Values with OrderBy

As mentioned in “Introduction to Facets” on page 309, facet results include a count (or frequency) by default. You can use `FacetDefinition.orderBy` to sort the results for a given facet by this frequency. Including an explicit sort order in your facet definition changes the structure of the results.

For example, the following query, which does not use `orderBy`, produces a set of facet values on “author”, in the form of a JSON object. This is the default behavior. Since the facet is an object with *`facetValue`*:*`count`* properties, the facet values are effectively unordered.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets([
    jsearch.facet('Author', 'author')])
  .where(jsearch.byExample({price: {$lt: 50}}))
  .result();

// Produces the following output:
// {"facets":{
//   "Author":{
//     "John Steinbeck":3,
//     "Mark Twain":4,
```

```
//     "Robert Frost":1}
// }}
```

If you add an `orderBy` clause to the facet definition, then the value of the facet is an array of arrays, where each inner array is of the form `[ item_value, count ]`. The array items are ordered by the frequency. For example:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets([
    jsearch.facet('Author', 'author').orderBy('frequency') ])
  .where(jsearch.byExample({price: {$lt: 50}}))
  .result();

// Produces the following output:
// {"facets":{
//   "Author":[
//     ["Mark Twain", 4],
//     ["John Steinbeck", 3],
//     ["Robert Frost", 1]
//   ]
// }}
```

You can also sort by item (the value of "author" in our example), and choose whether to list the facet values in ascending or descending order. For example, if you use the `orderBy` clause `orderBy('item', descending)`, the you get the following output:

```
{"facets":{
  "Author":[
    ["Robert Frost", 1],
    ["Mark Twain", 4],
    ["John Steinbeck", 3]
  ]
}}
```

If the default structure does not meet the needs of your application, you can modify the output using a custom mapper. For more details, see "Transforming Results with Map and Reduce" on page 343.

7.5.7 Retrieving Facets and Content in a Single Operation

By default, the result of facet generation does not include content from the documents from which the facets are derived. Add snippets, complete documents, or document projections to the results by including a `documents` query in your `facets` call.

For example, the following query returns both facets and snippets for documents that contain a "price" property with a value less than 15:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets([
    jsearch.facet('Author', 'author'),
```

```
    jsearch.facet('MediaFormat', 'format')],
    jsearch.documents())
  .where(jsearch.byExample({price: {$lt: 15}}))
  .result()
```

Running this query against the database created by "Preparing to Run the Examples" on page 375 produces the following output. Notice the output includes facets on author and format, plus the document search results containing snippets (in the "properties" property).

```
{ "facets": {
    "Author": {
      "Mark Twain": 2,
      "John Steinbeck": 1
    },
    "MediaFormat": { "paperback": 3 }
  },
  "documents": [
    { "uri": "/books/twain1.json",
      "path": "fn:doc(\"/books/twain1.json\")",
      "index": 0,
      "matches": [ {
        "path":
          "fn:doc(\"/books/twain1.json\")/edition/number-node(\"price\")",
        "matchText": [ { "highlight": "9" } ]
      } ]
    },
    ...additional documents...
  ],
  "estimate": 3
}
```

The `matches` property of each `documents` item contains the snippets. For example, if the above facets results are saved in a variable named "results", then you can access the snippets for a given document through `results.documents[n].matches`.

To include the complete documents in your facet results instead of just snippets, configure the built-in mapper on the documents query to extract "all". For example:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets([
    jsearch.facet('Author', 'author'),
    jsearch.facet('MediaFormat', 'format')],
    jsearch.documents().map({extract:{select:'all'}})
  .where(jsearch.byExample({price: {$lt: 15}}))
  .result()
```

In this case, you access the document contents through the extracted property of each document. For example, `results.documents[n].extracted`. The `extracted` property value is an array because you can potentially project multiple subsets of content out of the matched document using the map and reduce features. For details, see "Extracting Portions of Each Matched Document" on page 337.

The documents query can include `where`, `orderBy`, `filter`, `slice`, `map`/`reduce`, and `withOptions` qualifiers, just as with a standalone document search. For details, see "Document Search Basics" on page 296.

The document search combines the queries in the `where` qualifier of the `facets` query, the `where` qualifier of the documents query, and any `othersWhere` queries on facet definitions into a single AND query.

For example, the following `facets` query includes uses all three query sources.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets([
    jsearch.facet('Author', 'author'),
    jsearch.facet('MediaFormat','format')
      .othersWhere(jsearch.byExample({format: 'paperback'}))],
    jsearch.documents()
      .where(jsearch.byExample({author: 'Mark Twain'})))
  .where(jsearch.byExample({price: {$lt: 20}}))
  .result()
```

This query has the following effect on the returned results:

- Only generate facets from documents where "price < 20". From this part of the query: `jsearch.facets(...).where(jsearch.byExample({price: {$lt: 20}})).`
- For facets other than `format`, only return facet values for documents where "format is paperback". From this part of the query: `jsearch.facet('MediaFormat','format').othersWhere(jsearch.byExample({format: 'paperback'}))`
- Only return documents where "author is Mark Twain". From this part of the query: `jsearch.documents().where(jsearch.byExample({author: 'Mark Twain'}))`

Thus, the query only returns matches where all the following conditions are met: "price < 20" and "format is paperback" and "author is Mark Twain".

You can use the `returnQueryPlan` option to explore this relationship. For example, adding a `withOptions` call to the documents query as shown below returns the following information in the results:

```
...
jsearch.documents()
  .where(jsearch.byExample({author: 'Mark Twain'}))
  .withOptions({returnQueryPlan: true})
...

==> results.queryPlan includes the following information
    (reformatted for readability)

Search query contributed 3 constraints:
  cts.andQuery([
```

```
    cts.jsonPropertyRangeQuery("price", "<", xs.float("20"), [], 1),
    cts.jsonPropertyValueQuery("format", "paperback",
      ["case-sensitive","diacritic-sensitive","punctuation-sensitive",
       "whitespace-sensitive","unstemmed","unwildcarded","lang=en"], 1),
    cts.jsonPropertyValueQuery("author", "Mark Twain",
      ["case-sensitive","diacritic-sensitive","punctuation-sensitive",
       "whitespace-sensitive","unstemmed","unwildcarded","lang=en"], 1)
  ], [])
```

7.5.8 Multi-Facet Interactions Using othersWhere

Use the `FacetDefinition.othersWhere` method to efficiently vary facet values across user interactions and deliver a more intuitive faceted navigation user experience.

Imagine an application that enables users to filter a search using facet-based filtering controls. Each time a user interacts with the filtering controls, the application makes a request to MarkLogic to retrieve new search results and facet values that reflect the current search criteria.

A naive implementation might apply the selection criteria across all facets and document results. However, this causes values to "drop out" of the filtering choices, making it more difficult for users to be aware of other choice or change the filters.

The application could generate the values for each facet and for the matching documents independently, but this is inefficient because it requires multiple requests to MarkLogic. A better approach is to use the `othersWhere` method to apply criteria asymmetrically to the facets and collectively to the document search portion.

The following example uses `othersWhere` to generate facet values for two selection criteria, an author value of "Mark Twain" and a format value of "paperback":

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets(
    [jsearch.facet('Author', 'author')
       .othersWhere(jsearch.byExample({author: 'Mark Twain'})),
     jsearch.facet('MediaFormat', 'format')
       .othersWhere(jsearch.byExample({format: 'paperback'}))],
    jsearch.documents())
  .where(cts.directoryQuery('/books/'))
  .result()
```

When each facet applies `othersWhere` to selection criteria based on itself, you get multi-facet interactions. For example, the above query returns the following results. Thanks to the use of `othersWhere` on each facet definition, the author facet values are unaffected by the "Mark Twain" selection and the format facet values are unaffected by "paperback" selection. The document search is affected by both.

```
{"facets":{
   "Author":{"John Steinbeck":1, "Mark Twain":2, "Robert Frost":1},
   "MediaFormat":{"hardback":2, "paperback":2}},
```

```
 "documents":[ ...snippets for docs matching both criteria... ]
}
```

If you pass the criteria in through the `where` method instead, some facet values "drop out", making it more difficult for users to see the available selections or to change selections. For example, the following query puts the author and format criteria in the `where` call, resulting in the facet values shown:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets(
    [jsearch.facet('Author', 'author'),
     jsearch.facet('MediaFormat', 'format')],
    jsearch.documents())
  .where([cts.directoryQuery('/books/'),
          jsearch.byExample({author: 'Mark Twain'}),
          jsearch.byExample({format: 'paperback'})])
  .result()

==>
{"facets":{
    "Author":{"Mark Twain":2},
    "MediaFormat":{"paperback":2}},
 "documents":[ ...snippets for docs matching both criteria... ]
```

The differences in these two approaches are explored in more detail in "Example: Multi-Facet Interactions Using othersWhere" on page 323.

The JSearch API also includes a `FacetDefinition.thisWhere` modifier which has the opposite effect of `othersWhere`: The selection criteria is applied only to the subject facet, not to any other facets or to the document search. For details, see `FacetDefinition.thisWhere` in the *MarkLogic Server-Side JavaScript Function Reference*.

7.5.9 Example: Multi-Facet Interactions Using othersWhere

This example explores the use of `othersWhere` to enable search selection criteria to affect related facets asymmetrically, as described in "Multi-Facet Interactions Using othersWhere" on page 322.

This example assumes the database configuration and content described in "Preparing to Run the Examples" on page 375.

Suppose you have an application that enables users to search for books, and the application displays facets on author and format (hardback, paperback, etc.) that can be used to narrow a search.

The following diagram contrasts two possible approaches to implementing such a faceted navigation control. The middle column represents a faceted navigation control when the user's selection criteria are applied symmetrically to all facets through the `where` method. The rightmost column represents the same control when the user's criteria are applied asymmetrically using `othersWhere`. Notice that, in the rightmost column, the user can always see and select alternative criteria.

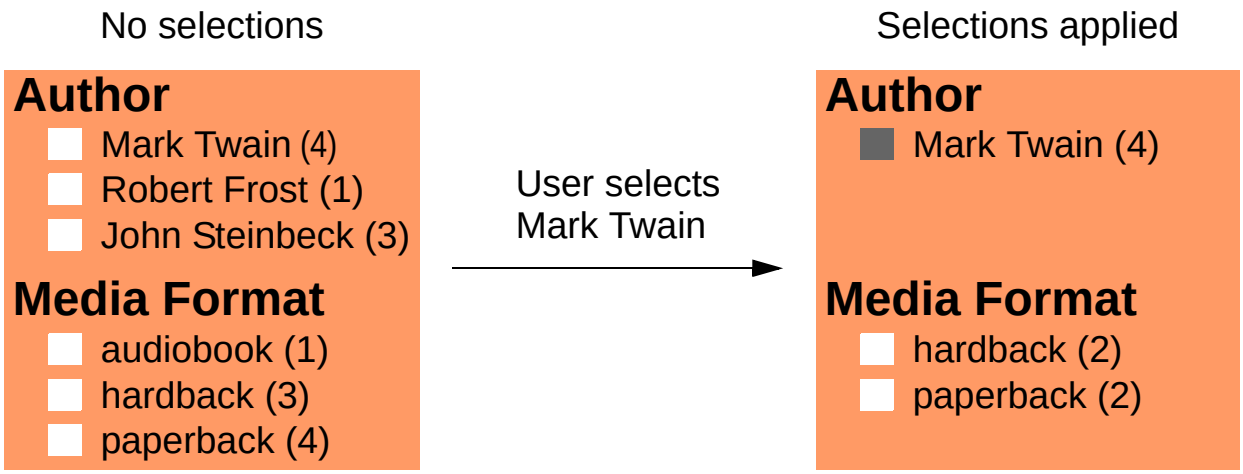
Facet criteria	Selections apply to ALL facets (where())	Selections apply to OTHER facets (othersWhere())
No selection criteria	**Author** ☐ Mark Twain (4) ☐ Robert Frost (1) ☐ John Steinbeck (3) **Media Format** ☐ audiobook (1) ☐ hardback (3) ☐ paperback (4)	**Author** ☐ Mark Twain (4) ☐ Robert Frost (1) ☐ John Steinbeck (3) **Media Format** ☐ audiobook (1) ☐ hardback (3) ☐ paperback (4)
author: Mark Twain	**Author** ■ Mark Twain (4) **Media Format** ☐ hardback (2) ☐ paperback (2)	**Author** ■ Mark Twain (4) ☐ Robert Frost (1) ☐ John Steinbeck (3) **Media Format** ☐ hardback (2) ☐ paperback (2)
format: paperback	**Author** ☐ Mark Twain (2) ☐ Robert Frost (1) ☐ John Steinbeck (1) **Media Format** ■ paperback (4)	**Author** ☐ Mark Twain (2) ☐ Robert Frost (1) ☐ John Steinbeck (1) **Media Format** ☐ audiobook (1) ☐ hardback (3) ■ paperback (4)
author: Mark Twain AND format: paperback	**Author** ■ Mark Twain (2) **Media Format** ■ paperback (2)	**Author** ■ Mark Twain (2) ☐ Robert Frost (1) ☐ John Steinbeck (1) **Media Format** ☐ hardback (2) ■ paperback (2)

The remainder of this example walks through the code that backs the results in both columns.

Before the user selects any criteria, the baseline facets are generated with the following request. Facet values are generated for the "author" and "format" JSON properties. The documents in the "/books/" directory seed the initial search results that the user can drill down on. (Matched documents are not shown.)

```
// baseline - no selection criteria
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets([
    jsearch.facet('Author', 'author'),
    jsearch.facet('MediaFormat', 'format')
  ], jsearch.documents())
  .where(cts.directoryQuery('/books/'))
  .result()
```

Consider the case where the user then selects an author, and the application applies the selection criteria unconditionally, resulting in the following filtering control changes:



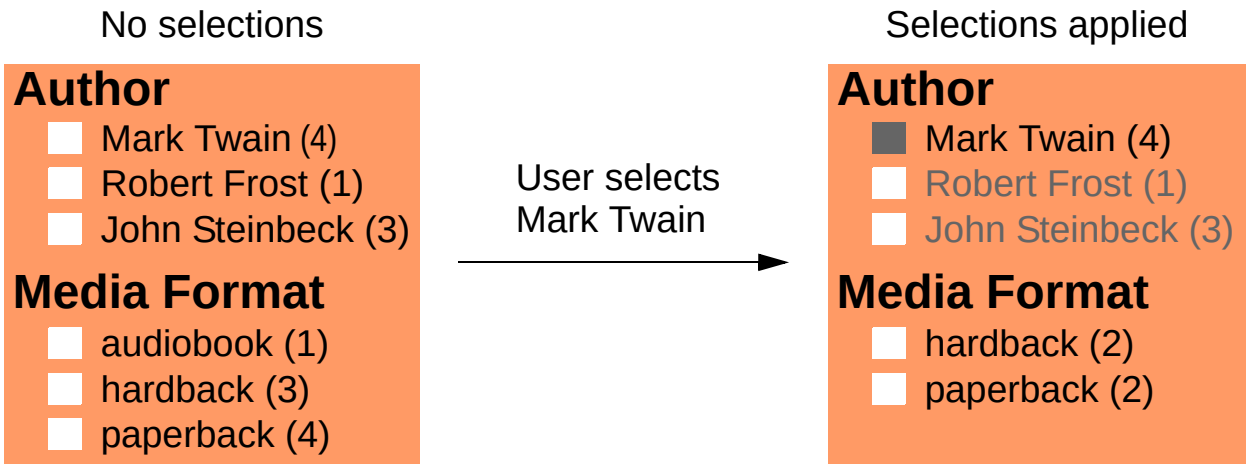
The user can no longer readily see the other available authors. These results were generated by the following query, where the `cts.directoryQuery` query represents the baseline search, and the `jsearch.byExample` query represents the user selection. Passing the author query to the `where` method applies it to all facets and the document search.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets(
    [jsearch.facet('Author', 'author'),
     jsearch.facet('MediaFormat', 'format')],
    jsearch.documents())
  .where([cts.directoryQuery('/books/'),
          jsearch.byExample({author: 'Mark Twain'})])
  .result()
```

By moving the author query to an `othersWhere` modifier on the `author` facet, you can apply the selection to other facets, such as `format`, and to the document search, but leave the author facet unaffected by the selection criteria. For example:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets(
    [jsearch.facet('Author', 'author')
       .othersWhere(jsearch.byExample({author: 'Mark Twain'})),
     jsearch.facet('MediaFormat', 'format')],
    jsearch.documents())
  .where(cts.directoryQuery('/books/'))
  .result()
```

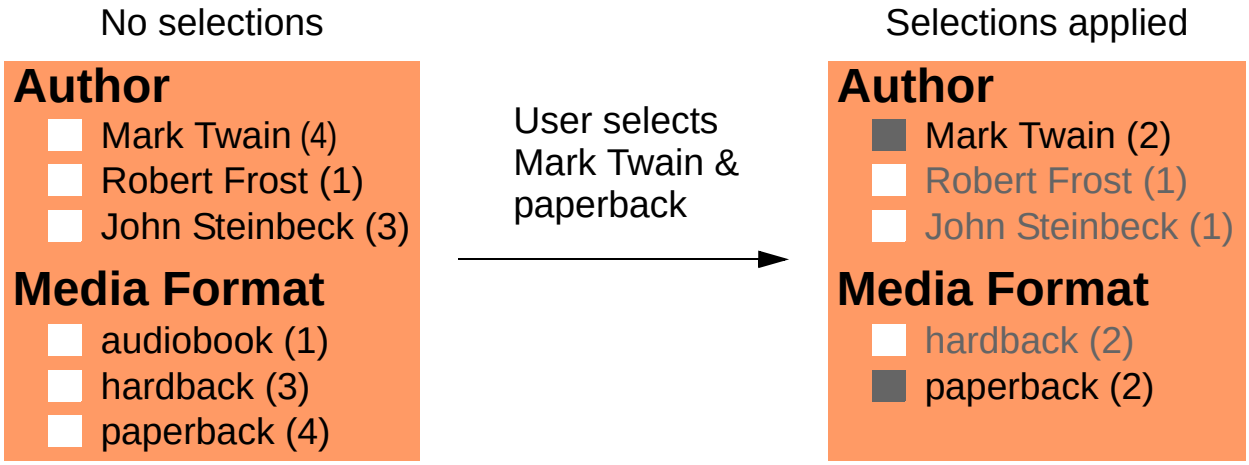
Using using `othersWhere` instead of `where` to pass the criteria results in the following display. The user can clearly see the alternative author choices and the number of items that match each other. Yet, the user can still see how his author selection affects the available media formats and the matching documents. The diagram below illustrates how the application might display the returned facet values. Snippets are returned for all documents with "Mark Twain" as the author.



If the user chooses to further filter on the "paperback" media format, you can use `othersWhere` on the `format` facet to apply this criteria to the `author` facet values and the document search, but leave all the `format` facets values available. For example:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets(
    [jsearch.facet('Author', 'author')
       .othersWhere(jsearch.byExample({author: 'Mark Twain'})),
     jsearch.facet('MediaFormat', 'format')
       .othersWhere(jsearch.byExample({format: 'paperback'}))],
    jsearch.documents())
  .where(cts.directoryQuery('/books/'))
  .result()
```

The above query results in the following display. The user can easily see and select a different author or format. The matched documents are not shown, but they consist of documents that match both the author and format selections.



7.6 Controlling the Ordering of Results

Use the `orderBy` function to control the order in which your query results are returned. You can apply an `orderBy` "clause" to a document search, word lexicon query, values query, or tuples query.

Though you can use `orderBy` with all these query types, the specifics vary. For example, you can only specify content-based sort keys in a document search, and you can only choose between item order and frequency order on a values or tuples query.

This section covers the following topics.

- Sorting Document Search Results
- Sorting Values or Tuples Query Results
- Sorting Word Lexicon Query Results
- Sorting Facet Values

7.6.1 Sorting Document Search Results

By default, search results are returned in relevance order, with most relevant results displayed first. That is, the sort key is the relevance score and the sort order is descending.

You can use the `DocumentsSearch.orderBy` method to change the sort key and ordering (ascending/descending). You can sort the results by features of your content, such as the value of a specified JSON property, and by attributes of the match, such as fitness, confidence, or document order. You must configure a range index for each JSON property, XML element, XML attribute, field, or path on which you sort.

For example, the following code sorts results by value of the JSON property named "title". A range index for the "title" property must exist.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(jsearch.byExample({'author': { '$word': 'twain' }}))
  .orderBy('title')
  .result();
```

The use of a simple name in the `orderBy` call implies a `cts.jsonPropertyReference`. You can also explicitly construct a `cts.reference` by calling an index reference constructor such as `cts.jsonPropertyReference`, `cts.elementReference`, `cts.fieldReference`, or `cts.pathReference`. For example, the following call specifies ordering on the JSON property "price":

```
orderBy(cts.jsonPropertyReference('price'))
```

To sort results based on search metadata such as confidence, fitness, and quality, use the `cts.order` constructors. For example, the following `orderBy` specifies sorting by confidence rather than relevance score:

```
orderBy(cts.confidenceOrder())
```

You can also use the cts.order constructors to control whether results are sorted in ascending or descending order with respect to a sort key. For example, the following call sorts by the JSON property "price", in ascending order:

```
orderBy(
  cts.indexOrder(cts.jsonPropertyReference('price'), 'ascending'))
```

You can specify more than one sort key. When there are multiple keys, they're applied in the order they appear in the array passed to orderBy. For example, the following call says to first order results by the "price" JSON property values, and then by the "title" values.

```
orderBy(['price', 'title'])
```

For details, see `DocumentsSearch.orderBy` in the *MarkLogic Server-Side JavaScript Function Reference* and Sorting Searches Using Range Indexes in the *Query Performance and Tuning Guide*.

7.6.2 Sorting Values or Tuples Query Results

By default, values and tuples query results are returned in ascending item order. You can use the `ValuesSearch.orderBy` and `TuplesSearch.orderBy` methods to specify whether to order the results by value (item order) or frequency, and whether to use ascending or descending order.

For example, the following query returns all the values of the `price` JSON property, in ascending order of the values:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.values('price').result()

==> [8, 9, 10, 16, 18, 20, 30]
```

The following code modifies the query to return the results in frequency order. By default, frequency order returns results in descending order (most to least frequent). In this case, the database contained multiple documents with price 18, and only a single document containing each of the other price points, so the 18 value sorted to the front of the result array, and the remaining values that share the same frequency appear in document order.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.values('price').orderBy('frequency').result()

==> [18, 8, 9, 10, 16, 20, 30]
```

To order the results by ascending frequency value, pass 'ascending' as the second parameter of `orderBy`. For example:

```
orderBy('frequency', 'ascending')
```

You can also include the frequency values in the results using the map or reduce methods. For details, see “Querying the Values in a Lexicon or Index” on page 354.

7.6.3 Sorting Word Lexicon Query Results

When you query a word lexicon using the `jsearch.words` resource selector method, results are returned in ascending order. Use the `WordsSearch.orderBy` method to control whether the results are returned in ascending or descending order.

For example, the following query returns the first 10 results in the default (ascending) order:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.words('title').result()

==>
["Adventures", "and", "Collected",
 "East", "Eden", "Finn", "Grapes",
 "Huckleberry", "Men", "Mice"]
```

You can use `orderBy` to change the order of results. For example, the following call returns the 10 results when the words in `title` are sorted in descending order:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.words('title').orderBy('descending').result()

==>
["Wrath", "Works", "Tom", "The",
 "Sawyer", "Of", "of", "Mice",
 "Men", "Huckleberry"]
```

Note that this example assumes the database configuration includes a word lexicon on the "title" JSON property. For more details on querying word lexicons, see "Querying Values in a Word Lexicon" on page 359.

7.6.4 Sorting Facet Values

When you generate facets with frequencies using `jsearch.facets`, the values of each facet are expressed as a JSON object, so they are effectively unordered. You can use `FacetDefinition.orderBy` to control the sort order and change the output to a structure that can be meaningfully ordered (an array of arrays).

For more details, see "Sorting Facet Values with OrderBy" on page 318.

7.7 Returning a Result Subset

You can use the `slice` method to return a subset of the results from a top level documents, values, tuples, or words query, or when generating facets.

A slice specification works like `Array.slice` and has the following form:

```
slice(firstPosToReturn, lastPosToReturn + 1)
```

The positions use a 0-based index. That is, the first item is position 0 in the result list. Thus, the following returns the first 3 documents in the "classics" collection:

```
import jsearch from '/MarkLogic/jsearch.mjs';
const classics = jsearch.collections('classics');

classics.documents()
  .slice(0,3)
  .result()
```

You cannot request items past the end of result set, so it is possible get fewer than the requested number of items back. When the search results are exhausted, the `results` property of the return value is `null`, just as for a search which matches nothing. For example:

```
{ results: null, estimate: 4 }
```

Applying `slice` iteratively to the same query enables you to return successive "pages" of results. For example, the following code iterates over search results in blocks of three results at a time:

```
import jsearch from '/MarkLogic/jsearch.mjs';
const sliceStep = 3;      // max results per batch
const sliceStart = 0;
const sliceEnd = sliceStep;
const response = {};
do {
  response = jsearch.documents().slice(sliceStart, sliceEnd).result();
  if (response.results != null) {
    // do something with the results
    sliceStart += response.results.length;
    sliceEnd += sliceStep;
  }
} while (response.results != null);
```

You can set the slice end position to zero to suppress returning results when you're only interested in query metadata, such as the estimate or when using `returnQueryPlan:true`. For example, the following returns the estimate without results:

```
import jsearch from '/MarkLogic/jsearch.mjs';

jsearch.documents()
  .where(cts.jsonPropertyValueQuery('author', 'Mark Twain'))
  .slice(0,0)
  .result()

==>

{ results: null, estimate: 4 }
```

For details, see the following methods:

- `DocumentsSearch.slice`
- `FacetDefinition.slice`
- `ValuesSearch.slice`
- `TuplesSearch.slice`
- `WordsSearch.slice`

7.8 Including Snippets of Matching Content in Search Results

When you perform a document search using `jsearch.documents`, the result is an array or `Iterable` over descriptors of each match. Each descriptor includes the contents of the matching document by default. You can use snippeting to a include portion of the content around the match in each result, instead of (or in addition to) the complete document.

This section covers the following topics:

- Enabling Snippet Generation
- Configuring the Built-In Snippet Generator
- Returning Snippets and Documents Together
- Generating Custom Snippets
- Standalone Snippet Generation

7.8.1 Enabling Snippet Generation

You can include snippets in a document query by adding a `map` clause to your query that sets the built-in mapper configuration property `snippet` to `true` or setting `snippet` to a configuration object, as described in "Configuring the Built-In Snippet Generator" on page 334. (Snippets are generated by default when you include any document query in a `jsearch.facets` operation.)

For example, the following query matches occurrences of the word "california" and returns the default snippets instead of the matching document:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(jsearch.byExample({synopsis: {$word: 'california'}}))
  .map({snippet: true})
  .result()

==>
{"results":[
  {"score":28672,
   "fitness":0.681636929512024,
   "uri":"/books/steinbeck1.json",
   "path":"fn:doc(\"/books/steinbeck1.json\")",
   "confidence":0.529645204544067,
   "index":0,
   "matches":[{
     "path":"fn:doc(\"/books/steinbeck1.json\")/text(\"synopsis\")",
     "matchText":[
       "...from their homestead and forced to the promised land of ",
       {"highlight":"California"}, "."
     ]
   }]
  },
  { ... }, ...
  ],
  "estimate":3
}
```

If this was a default search (no snippets), there would be a "document" property instead of the "matches" property, as shown in "Example: Basic Document Search" on page 298.

For more details, see `DocumentsSearch.map`.

7.8.2 Configuring the Built-In Snippet Generator

You can configure the built-in snippet generator by setting the built-in mapper `snippet` property to a configuration object instead of a simple boolean vaue.

You can set the following snippet configuration properties:

Property	Description
`maxMatches`	The maximum number of nodes containing a highlighted term to include in the snippet. Default: 4.
`perMatchTokens`	The maximum number of tokens (typically words) per matching node that surround the highlighted term(s) in the snippet. Default: 30.
`maxSnippetChars`	The maximum total snippet size, in characters. Default: 200.
`preferredMatches`	The snippet alogorithm looks for matches first in the specified XML element or JSON property nodes in each snippet. If no matches are found in the preferred elements or properties, the algorithm falls back to default content. XML element names can be namespace qualified; use the `namespaces` property (sibling of `snippet`) to define your prefixes.
`query`	Generate snippets based on matches to the specified query. Required when snippeting with `documentSelect`, optional when snippeting with `documents`. This is only useful for `documents().map()` when the snippet query needs to be different from the document retrieval query (e.g. the query in the `where` clause).

For example, the following configuration only returns snippets for matches occurring in the `synopsis` property and surrounds the highlighted matching text by at most 5 tokens.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(cts.wordQuery('california'))
  .map({snippet: {
    preferredMatches: ['synopsis'],
    perMatchTokens: 5
  }})
  .result()
```

Thus, if the word query for occurrences of "california" matched text in both the `title` and `synopsis` for some documents, only the matches in `synopsis` are returned. Also, the snippet match text is shorter, as shown below.

```
// match text in snippet with default perMatchTokens
"matchText":[
  "...an unlikely pair of drifters who move from
   job to job as farm laborers in ",
  {"highlight":"California"},
  ", until it all goes horribly awry."
]

// match text in snippet with perMatchTokens set to 5
"matchText":[
  "...farm laborers in ",
  {"highlight":"California"},
  ", until it..."
]
```

When snippeting over XML documents and using `preferredMatches`, use a QName rather than a simple string to specify namespace-qualified elements. For example:

```
{snippet: {
  preferredMatches: [fn.QName('/my/namespace','synopsis')]
}}
```

For more details, see `DocumentsSearch.map`.

7.8.3 Returning Snippets and Documents Together

To return snippets and complete documents or document projections together, set `snippet` to `true` and configure the `extract` property of the built-in mapper to select the desired document contents. For details about `extract`, see "Extracting Portions of Each Matched Document" on page 337.

The following example returns the entire matching document in an `extracted` property and the snippets in the `matches` property of the results:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(jsearch.byExample({synopsis: {$word: 'California'}}))
  .map({snippet: true, extract: {selected: 'all'}})
  .result()

==>
{"results":[
  {"score":28672,
   "fitness":0.681636929512024,
   "uri":"/books/steinbeck1.json",
   "path":"fn:doc(\"/books/steinbeck1.json\")",
   "extracted":[{
     "title":"The Grapes of Wrath",
```

```
      "author":"John Steinbeck",
      "edition":{"format":"paperback", "price":9.99},
      "synopsis":"Chronicles the 1930s Dust Bowl migration of one
         Oklahoma farm family, from their homestead and forced to
         the promised land of California."
    }]
    "confidence":0.529645204544067,
    "index":0,
    "matches":[{
      "path":"fn:doc(\"/books/steinbeck1.json\")/text(\"synopsis\")",
      "matchText":[
        "...from their homestead and forced to the promised land of ",
        {"highlight":"California"}, "."
      ]
    }]
   },
   { ... }, ...
   ],
   "estimate":3
}
```

For more details, see `DocumentsSearch.map`.

7.8.4 Generating Custom Snippets

If the snippets and projections generated by the built-in mapper do not meet the needs of your application, you can use a custom mapper to generate customized results. For details, see “Transforming Results with Map and Reduce” on page 343.

7.8.5 Standalone Snippet Generation

You can use the `jsearch.documentSelect` method to generate snippets from an arbitrary set of documents, such as the output from `cts.search` or `fn.doc`. The output is a `Sequence` of results.

If the input is the result of a search that matches text, then the results include search result metadata such as score, along with your snippets. Search metadata is not included if the input is an arbitrary set of documents or the result of a search that doesn’t match text, such as a collection or directory query.

You must include a query in the snippet configuration when using `documentSelect` so the snippeter has search matches against which to generate snippets. You can also include the other properties described in “Configuring the Built-In Snippet Generator” on page 334.

The following example uses `documentSelect` to generate snippets from the result of calling `cts.search` (instead of `jsearch.documents`).

```
import jsearch from '/MarkLogic/jsearch.mjs';
const myQuery =
    cts.andQuery([
      cts.directoryQuery('/books/'),
```

```
      cts.jsonPropertyWordQuery('synopsis', 'california')])
jsearch.documentSelect(
  cts.search(myQuery),
  {snippet: {query: myQuery}})
```

7.9 Extracting Portions of Each Matched Document

You can use the built-in mapper of document search to return selected portions of each document that matches a search. You can use the extraction feature with `jsearch.documents` and `jsearch.documentSelect`.

This section includes the following topics:

- Extraction Overview
- How selected Affects Extraction
- Combining Extraction With Snippeting

7.9.1 Extraction Overview

By default, a document search returns the complete document for each search match. You can use `extract` feature of the built-in `documents` mapper to extract only selected parts of each matching document instead. Such a subset of the content in a document is sometimes called a *sparse document projection*. This feature is similar to the query option `extract-document-data`. available to the XQuery Search API and the Client APIs.

You use XPath expressions to identify the portions of the document to include or exclude. XPath is a standard expression language for addressing XML content. MarkLogic has extended XPath so you can also use it to address JSON. For details, see Traversing JSON Documents Using XPath in the *Application Developer's Guide* and XPath Quick Reference in the *XQuery and XSLT Reference Guide*.

To generate sparse projections, configure the `extract` property of the built-in mapper of a document search. The property has the following form:

```
extract: {
  paths: xPathExpr | [xPathExprs],
  selected: 'include' | 'include-with-ancestors' | 'exclude' | 'all'
}
```

Specify one or more XPath expressions in the `paths` value; use an array for specifying multiple expressions. The `selected` property controls how the content selected by the paths affects the document projection. The `selected` property is optional and defaults to `'include'` if not present; for details, see "How selected Affects Extraction" on page 339.

For example, the following code extracts just the `title` and `author` properties of documents containing the word "California" in the `synopsis` property.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(jsearch.byExample({synopsis: {$word: 'California'}}))
  .map({extract: {paths: ['/title', '/author']}})
  .result()
```

The table below displays the default output of the query (without a mapper) on the left and the result of using the example extraction on the right. Notice that the `document` property that contains the complete document contents has been replaced with an `extracted` property that contains just the requested subset of content.

Default Output	With Extract

{ "results": [
  { "index": 0,
  "uri": "/books/steinbeck1.json",
  "score": 34816,
  "confidence": 0.54882669448852,
  "fitness": 0.6809344291687,
  "document": {
  "title": "The Grapes of Wrath",
  "author": "John Steinbeck",
  "edition": {
  "format": "paperback",
  "price": 10
  },
  "synopsis": "Chronicles the 1930s Dust Bowl migration of one Oklahoma farm family, from their homestead and forced to the promised land of California."
  }
  }, ...*additonal results*...],
  "estimate": 3
}
	{ "results": [
	  { "index": 0
	  "uri": "/books/steinbeck1.json",
	  "score": 18432,
	  "confidence": 0.4903561770916,
	  "fitness": 0.71398365497589,
	  "path": "fn:doc(\"/books/steinb...",
	  "extracted": [
	  { "title": "The Grapes of Wrath" },
	  { "author": "John Steinbeck" }
	  ],
	  }, ...*additional results*...],
	  "esitmate": 3
	}

When extracting XML content that uses namespaces, you can use namespace prefixes in your extract paths. Define the prefix bindings in the namespaces property of the mapper configuration object. For example, the following configuration binds the prefix “my” to the namespace URI “/my/namespace”, and then uses the “my” prefix in an extract path.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documentSelect(fn.doc('/books/some.xml'),
  {
    namespaces: {my: '/my/namespace'},
    extract: {paths: ['/my:book/my:title']}
  })
```

Since the extraction feature is a capability of the built-in mapper for a document search, you cannot use it when using a custom mapper. If you want to return document subsets when using a custom mapper, you must construct the projections yourself.

For more details on using and configuring mappers, see "Transforming Results with Map and Reduce" on page 343.

7.9.2 How selected Affects Extraction

The `selected` property of the `extract` configuration for `DocumentsSearch.map` determines what to include in the extracted content. By default, the extracted content includes only the content selected by the path expressions. However, you can use the `select` property to configure these alternatives:

- include enclosing objects or elements (ancestors) in addition to the named nodes
- exclude the specified nodes rather than include them
- include all nodes, effectively ignoring the specified paths and including the whole document

For example, the documents loaded by "Preparing to Run the Examples" on page 375 have the following form:

```
{ "title": string,
  "author": string,
  "edition": {
    "format": string,
    "price": number
  },
  "synopsis": string
}
```

The table below illustrates how various `selected` settings affect the extraction of the `title` and `price` properties. The first row (`'include'`) also represents the default behavior when `selected` is not explicitly set.

`extract` Configuration	`extracted` Value

{extract: {
  paths: ['/title','/price'],
  selected: 'include'
}}
	"extracted":[
	  {"title": "The Grapes of Wrath"},
	  {"price": 10}
	]

{extract: {
  paths: ['/title','/price'],
  selected: 'include-with-ancestors'
}}
	"extracted":[{
	  "title": "The Grapes of Wrath",
	  "edition": {"price":10}
	}]

`extract` Configuration	`extracted` Value

`{extract: {`
`  paths: ['/title','/price'],`
`  **selected: 'exclude'**`
`}}`
	`"extracted":[{`
	`  "author": "John Steinbeck",`
	`  "edition": {"format":"paperback"},`
	`  "synopsis": ...`
	`}]`

`{extract: {`
`  paths: ['/title','/price'],`
`  **selected: 'all'**`
`}}`
	`"extracted":[{`
	`  "title":"The Grapes of Wrath",`
	`  "author":"John Steinbeck",`
	`  "edition":{`
	`  "format":"paperback",`
	`  "price":10`
	`  },`
	`  "synopsis": ...`
	`}]`

If the combination of `paths` and `select` selects no content for a given document, then the results contain an `extractedNone` property instead of an `extracted` property. For example:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(jsearch.byExample({synopsis: {$word: 'California'}}))
  .map({extract: {paths: ['/no/matches'], selected: 'include'}})
  .result()

==>

{"results":[
  { ...,
    "extractedNone":true,
    ...
}]}
```

7.9.3 Combining Extraction With Snippeting

By default, snippets are not generated when you use extraction, but you can configure your search to return both snippets and extracted content by setting `snippet` to true in the mapper configuration. For example, the following search:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(jsearch.byExample({synopsis: {$word: 'California'}}))
  .map({snippet: true, extract: {paths: ['/title', '/author']}})
  .result()
```

Produces output similar to the following, with the document projects in the extracted property and the snippets in the matches property:

```
{ "results": [
  { "score": 18432,
    "fitness": 0.71398365497589,
    "uri": "/books/steinbeck1.json",
    "path": "fn:doc(\"/books/steinbeck1.json\")",
    "extracted": [
      { "title": "The Grapes of Wrath" },
      { "author": "John Steinbeck" }
    ],
    "confidence": 0.4903561770916,
    "index": 0,
    "matches": [{
      "path": "fn:doc(\"/books/steinbeck1.json\")/text(\"synopsis\")",
      "matchText": [
        "...from their homestead and forced to the promised land of ",
        { "highlight": "California" },
        "."
      ]
    }]
  }, ...]
}
```

Similarly, you can include both `snippet` and `extract` specifications in the configuration for `jsearch.documentSelect`. For example:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documentSelect(
  cts.search(cts.jsonPropertyWordQuery('synopsis', 'California')),
  {snippet: {
    query: cts.jsonPropertyWordQuery('synopsis', 'California') }
   extract: {paths: ['/title', '/author'], selected: 'include'}
  }
)
```

For more details on snippeting, see “Including Snippets of Matching Content in Search Results” on page 332.

7.10 Using Options to Control a Query

You can control a document search with options in two ways:

- Specify query-specific options during construction of a query.
- Specify search-wide options using the `DocumentsSearch.withOptions` method.

Other JSearch operations, such as lexicon searches, use a similar convention for passing options to a specific query or applying them to the entire operation.

For example, the following query uses the query-specific `$exact` option of QBE to disable exact match semantics on the value query constructed with `jsearch.byExample`. However, this setting has no effect on the query constructed by `cts.jsonPropertyValueQuery` or on the top level `cts.orQuery`.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(cts.orQuery([
    jsearch.byExample({author: {$value: 'mark twain', $exact: false}}),
    cts.jsonPropertyValueQuery('author', 'john steinbeck')
  ]))
  .result()
```

The available per-query options depend on the type of query. The mechanism for specifying per-query options depends on the construction method you choose. For details, consult the appropriate API reference.

For example, `cts.jsonPropertyValueQuery` accepts a set of options as a parameter. through these options you can control attributes such as whether or not to enable stemming:

```
cts.jsonPropertyValueQuery(
  'author', 'mark twain', ['case-insensitive', 'lang=en'])
```

Options that can apply to the entire search are specified using the `withOptions` method. For example, you can use `withOptions` to pass options to the underlying `cts.search` operation of a documents search:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(cts.jsonPropertyValueQuery('author','mark twain'))
  .withOptions({search: ['format-xml','score-logtf']})
  .result()
```

For more details, see the following methods:

- `DocumentsSearch.withOptions`
- `ValuesSearch.withOptions`
- `TuplesSearch.withOptions`
- `WordsSearch.withOptions`
- `FacetsSearch.withOptions`
- `FacetDefinition.withOptions`

Note that, specifically in the case of passing options through to `cts.search`, some commonly used options are surfaced directly through JSearch methods, such as the `DocumentsSearch.filter` method. You should use the JSearch mechanism when this overlap is present.

7.11 Transforming Results with Map and Reduce

The top level JSearch query options such as documents, values, tuples, and words include map and reduce methods you can use to tailor the final results in a variety of ways, such as including snippets in a document search or applying a content transformation.

This section includes the following topics:

- Map and Reduce Overview
- Configuring the Built-In Mapper
- Using a Custom Mapper
- Configuring the Built-In Reducer
- Using a Custom Reducer
- Example: Returning Only Documents
- Example: Using a Custom Mapper for Content Transformation
- Example: Custom Reducer For Document Search
- Example: Custom Reducer For Values Query

7.11.1 Map and Reduce Overview

The top level JSearch operations for document search (`documents`) and lexicon queries (`values`, `tuples`, and `words`) include `map` and `reduce` methods for customizing your query results. You can choose to use either `map` or `reduce`, but not both.

A mapper takes in a single value and produces zero results or one result. The mapper is invoked once for each item (search result, value, or tuple) processed by the query operation. The output from the mapper is pushed on to the results array. A mapper is well suited for applying transformations to results.

In broad strokes, a reducer takes in a previous result and a single value and returns either an item to pass to next invocation of the reducer, or a final result. The output from the final invocation becomes the result. Reducers are well suited for computing aggregates over a set of results.

You can supply a custom mapper or reducer by passing a function reference to the `map` or `reduce` method. Some operations also have a built-in mapper and/or reducer that you can invoke by passing a configuration object in to the `map` or `reduce` method. For example, the built-in mapper for document search can be used to generate snippets.

Thus, your `map` or `reduce` call can have one of the following forms:

```
// configure the built-in mapper, if supported
.map({configProperties...})
```

```
// use a custom mapper
.map(function (currentItem) {...})

// configure the built-in reducer, if supported
.reduce({configProperties...})

// use a custom reducer
.reduce(function (prevResult, currentItem, index, state) {...})
```

The available configuration properties and behavior of the built-in mapper and reducer depend on the operation you apply map or reduce to. For details, see "Configuring the Built-In Mapper" on page 344.

The following methods support map and reduce operations. For configuration details, see the *MarkLogic Server-Side JavaScript Function Reference*.

- `DocumentsSearch.map` and `DocumentsSearch.reduce`
- `FacetDefinition.map` and `FacetDefinition.reduce`
- `ValuesSearch.map` and `ValuesSearch.reduce`
- `TuplesSearch.map` and `TuplesSearch.reduce`
- `WordsSearch.map` and `WordsSearch.reduce`

7.11.2 Configuring the Built-In Mapper

The capabilities of the built-in mapper vary, depending on the type of query operation (`documents`, `values`, or `tuples`). For example, the built-in mapper for a document search can be configured to generate snippets and document projections, while the built-in mapper on a values query can be configured to include frequency values in the results.

Configure the built-in mapper by passing a configuration object to the `map` method instead of a function reference. For example, the following call chain configures the built-in mapper for document search to return snippets:

```
jsearch.documents().map({snippet:true}).result()
```

The table below outlines the capabilities of the built-in mapper for each JSearch query operation.

Operation	Built-In Mapper Capabilities
`documents`	Generation of snippets, document projections, and/or URIs for similar documents. For details, see "Including Snippets of Matching Content in Search Results" on page 332, "Extracting Portions of Each Matched Document" on page 337, and `DocumentsSearch.map` in the *MarkLogic Server-Side JavaScript Function Reference*.
`values`	Control and generation of frequency data in the results. Optionally, add labels to returned values and frequencies. For details, see `ValuesSearch.map` in the *MarkLogic Server-Side JavaScript Function Reference*.
`tuples`	Control and generation of frequency data in the results. Optionally, adds labels to returned tuples and frequencies. For details, `TuplesSearch.map` in the *MarkLogic Server-Side JavaScript Function Reference*.
`words`	None. The `words` operation only supports a custom mapper.

7.11.3 Using a Custom Mapper

You can supply a custom mapper to the map method of the `documents`, `values`, `tuples`, and `words` queries. To use a custom mapper, pass a function reference to the `map` method in your query call chain:

```
... .map(funcRef)
```

The mapper function must have the following signature and should produce either no result or a single result. If the function returns a value, it is pushed on to the final results array or iterator.

```
function (currentItem)
```

The *`currentItem`* parameter can be a search result, tuple, value, or word, depending on the calling context. For example, the mapper on a document search (the `documents` method) takes a single search result descriptor as input.

Any value returned by your mapper is pushed on to the "results" array.

The following example uses a custom mapper on a document search to add a property named "iWasHere" to each search result. The input in this case is the search result for one document.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(cts.jsonPropertyValueQuery('author','Mark Twain'))
  .map(function (value) {value.iWasHere = true; return value;})
```

```
  .result()

==>
{"results":[
   {"index":0,
    "uri":"/books/twain4.json",
    "score":14336,
    "confidence":0.3745157122612,
    "fitness":0.7490314245224,
    "document":{...},
    "iWasHere":true
   },
   {"index":1, ...},
   ...
  ],
  "estimate":4
}
```

Your mapper is not required to return a value. If you return nothing or explicitly return `undefined`, then the final results will contain no value corresonding to the current input item. For example, the following mapper eliminates every other search result:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .map(function (value) {
    if (value.index % 2 > 0) {
      return value;
    }
  })
  .result().results.length
```

If your database contains only the documents from "Preparing to Run the Examples" on page 375, then the script should produce the answer 4 when run in Query Console.

For an additional example, see "Example: Using a Custom Mapper for Content Transformation" on page 349.

7.11.4 Configuring the Built-In Reducer

The capabilities of the built-in reducer vary, depending on the type of query operation. Currently, only `values` offers a built-in reducer.

Configure the built-in reducer by passing a configuration object to the `reduce` method instead of a function reference. For example, the following configures the built-in reducer for a values query to return item frequency data along with the values:

```
jsearch.values('price').reduce({frequency: 'item'}).result()
```

The table below outlines the capabilities of the built-in reducer for each JSearch query operation.

Operation	Built-In Reducer Capabilities
`documents`	None. The `documents` operation only supports a custom reducer.
`values`	Control and generation of frequency data in the results. Optionally, adds labels to returned values and frequencies. For details, see `ValuesSearch.reduce` in the *MarkLogic Server-Side JavaScript Function Reference*.
`tuples`	None. The `tuples` operation only supports a custom reducer.
`words`	None. The `words` operation only supports a custom reducer.

7.11.5 Using a Custom Reducer

To use a custom reducer, pass a function reference and optional initial seed value to the `reduce` method of your query call chain:

```
... .reduce(funcRef, seedValue)
```

The reducer function must have the following signature:

```
function (prevResult, currentItem, index, state)
```

If you pass a seed value, it becomes the value of *`prevResult`* on the first invocation of your function. For example, the following reduce call seeds an accumulator object with initial values. On the first call to `myFunc`, *`prevResult`* contains `{count: 0, value: 0, matches: []}`.

```
... .reduce(myFunc, {count: 0, value: 0, matches: []}) ...
```

For example, the following call chain uses a custom mapper with an initial seed value as part of a document search.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(cts.jsonPropertyValueQuery('author','Mark Twain'))
  .reduce(function (prev, match, index, state) {
    // do something
  }, {count: 0, value: 0, matches: []})
  .result()
```

The value returned by the last invocation of your reducer becomes the final result of the query. You can detect or signal the last invocation through `state.isLast`.

The following table describes the inputs to the reducer function:

Parameter	Description
prevResult	The value returned by the previous invocation of your function during this reduction. If a seed value is passed to `reduce`, then the seed is the value of *prevResult* on the first invocation. Otherwise, *prevResult* is null on the first invocation.
currentItem	The current value to act upon. The structure of the value depends on the calling context: • word: The current word. • documents: The search result object. • values: The current value. • tuples: The current n-way co-ocurrence tuple.
index	The zero-based index of the *currentItem* in the set of items being iterated over.
state	An object describing the state of the reduction. It contains an `isLast` property that is true only if this is the last invocation of the reducer for this reduction. You can explicitly set `isLast` to true to force early termination.

Note that the `map` and `reduce` methods are exclusive of one another. If your query uses `reduce`, it cannot use `map`.

For more examples, see the following:

- “Example: Custom Reducer For Document Search” on page 351
- “Example: Custom Reducer For Values Query” on page 353

7.11.6 Example: Returning Only Documents

The following example uses a custom mapper to strip everything out of the results of a document search except the matched document. For more details, see “Using a Custom Mapper” on page 345 and `DocumentsSearch.map`.

By default, a document search returns a structure that includes metadata about each match, such as uri and score, as well as the matched document. For example:

```
{ "results":[
  { "index":0,
    "uri":"/books/frost1.json",
```

```
    "score":22528,
    "confidence":0.560400724411011,
    "fitness":0.698434412479401,
    "document": ...document node...
  }, ... ]}
```

The following code uses a custom mapper (expressed as a lambda function) to eliminate everything except the value of the document property. That is, it eliminates everything but the matched document node.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(cts.jsonPropertyValueQuery('format','paperback'))
  .slice(0,2)
  .map(match => match.document)
  .result()
```

The result is output similar to the following:

```
{ "results": [
  { "title": "Collected Works",
    "author": "Robert Frost",
    "edition": {
      "format": "paperback",
      "price": 29.99
    },
    "synopsis": "The complete works of the American Poet Robert Frost."
  },
  { "title": "The Grapes of Wrath",
    "author": "John Steinbeck",
    "edition": {
      "format": "paperback",
      "price": 9.99
    },
    "synopsis": "Chronicles the 1930s Dust Bowl migration of
                 one Oklahoma farm family, from their homestead
                 and forced to the promised land of California."
  }],
  "estimate": 4
}
```

The custom mapper lambda function (`.map(match => match.document)`) is equivalent to the following:

```
.map(function(match) { return match.document; })
```

7.11.7 Example: Using a Custom Mapper for Content Transformation

The following example demonstrates using a custom mapper to transform document content returned by a search. For more details, see "Using a Custom Mapper" on page 345 and `DocumentsSearch.map`.

The following example code uses a custom mapper to redact the value of the JSON property "author" in each document matched by the search.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(cts.jsonPropertyValueQuery('format','paperback'))
  .slice(0,2)
  .map(function (match) {
    match.document = match.document.toObject();
    match.document.author = 'READACTED';
    return match;
  })
  .result()
```

Each time the mapper is invoked, the "author" property value is changed to "REDACTED" in the document embedded in the search result. Notice the application of `toObject` to the document:

```
match.document = match.document.toObject();
```

This is necessary because `match.document` is initially a read-only document node. Applying `toObject` to the document node creates an in-memory, mutable copy of the contents.

If your database contains the documents created by "Preparing to Run the Examples" on page 375, then running the script produces output similar to the following. The part of each result affected by the mapper is shown in bold. Only two results are returned because of the `slice`(0,2) clause on the search.

```
{ "results": [
    { "index": 0,
      "uri": "/books/frost1.json",
      "score": 14336,
      "confidence": 0.43245348334312,
      "fitness": 0.7490314245224,
      "document": {
        "title": "Collected Works",
        "author": "REDACTED",
        "edition": {
          "format": "paperback",
          "price": 29.99
        },
        "synopsis": "The complete works of the American Poet
                     Robert Frost."
      }
    },
    { "index": 1,
      "uri": "/books/steinbeck1.json",
      "score": 14336,
      "confidence": 0.43245348334312,
      "fitness": 0.7490314245224,
      "document": {
        "title": "The Grapes of Wrath",
```

```
          "author": "REDACTED",
          "edition": {
            "format": "paperback",
            "price": 9.99
          },
          "synopsis": "Chronicles the 1930s Dust Bowl migration of
                       one Oklahoma farm family, from their homestead
                       and forced to the promised land of California."
        }
      }
    ],
    "estimate": 4
}
```

7.11.8 Example: Custom Reducer For Document Search

The following example demonstrates using `DocumentsSearch.reduce` to apply a custom reducer as part of a document search.

The search selects a random sample of 1000 documents by setting the search scoring algorithm to "score-random" in `withOptions`. and the slice size to 1000 with `slice`. Notice that there is no `where` clause, so the search matches all documents in the database.

The following code snippet is the core search that drives the reduction:

```
jsearch.documents()
  .slice(0, 1000)
  .reduce(...)
  .withOptions({search: 'score-random'})
  .result();
```

The reducer iterates over the node names (JSON property names or XML element names) in each document, adding each name to a map, along with a corresponding counter.

```
function nameExtractor(previous, match, index, state) {
  const nameCount = 0;
  for (const name of match.document.xpath('//*/fn:node-name(.)')) {
    nameCount = previous[name];
    previous[name] = (nameCount > 0) ? nameCount + 1 : 1;
  }
  return previous;
  }
```

Each time the reducer is invoked, the `match` parameter contains the search result for a single document. That is, input of the following form. The precise properties in the input object can vary somewhat, depending on the search options.

```
{ index: 0,
  uri: '/my/document/uri',
  score: 14336,
  confidence: 0.3745157122612,
```

```
  fitness: 0.7490314245224,
  document: { documentContents }
}
```

The following code puts all of the above together in a complete script. Notice that an empty object ({ }) is passed to reduce as a seed value for the initial value of the `previous` input parameter.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .slice(0, 1000)
  .reduce(function nameExtractor(previous, match, index, state) {
      const nameCount = 0;
      for (const name of match.document.xpath('//*/fn:node-name(.)')) {
        nameCount = previous[name];
        previous[name] = (nameCount > 0) ? nameCount + 1 : 1;
      }
      return previous;
    }, {})
  .withOptions({search: 'score-random'})
  .result();
```

Running this script with the documents created by “Preparing to Run the Examples” on page 375 produces output similar to the following.

```
{"results":{
   "title":8,
   "author":8,
   "edition":8,
   "format":8,
   "price":8,
   "synopsis":8
 },
 "estimate":8}
```

The property names are the JSON property names found in the sample documents. The property values are the number of occurrencesoccurrences of each name in the sampled documents. The values in this case are all the same because all the sample documents contain exactly the same properties. However, if you run the query on a less homogeneous set of documents you might get results such as the following:

```
{"results":{
    "Placemark":52,
    "name":53,
    "Style":52,
    "ExtendedData":52,
    "SimpleData":208,
    "Polygon":574,
    "coordinates":610,
    "MultiGeometry":24,
  },
  "estimate":58
}
```

If you want to retain the search results along with whatever computation is performed by your reducer, you must accumulate them yourself. For example, the reducer in the following script accumulates the results in an array in the result object:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(cts.jsonPropertyValueQuery('author','Mark Twain'))
  .reduce(function (prev, match, index, state) {
    prev.count++;
    prev.value += match.document.edition.price;
    prev.matches.push(match);
    if (state.isLast) {
      return {avgCost: prev.value / prev.count, matches: prev.matches};
    } else {
      return prev;
    }
  }, {count: 0, value: 0, matches: []})
  .result()
```

When run against the sample data from “Preparing to Run the Examples” on page 375, the output is similar to the following:

```
{"results":{
    "avgCost": 13.25,
    "matches": [{"index":0, "uri": ...}, ...more matches...]
  },
  estimate: 4
}
```

7.11.9 Example: Custom Reducer For Values Query

This example demonstrates using `ValuesSearch.reduce` to apply a custom reducer that computes an aggregate value from the results of a values query. The example relies on the sample data from “Preparing to Run the Examples” on page 375.

The query that produces the inputs to the reduction is a values query over the `price` JSON property. The database configuration should include a range index over `price` with scalar type `float`. The scalar type of the index determines the datatype of the value passed into the second parameter of the reducer.

The following code computes an average of the values of the `price` JSON property. Each call to the reducer accumulates the count and sum contributing to the final answer. When `state.isLast` becomes true, the final aggregate value is computed and returned. The reduction is seeded with an initial accumulator value of `{count: 0, sum: 0}`, through the second parameter passed to `reduce`.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.values('price')
  .where(cts.directoryQuery('/books/'))
  .reduce(function (accum, value, index, state) {
```

```
    const freq = cts.frequency(value);
    accum.count += freq;
    accum.sum += value * freq;
    return state.isLast ? (accum.sum / accum.count) : accum;
  }, {count: 0, sum: 0})
.result();
```

If you run the query in Query Console using the data from "Preparing to Run the Examples" on page 375, you should see output similar to the following:

```
16.125
```

Notice the use of `cts.frequency` in the example. The reducer is called once for each unique value in the index. If you're doing a reduction that depends on frequency, use `cts.frequency` on the input value to get this information.

Average and sum are only used here as a convenient simple example. In practice, if you needed to compute the average or sum, you would use built-in aggregate functions. For details, see "Computing Aggregates Over Range Indexes" on page 362.

7.12 Querying Lexicons and Range Indexes

- Querying the Values in a Lexicon or Index
- Finding Value Co-Occurrences in Lexicons and Indexes
- Querying Values in a Word Lexicon
- Computing Aggregates Over Range Indexes
- Constructing Lexicon and Range Index References

7.12.1 Querying the Values in a Lexicon or Index

Use `jsearch.values` to begin building a query over the values in a values lexicon or range index, and then use `result` to execute the query and return results. You can also use the `values` method to compute aggregates lexicon and index values; for details, see "Computing Aggregates Over Range Indexes" on page 362.

For example, the following code creates a values query over a range index on the "title" JSON property. The returned values are limited to those found in documents matching a directory query (`where`) and those that match the pattern "*adventure*" (`match`). The results are returned in frequency order (`orderBy`). Only the first 3 results are returned (`slice`).

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.values('title')
  .where(cts.directoryQuery('/books/'))
  .match('*adventure*')
  .orderBy('frequency')
```

```
    .slice(0,3)
    .result()
```

This query produces the following output when run against the sample data from “Preparing to Run the Examples” on page 375.

```
["Adventures of Huckleberry Finn", "Adventures of Tom Sawyer"]
```

Your database configuration must include an index or range index on each JSON property, XML element, XML element attribute, field, or path used in a `values` query.

For general information on lexicon queries, see “Browsing With Lexicons” on page 445.

Build and execute your values query following the pattern described in “Query Design Pattern” on page 291. The following table maps the applicable JSearch methods to the steps in the design pattern. Note that all the pipeline stages in Step 2 are optional, but you must use them in the order shown. For more details, see ValuesSearch in the *MarkLogic Server-Side JavaScript Function Reference*.

Pattern Step		Method(s)	Notes
1	Select resource	`values`	Required. Select index and lexicon values as the resource to work with. Supply one or more lexicon or index references or JSON property names as input to `values`.

Pattern Step		Method(s)	Notes
2	Add a query definition and result set pipeline	`where`	Optional. Constrain the set of results (and frequency computation) to values from documents matching a query, as described in "Constraining Lexicon Searches to a cts:query Expression" on page 456. If you pass in multiple queries, they are implicitly AND'd together. You can create a cts.query from a QBE, query text, cts.query constructors, or any other technique that creates a `cts.query`. For details, see "Creating a cts.query" on page 300.

		`match` | `groupInto`	Optional. You cannot use `match` and `groupInto` together. Use `match` to limit values to those matching a wildcard pattern. For example:
			
			`jsearch.values('title')`
			`  .where(cts.directoryQuery('/books/'))`
			`  .match('*adventure*')`
			
			Use `groupInto` to group values into value range buckets. For details and examples, see "Grouping Values and Facets Into Buckets" on page 367.

		`orderBy`	Optional. Specify the order of results. You can choose whether to order by frequency or item value, and ascending or descending order. For details, see "Controlling the Ordering of Results" on page 328
		`slice`	Optional. Select a subset of values from the result set. The default slice is the first 10 values. For details, see "Returning a Result Subset" on page 331.
		`map` | `reduce`	Optional. Apply a mapper or reducer function to the results. You cannot use `map` and `reduce` together. For details, see "Transforming Results with Map and Reduce" on page 343.
3	Add advanced options	`withOptions`	Optional. Specify additional, advanced options that customize the query behavior. For details, see "Using Options to Control a Query" on page 341 and `ValuesSearch.withOptions`.
4	Evaluate the query and get results	`result`	Required. Execute the query and receive your results, optionally specifying whether to receive the results as a value or an `Iterable`. The default is a value (typically an array).

7.12.2 Finding Value Co-Occurrences in Lexicons and Indexes

Use the `jsearch.tuples` method to find co-occurrences of values in lexicons and range indexes. Use `tuples` to begin building your query, and then use `result` to execute the query and return results. You can also use the `tuples` method to compute aggregates over tuples; for details, see "Computing Aggregates Over Range Indexes" on page 362.

For example, the following code creates a tuples query for 2-way co-occurences of the values in the "author" and "format" JSON properties. Only tuples in documents matching the directory query are considered (`where`). The results are returned in item order (`orderBy`).

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.tuples(['author','format'])
  .where(cts.directoryQuery('/books/'))
  .orderBy('item')
  .result()
```

This query produces the following output when applied to the data from "Preparing to Run the Examples" on page 375.

```
[["John Steinbeck", "audiobook"], ["John Steinbeck", "hardback"],
 ["John Steinbeck", "paperback"], ["Mark Twain", "hardback"],
 ["Mark Twain", "paperback"], ["Robert Frost", "paperback"]]
```

Your database configuration must include an index or range index on each JSON property, XML element, XML element attribute, field, or path used in a `tuples` query.

Build and execute your tuples query following the pattern described in "Query Design Pattern" on page 291. The following table maps the applicable JSearch methods to the steps in the design pattern. Note that all the pipeline stages in Step 2 are optional, but you must use them in the order shown. For more details, see TuplesSearch in the *MarkLogic Server-Side JavaScript Function Reference*.

Pattern Step		Method(s)	Notes
1	Select resource	`tuples`	Required. Select index and lexicon value co-occurrences as the resource to work with. Supply one or more lexicon or index references or JSON property names as input to `values`.
2	Add a query definition and result set pipeline	`where`	Optional. Constrain the set of tuples (and frequency computation) to values in documents matching a query, as described in "Constraining Lexicon Searches to a cts:query Expression" on page 456. If you pass in multiple queries, they are implicitly AND'd together. You can create a cts.query from a QBE, query text, cts.query constructors, or any other technique that creates a `cts.query`. For details, see "Creating a cts.query" on page 300.
		`orderBy`	Optional. Specify the order of results. You can choose whether to order by frequency or item value, and ascending or descending order. For details, see "Controlling the Ordering of Results" on page 328
		`slice`	Optional. Select a subset of tuples from the result set. The default slice is the first 10 tuples. For details, see "Returning a Result Subset" on page 331.
		`map	reduce`	Optional. Apply a mapper or reducer function to the results. You cannot use `map` and `reduce` together. For details, see "Transforming Results with Map and Reduce" on page 343.
3	Add advanced options	`withOptions`	Optional. Specify additional, advanced options that customize the query behavior. For details, see "Using Options to Control a Query" on page 341 and `TuplesSearch.withOptions`.
4	Evaluate the query and get results	`result`	Required. Execute the query and receive your results, optionally specifying whether to receive the results as a value or an `Iterable`. The default is a value (typically an array).

7.12.3 Querying Values in a Word Lexicon

Use the `jsearch.words` method to create a word lexicon query, and then use `result` to execute the query and return results.

For example, the following code performs a word lexicon query for all words in the `synopsis` JSON property that begin with 'c' (`match`). Only occurrences in documents where the `author` property contains "steinbeck" (`where`) are returned. At most the first 5 words are returned (`slice`).

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.words('synopsis')
  .where(cts.jsonPropertyWordQuery('author', 'steinbeck'))
  .match('c*')
  .slice(0,5)
  .result();
```

When run against the data from "Preparing to Run the Examples" on page 375, this query produces the following output:

```
["Cain", "California", "Chronicles"]
```

Your database configuration must either enable the database-wide word lexicon or include a word lexicon on each JSON property, XML element, XML element attribute, or field used in a `words` query. For details on lexicon configuration, see Range Indexes and Lexicons in the *Administrator's Guide*.

For general information on lexicon queries, see "Browsing With Lexicons" on page 445.

Build and execute your word query following the pattern described in “Query Design Pattern” on page 291. The following table maps the applicable JSearch methods to the steps in the design pattern. Note that all the pipeline stages in Step 2 are optional, but you must use them in the order shown. For more details, see WordsSearch in the *MarkLogic Server-Side JavaScript Function Reference*.

Pattern Step		Method(s)	Notes

1	Select resource	`words`	Required. Select index and word lexicons as the resource to work with. Supply one or more lexicon or index references or JSON property names as input to `values`. For example:
			
			`// query word lexicon on a JSON property`
			`jsearch.words('synopsis'). ...`
			
			`// query the database wide word lexicon`
			`jsearch.words(jsearch.databaseLexicon()).`
			`...`
			
			`// query the word lexicon on an XML element`
			`jsearch.words(`
			`  jsearch.elementLexicon(`
			`  fn.QName(`
			`  'http://marklogic.com/example',`
			`  'myElem')))`

Pattern Step		Method(s)	Notes
2	Add a query definition and result set pipeline	`where`	Optional. Constrain the set of tuples (and frequency computation) to words in documents matching a query, as described in "Constraining Lexicon Searches to a cts:query Expression" on page 456. If you pass in multiple queries, they are implicitly AND'd together. You can create a cts.query from a QBE, query text, cts.query constructors, or any other technique that creates a `cts.query`. For details, see "Creating a cts.query" on page 300.
		`match`	Optional. Limit words to those matching a wildcard pattern. For example, the following match clause selects words beginning with 'c': `import jsearch from '/MarkLogic/jsearch.mjs'; jsearch.words('synopsis') .where(cts.directoryQuery('/books/')) .match('c*')`
		`orderBy`	Optional. Specify whether to list the results in ascending or descending order. For details, see "Controlling the Ordering of Results" on page 328
		`slice`	Optional. Select a subset of tuples from the result set. The default slice is the first 10 results. For details, see "Returning a Result Subset" on page 331.
		`map | reduce`	Optional. Apply a mapper or reducer function to the results. You cannot use `map` and `reduce` together. For details, see "Transforming Results with Map and Reduce" on page 343.
3	Add advanced options	`withOptions`	Optional. Specify additional, advanced options that customize the query behavior. For details, see "Using Options to Control a Query" on page 341 and `WordsSearch.withOptions`.
4	Evaluate the query and get results	`result`	Required. Execute the query and receive your results, optionally specifying whether to receive the results as a value or an `Iterable`. The default is a value (typically an array).

7.12.4 Computing Aggregates Over Range Indexes

You can compute aggregate values over range indexes and lexicons using built-in or user-defined aggregate functions using `ValuesSearch.aggregate` or `TuplesSearch.aggregate`. This section covers the following topics:

- Aggregate Function Overview
- Using Built-In Aggregate Functions
- Using Aggregate User-Defined Functions

7.12.4.1 Aggregate Function Overview

An aggregate function performs an operation over values or tuples in lexicons and range indexes. For example, you can use an aggregate function to compute the sum of values in a range index. You can apply an aggregate computation to the results of a values or tuples query using `ValuesSearch.aggregate` or `TuplesSearch.aggregate`.

MarkLogic Server provides built-in aggregate functions for many common analytical functions; for a list of functions, see "Using Built-In Aggregate Functions" on page 364. For a more detailed description of each built-in, see Using Builtin Aggregate Functions in the *Search Developer's Guide*.

You can also implement aggregate user-defined functions (UDFs) in C++ and deploy them as native plugins. Aggregate UDFs must be installed before you can use them. For details, see Implementing an Aggregate User-Defined Function in the *Application Developer's Guide*. You must install the native plugin that implements your UDF according to the instructions in Using Native Plugins in the *Application Developer's Guide*.

> **Note:** You cannot use the JSearch API to apply aggregate UDFs that require additional parameters.

Build and execute your aggregate computation following the pattern described in "Query Design Pattern" on page 291. The following table maps the applicable JSearch methods to the steps in the design pattern. Note that you must use the pipeline stages in Step 2 in the order shown. For more details, see ValuesSearch or TuplesSearch in the *MarkLogic Server-Side JavaScript Function Reference*.

Pattern Step		Method(s)	Notes
1	Select resource	`values | tuples`	Required. Select index and lexicon values or tuples (co-occurrences) as the resource to work with. Supply one or more lexicon or index references or JSON property names as input.
2	Add a query definition and result set pipeline	`where`	Optional. Constrain the values or tuples to values in documents matching a query, as described in "Constraining Lexicon Searches to a cts:query Expression" on page 456. If you pass in multiple queries, they are implicitly AND'd together. You can create a cts.query from a QBE, query text, cts.query constructors, or any other technique that creates a `cts.query`. For details, see "Creating a cts.query" on page 300.
		`aggregate`	Required. Specify one or more built-in or user-defined aggregate functions. You can combine built-in and user-defined aggregates in the same query. For details, see "Using Built-In Aggregate Functions" on page 364 and "Using Aggregate User-Defined Functions" on page 365.
3	Add advanced options	`withOptions`	Optional. Specify additional, advanced options that customize the query behavior. For details, see "Using Options to Control a Query" on page 341 and `ValuesSearch.withOptions` or `TuplesSearch.withOptions`.
4	Evaluate the query and get results	`result`	Required. Execute the query and receive your results, optionally specifying whether to receive the results as a value or an `Iterable`. The default is a value (typically an array).

7.12.4.2 Using Built-In Aggregate Functions

To use a builtin aggregate function, pass the name of the function to the `aggregate` method of a values or tuples query. The built-in aggregate functions only support tuples queries on 2-way co-occurrences. That is, you cannot use them on tuples queries involving more than 2 lexicons or indexes.

The following example uses built-in aggregate functions to compute the minimum, maximum, and average of the values in the `price` JSON property and produces the results shown. As with all values queries, the database must include a range index over the target property or XML element.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.values('price')
  .aggregate(['min','max','avg'])
  .result();

==> {"min":8, "max":30, "avg":16.125}
```

The following built-in aggregate functions are supported on values queries:

Values Aggregate Name	Description
`avg`	Compute the average of the values in a lexicon or range index. For details, see `cts.avgAggregate`.
`count`	Returns a count of the values in a lexicon or range index. For details, see `cts.countAggregate`.
`max`	Compute the maximum of the values in a lexicon or range index. For details, see `cts.max`.
`min`	Compute the minimum of the values in a lexicon or range index. For details, see `cts.min`.
`stddev`	Compute the frequency-weighted sample standard deviation of the values in a lexicon or range index. For details, see `cts.stddev`.

Values Aggregate Name	Description
`stddev-population`	Compute the frequency-weighted sample standard deviation of the population from the values in a lexicon or range index. For details, see `cts.stddevP`.
`sum`	Compute the sum of the values in a lexicon or range index. For details, see `cts.sumAggregate`.
`variance`	Compute the frequency-weighted sample variance of the values in a lexicon or range index. For details, see `cts.variance`.
`variance-population`	Compute the frequency-weighted variance of population of the values in a lexicon or range index. For details, see `cts.varianceP`.

The following built-in aggregate functions are supported on tuples queries:

Tuples Aggregate Name	Description
`correlation`	Compute the frequency-weighted correlation of 2-way co-occurences. For details, see `cts.correlation`.
`covariance`	Compute the frequency-weighted correlation of 2-way co-occurrences. For details, see `cts.covariance`.
`covariance-population`	Compute the frequency-weighted correlation of the population of 2-way co-occurrences. For details, see `cts.covarianceP`.

7.12.4.3 Using Aggregate User-Defined Functions

An aggregate UDF is identified by the function name and a relative path to the plugin that implements the aggregate, as described in “Using Aggregate User-Defined Functions” on page 465. You must install your UDF plugin on MarkLogic Server before you can use it in a query. For details on creating and installing aggregate UDFs, see Aggregate User-Defined Functions in the *Application Developer's Guide*.

Once you install your plugin, use `jsearch.udf` to create a reference to your UDF, and pass the reference to the `aggregate` clause of a values or tuples query. For example, the following script uses a native UDF called “count” provided by a plugin installed in the modules database under “native/sampleplugin”:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.values('price')
  .aggregate(jsearch.udf('native/sampleplugin', 'count'))
  .result();
```

For more details, see `ValuesSearch.aggregate` and `TuplesSearch.aggregate`.

7.12.5 Constructing Lexicon and Range Index References

This section provides a brief overview of the functions available for constructing the index and lexicon reference you may need for values queries, tuples queries, and facet generation.

Most JSearch interfaces that accept index or lexicon references also accept a simple JSON property name string. In most contexts, this is interpreted as a `cts.jsonPropertyReference` for a string property. If the referenced property (and associated index) have a type other than string, you can create a properly typed index reference as shown in these examples:

```
cts.jsonPropertyReference('price', ['type=float'])
cts.jsonPropertyReference('start', ['type=date'])
```

Similar reference constructors are available for XML element indexes, XML element attribute index, path indexes, field indexes, and geospatial property, element, and path indexes. The following is a small sample of the available constructors:

- `cts.elementReference`
- `cts.field-reference`
- `cts.path-reference`
- `cts.geospatialJsonPropertyReference`

Use the following reference constructors for the database-wide URI and collection lexicons. (These lexicons must be enabled on the database before you can use them.)

- `cts.uriReference`
- `cts.collectionReference`

JSearch also provides the following word lexicon reference constructors for constructing references to word lexicons specifically for use with `jsearch.words`. Using these constructors ensures you only create word lexicons queries on lexicon types that support them.

- `jsearch.databaseLexicon`
- `jsearch.jsonPropertyLexicon`
- `jsearch.elementLexicon`
- `jsearch.elementAttributeLexicon`
- `jsearch.fieldLexicon`

For more details, see the *MarkLogic Server-Side JavaScript Function Reference* and “Browsing With Lexicons” on page 445.

7.13 Grouping Values and Facets Into Buckets

This section covers the following topics related to using the `ValuesSearch.groupInto` and `FacetDefinition.groupInto` to group values by range:

- Bucketing Overview
- Example: Generating Buckets With makeBuckets
- Example: Grouping Using Custom Buckets

7.13.1 Bucketing Overview

You can use the `groupInto` method to group values into ranges when performing a values query or generating facets. Such grouping is sometimes called "bucketed search". The `groupInto` method of `values` and `facets` has the following form:

```
groupInto(bucketDefinition)
```

You can apply `groupInto` to a values query or a facet definition. For example:

```
// using groupInto with a values query
jsearch.values(...).groupInto(bucketDefinition).result()

// using groupInto for facet generation
jsearch.facets(
  jsearch.facet(...).groupInto(bucketDefinition),
  ...more facet definitions...
).result()
```

A bucket definition can be an array of boundary values or an array of alternating bucket names and boundary value pairs. For geospatial buckets, a boundary value can be an object with lat and lon properties (`{lat: latVal, lon: lonVal}`). The JSearch API includes helper functions for creating bucket names (`jsearch.bucketName`), generating a set of buckets from a value range and step (`jsearch.makeBuckets`), and generating buckets corresponding to a geospatial heatmap (`jsearch.makeHeatmap`).

Buckets can be unnamed, use names generated from the boundary values, or use custom names. For example:

```
// Unnamed buckets with boundaries X < 10, 10 <= X < 20, and X > 20
groupInto([10,20])

// The same set of buckets with generated default bucket names
groupInto([
  jsearch.bucketName(),10,
  jsearch.bucketName(),20,
  jsearch.bucketName()])

// The same set of buckets with custom bucket names
```

```
groupInto([
  jsearch.bucketName('under $10'), 10,
  jsearch.bucketName('$10 to $19.99'), 20,
  jsearch.bucketName('over $20')])

// Explictly specify geospatial bucket boundaries
groupInto([
  jsearch.bucketName(),{lat: lat1, lon: lon1,
  jsearch.bucketName(),{lat: lat2, lon: lon2,
  jsearch.bucketName(),{lat: lat3, lon: lon3}])
```

You can create a bucket definition in the following ways:

- Define a set of unnamed buckets by creating an array of boundary values. For example, `[10,20]` defines 3 buckets with boundaries `X < 10`, `10 <= X < 20`, and `X > 20`.
- Define a set of named buckets by creating an array of (*`bucketName`*, *`upperBound`*) pairs. Use the `buckeName` helper function to generate the name of each bucket. You can specify custom bucket names or groupInto generate bucket names from the boundary values.
- Use the `makeBuckets` helper function to create a set of buckets over a range of values (min and max) and a step or number of divisions. For example, create a series of buckets that each correspond to a decade over a 100 year time span.
- Use the `makeHeatMap` helper function to generate buckets from a geospatial lexicon based on a heatmap box with latitude and longitude divisions.

The bounds for bucket for a scalar value or date/time range are determined by an explicit upper bound and the position of the bucket in a set of bucket definitions. For example, in the following custom bucket definition, each line represents one bucket as a name and upper bound.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.values('price')
  .groupInto([
    jsearch.bucketName(), 10,
    jsearch.bucketName(), 20,
    jsearch.bucketName()])
  .result()
```

The first bucket has no lower bound because it occurs first. The lower bound of the second bucket is the upper bound of the previous bucket (10), inclusive. The upper bound of the second bucket is 20, exclusive. The last bucket has no upper bound. When plugged into a values or facets query, the results are grouped into the following ranges:

```
x < 10
10 <= x < 20
20 <= x
```

For geospatial data, you can use `makeHeatMap` to sub-divide a region into boxes. For example, the following constraint includes a heat map that corresponds very roughly to the continental United States, and divides the region into a set of 20 boxes (5 latitude divisions and 4 longitude divisions).

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.values('incidents')
  .groupInto(makeHeatMap({
    north: 49.0,
    east: -67.0,
    south: 24.0,
    west: -125.0,
    lonDivs: 4,
    latDivs: 5
  }))
  .result()
```

When combined with a reducer that returns frequency, you can use the resulting set of boxes and frequencies to illustrate the concentration of points in each box, similar to a grid-based heat map.

You can create more customized geospatial buckets by specifying a series of latitude bounds and longitude bounds that define a grid in an object of the form `{lat:[...], lon:[...]}`. The points defined by the latitude bounds and longitude bounds are divided into box-shaped buckets. The lat and lon values must be ascending order. For example:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.values('incidents')
  .groupInto({lat: [15, 30, 45, 60, 75], lon: [0, 30, 60, 90, 120]})
  .result()
```

For more details, see `jsearch.makeHeatmap`, `cts:geospatial-boxes`, and "Creating Geospatial Facets" on page 525.

7.13.2 Example: Generating Buckets With makeBuckets

The examples in this section demonstrate the following features:

- Using `jsearch.makeBuckets` to generate buckets for a values query.
- Using `jsearch.makeBuckets` to generate bucketed facets.
- Using a custom mapper to decorate your buckets.

The example uses `makeBuckets` to group date information by month, leveraging MarkLogic's built-in support for date, time and duration data..

The example assumes the following conditions exist in the database:

- The database contains documents of the following form describing events. Each document includes a `start` property that represents the start date of the event.

```
{ title: 'San Francisco Ocean Film Festival',
  venue: 'Fort Mason, San Francisco',
  start: '2015-02-27',
  end: '2015-03-01'
}
```

- All the event documents of interest are in a collection with the URI ‘events’.
- The database configuration includes an element range index of type ‘date’ on the `start` proeprty.

The following query groups the values in the lexicon for the year 2015 by month, using `jsearch.makeBuckets` and `ValuesSearch.groupInto`. The results include frequency data in each bucket.

```
import jsearch from '/MarkLogic/jsearch.mjs';
const events = jsearch.collections('events');
events.values(cts.jsonPropertyReference('start', ['type=date']))
  .groupInto( jsearch.makeBuckets({
    min: xs.date('2015-01-01'),
    max: xs.date('2015-12-31'),
    step: xs.yearMonthDuration('P1M')}))
  .map({frequency: 'item', names: ['bucket', 'count']})
  .result()
```

Notice the use of a 1 month duration (‘P1M’) for the step between buckets. You can use many MarkLogic date, dateTime, and duration operations from Server-side JavaScript. For details, see JavaScript Duration and Date Arithmetic and Comparison Methods in the *JavaScript Reference Guide*.

The query generates results similar to the following:

```
[ {
    "bucket": {
      "minimum": "2015-02-27",
      "maximum": "2015-02-27",
      "lowerBound": "2015-02-01",
      "upperBound": "2015-03-01"
    },
    "count": 1
  },
  {
    "bucket": {
      "minimum": "2015-03-07",
      "maximum": "2015-03-14",
      "lowerBound": "2015-03-01",
      "upperBound": "2015-04-01"
    },
```

```
    "count": 2
  },
  ...
]
```

You can use a custom mapper to name each bucket after the month it covers. Note that plugging in a custom mapper also eliminates the frequency data, so you must add it back in explicitly. The following example mapper adds a month name and count property to each bucket:

```
// For mapping month number to user-friendly bucket name
const months = [
  'January', 'February', 'March',
  'April', 'May', 'June',
  'July', 'August', 'September',
  'October', 'November', 'December'
  ];

// Add a name and count field to each bucket. Use month for name.
function supplementBucket(bucket) {
  // get a mutable copy of the input
  const result = bucket.toObject();
  // Compute index into month names. January == month 1 == index 0.
  const monthNum = fn.monthFromDate(xs.date(bucket.lowerBound)) - 1;

  result.name = months[monthNum];
  result.count = cts.frequency(bucket);
  return result;
};

// Generate buckets and counts
import jsearch from '/MarkLogic/jsearch.mjs';
const events = jsearch.collections('events');
events.values(cts.jsonPropertyReference('start', ['type=date']))
  .groupInto(jsearch.makeBuckets({
    min: xs.date('2015-01-01'),
    max: xs.date('2015-12-31'),
    step: xs.yearMonthDuration('P1M')}))
  .map(supplementBucket)
  .result()
```

The output generated is similar to the following:

```
[ {
    "minimum": "2015-02-27",
    "maximum": "2015-02-27",
    "lowerBound": "2015-02-01",
    "upperBound": "2015-03-01",
    "name": "February",
    "count": 1
  }, {
    "minimum": "2015-03-07",
    "maximum": "2015-03-14",
    "lowerBound": "2015-03-01",
```

```
      "upperBound": "2015-04-01",
      "name": "March",
      "count": 2
    }, ...
]
```

Similarly, you can use the `FacetDefinition.groupInto` and `FacetDefinition.map` when generating facets for a document search with `jsearch.facets`. For example, the following query generates facets based on the same set of buckets:

```
import jsearch from '/MarkLogic/jsearch.mjs';
const events = jsearch.collections('events');
events.facets(
  events.facet('events', cts.jsonPropertyReference('start', ['type=date']))
    .groupInto(jsearch.makeBuckets({
      min: xs.date('2015-01-01'),
      max: xs.date('2015-12-31'),
      step: xs.yearMonthDuration('P1M')}))
    .map(supplementBucket),
    events.documents()
  ).result()
```

The output from this query is similar to the following:

```
{"facets": {
  "events": [ {
    "minimum": "2015-02-27",
    "maximum": "2015-02-27",
    "lowerBound": "2015-02-01",
    "upperBound": "2015-03-01",
    "name": "February",
    "count": 1
  }, {
    "minimum": "2015-03-07",
    "maximum": "2015-03-14",
    "lowerBound": "2015-03-01",
    "upperBound": "2015-04-01",
    "name": "March",
    "count": 2
  }, ...
  ]},
  "documents": [ ...]
}
```

For more details on faceting, see “Including Facets in Search Results” on page 308.

7.13.3 Example: Grouping Using Custom Buckets

This example demostrates how to use custom buckets for grouping. The example applies the grouping to facet generation, but you can use the same technique with a values query.

The following code defines custom buckets that group the values of the 'price' JSON property into 3 price range buckets.

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.facets(
  jsearch.facet('Price','price')
    .groupInto([
      jsearch.bucketName('under $10'), 10,
      jsearch.bucketName('$10 to $19.99'), 20,
      jsearch.bucketName('over $20')
    ]))
  .where(cts.directoryQuery('/books/'))
  .result();
```

If the lexicon contains the values `[8, 9, 10, 16, 18, 20, 30]`, then the query results in the following output. (Comments were added for clarity and are not part of the actual output.)

```
{"facets": {
  "price": {
    "under $10": {          // bucket label (for display purposes)
      "value": {
        "minimum": 8,       // min value found in bucket range
        "maximum": 9,       // max value found in bucket range
        "upperBound": 10    // bucket upper bound
      },
      "frequency": 2
    },
    "$10 to $19.99": {
      "value": {
        "minimum": 10,
        "maximum": 18,
        "lowerBound": 10,
        "upperBound": 20
      },
      "frequency": 4
    },
    "over $20": {
      "value": {
        "minimum": 20,
        "maximum": 30,
        "lowerBound": 20
      },
      "frequency": 2
    }
  }
} }
```

The results tell you, for example, that the `price` lexicon contains values under 10, with the maximum value in that range being 9 and the minimum being 8. Similarly, the lexicon contains values greater than or equal to 10, but less than 20. The minimum value found in that range is 10 and the maximum value is 18.

f you use the same grouping specification with `ValuesSearch.groupInto`, you get the same information, but it is arranged slightly differently. For example, the following output was produced using the `values` operation with the same `groupInto` clause.

```
[ {
    "minimum": 8,
    "maximum": 9,
    "upperBound": 10,
    "name": "under $10"
  }, {
    "minimum": 10,             // min value found in bucket range
    "maximum": 18,             // max value found in bucket range
    "lowerBound": 10,          // bucket lower bound
    "upperBound": 20,          // bucket upper bound
    "name": "$10 to $19.99"    // bucket label (for display purposes)
  }, {
    "minimum": 20,
    "maximum": 30,
    "lowerBound": 20,
    "name": "over $20"
} ]
```

If you specify an empty bucket name, a default name is generated from the bucket bounds. For example, the following code applies a similar set of buckets to a values query, using generated bucket names:

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.values('price')
  .where(cts.directoryQuery('/books/'))
  .groupInto([
    jsearch.bucketName(), 10,
    jsearch.bucketName(), 20,
    jsearch.bucketName()
    ])
  .result();
```

This code produces the following output. The bucket min, max, and bounds are the same as before, but the bucket names are the default generated ones:

```
[ {
    "minimum": 8,
    "maximum": 9,
    "upperBound": 10,
    "name": "x < 10"
  }, {
    "minimum": 10,
    "maximum": 19,
    "lowerBound": 10,
    "upperBound": 20,
    "name": "10 <= x < 20"
  }, {
    "minimum": 20,
```

```
    "maximum": 30,
    "lowerBound": 20,
    "name": "20 <= x"
} ]
```

7.14 Preparing to Run the Examples

Use the instructions and scripts in this section to set up your MarkLogic environment to run the examples in this chapter. This includes loading the sample documents and configuring your database to have the required indexes and lexicons.

- Configuring the Database
- Loading the Sample Documents

7.14.1 Configuring the Database

This section guides you through creation of a database configured to run the examples in this chapter. Many examples do not require the indexes, and only the word lexicon query examples require a word lexicon. However, this setup will ensure you have the configuration needed for all the examples.

Running the setup scripts below will do the following. The configuration details are summarized in a table at the end of the section.

- Create a database named jsearch-ex with one forest, named jsearch-ex-1, attached.
- Create element range indexes on the `title`, `author`, `format`, and `price` JSON properties found in the sample documents.
- Create an element word lexicon on the title JSON property found in the sample documents.

The instructions below use Query Console and XQuery to create and configure the database. You do not need to know XQuery to use these instructions. However, if you prefer to do the setup manually using the Admin Interface, see the table at the end of this section for configuration details.

Follow this procedure to create and configure the example database.

1. In your browser, navigate to Query Console and authenticate as a user with Admin privileges. For example, navigate to the following URL is MarkLogic is installed on localhost:

   ```
   http://localhost:8000/qconsole
   ```

2. Use the “+” button to create a new, empty script.

3. Select XQuery in the Query Type dropdown.

4. Paste the following in Query Console as the text of the script just created.

```
xquery version "1.0-ml";

(: Create the database and forest :)
import module namespace admin = "http://marklogic.com/xdmp/admin"
    at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $config :=
  admin:database-create(
    $config, "jsearch-ex",
    xdmp:database("Security"),
    xdmp:database("Schemas"))
let $config :=
  admin:forest-create(
    $config, "jsearch-ex-1",
    xdmp:host(), (), (), ())
return admin:save-configuration($config);

(: Attach the forest to the database :)
import module namespace admin = "http://marklogic.com/xdmp/admin"
    at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $config :=
  admin:database-attach-forest(
    $config, xdmp:database("jsearch-ex"),
    xdmp:forest("jsearch-ex-1"))
return admin:save-configuration($config);
```

5. Click the Run button to execute the script. The database and forest are created.

6. Optionally, confirm creation of the database using the Admin Interface. For example, navigate to the following URL:

```
http://localhost:8001
```

7. In Query Console, click "+" to create another new script. Confirm that the Query Type is still XQuery.

8. Paste the following in Query Console as the text of the script just created. This script will create the indexes and lexicons needed by the examples.

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

let $title-index := admin:database-range-element-index(
```

```
    "string", "", "title", "http://marklogic.com/collation/",
fn:false())
let $author-index := admin:database-range-element-index(
    "string", "", "author", "http://marklogic.com/collation/",
fn:false())
let $format-index := admin:database-range-element-index(
    "string", "", "format", "http://marklogic.com/collation/",
fn:false())
let $price-index := admin:database-range-element-index(
    "float", "", "price", "", fn:false())
let $config := admin:get-configuration()
let $config :=
  admin:database-add-range-element-index(
    $config, xdmp:database("jsearch-ex"),
    ($title-index, $author-index, $format-index, $price-index))
return admin:save-configuration($config);

import module namespace admin = "http://marklogic.com/xdmp/admin"
    at "/MarkLogic/admin.xqy";

let $title-lexicon := admin:database-element-word-lexicon(
    "", "title", "http://marklogic.com/collation/")
let $config := admin:get-configuration()
let $config :=
  admin:database-add-element-word-lexicon(
    $config, xdmp:database("jsearch-ex"),
    ($title-lexicon))
return admin:save-configuration($config);
```

9. Click the Run button. The range indexes and word lexicon are created.

You should now proceed to “Loading the Sample Documents” on page 379.

If you choose to create the example environment manually with the Admin Interface, use the configuration summary below.

Resource	Configuration	
	Setting	Value
Forest	name	jsearch-ex-1
Database	name	jsearch-ex

Resource	Configuration	
	Setting	Value
`title` element range index	type	`string`
	namespace URI	none
	localname	`title`
	collation	`http://marklogic.com/collation/`
	range value positions	`false`
`author` element range index	type	`string`
	namespace URI	none
	localname	`author`
	collation	`http://marklogic.com/collation/`
	range value positions	`false`
`format` element range index	type	`string`
	namespace URI	none
	localname	`format`
	collation	`http://marklogic.com/collation/`
	range value positions	`false`
`price` element range index	type	`float`
	namespace URI	none
	localname	`price`
	range value positions	`false`
`title` element wod lexicon	namespace URI	none
	localname	`title`
	collation	`http://marklogic.com/collation/`

7.14.2 Loading the Sample Documents

After you create and configure the sample database, follow the instructions in this section to load the sample documents.

1. In your browser, navigate to Query Console and authenticate as a user with write privileges for the `jsearch-ex` database. For example, navigate to the following URL is MarkLogic is installed on localhost:

   ```
   http://localhost:8000/qconsole
   ```

2. Use the "+" button to create a new, empty script.

3. Select `JavaScript` in the Query Type dropdown.

4. Select `jsearch-ex` in the Content Source dropdown.

   You will not see it if you have just finished creating and configuring the database and are still using the same Query Console session. If this happen, reload Query Console in your browser to refresh the Content Source list.

5. Paste the following in Query Console as the text of the script just created.

```
const directory = '/books/';
const books = [
  {uri: 'frost1.json',
   data: { title: 'Collected Works', author: 'Robert Frost',
          edition: {format: 'paperback', price: 30 },
          synopsis: 'The complete works of the American Poet Robert
Frost.'
         }},
  {uri: 'twain1.json',
   data: { title: 'Adventures of Tom Sawyer', author: 'Mark Twain',
          edition: {format: 'paperback', price: 9 },
          synopsis: 'Tales of mischief and adventure along the
Mississippi River with Tom Sawyer, Huck Finn, and Becky Thatcher.'
         }},
  {uri: 'twain2.json',
   data: { title: 'Adventures of Tom Sawyer', author: 'Mark Twain',
          edition: {format: 'hardback', price: 18 },
          synopsis: 'Tales of mischief and adventure along the
Mississippi River with Tom Sawyer, Huck Finn, and Becky Thatcher.'
         }},
  {uri: 'twain3.json',
   data: { title: 'Adventures of Huckleberry Finn', author: 'Mark
Twain',
          edition: {format: 'paperback', price: 8 },
             synopsis: 'The adventures of Huck, a boy of 13, and Jim,
an escaped slave, rafting down the Mississippi River in pre-Civil War
America.'
         }},
```

```
  {uri: 'twain4.json',
   data: { title: 'Adventures of Huckleberry Finn', author: 'Mark
Twain',
           edition: {format: 'hardback', price: 18 },
              synopsis: 'The adventures of Huck, a boy of 13, and Jim,
an escaped slave, rafting down the Mississippi River in pre-Civil War
America.'
         }},
  {uri: 'steinbeck1.json',
   data: { title: 'The Grapes of Wrath', author: 'John Steinbeck',
           edition: {format: 'paperback', price: 10 },
           synopsis: 'Chronicles the 1930s Dust Bowl migration of one
Oklahoma farm family, from their homestead and forced to the promised
land of California.'
  }},
  {uri: 'steinbeck2.json',
   data: { title: 'Of Mice and Men', author: 'John Steinbeck',
           edition: {format: 'hardback', price: 20 },
            synopsis: 'A tale of an unlikely pair of drifters who move
from job to job as farm laborers in California, until it all goes
horribly awry.'
  }},
  {uri: 'steinbeck3.json',
   data: { title: 'East of Eden', author: 'John Steinbeck',
           edition: {format: 'audiobook', price: 16 },
           synopsis: 'Follows the intertwined destinies of two
California families whose generations reenact the fall of Adam and Eve
and the rivalry of Cain and Abel.'
  }}
];

books.forEach( function(book) {
  xdmp.eval(
    'declareUpdate(); xdmp.documentInsert(uri, data,
xdmp.defaultPermissions(), ["classics"]);',
    {uri: directory + book.uri, data: book.data}
  );
});
```

6. Click the Run button to execute script. The sample documents are inserted into the database.

7. Optionally, click the Explore button to examine the database contents. You should see 8 JSON documents with URIs such as “/books/frost1.json”.

The jsearch-ex database is now fully configured to support all the samples in this chapter in Query Console. When running the examples, set the Content Source to `jsearch-ex` and the Query Type to `JavaScript`.

8.0 Search Customization Using Query Options

When you use the XQuery Search API, REST API, or Java API, you can customize and control your searches using query options. This chapter highlights key query option features. The following topics are covered:

- Introduction
- Getting the Default Query Options
- Checking Query Options for Errors
- Constraint Options
- Operator Options
- Return Options
- Searchable Expression Option
- Fragment Scope Option
- Searching Key-Value Metadata Fields
- Modifying Your Snippet Results
- Extracting a Portion of Matching Documents
- Customizing Search Results with a Decorator
- Other Search Options
- Query Options Examples

For details on the syntax of each option, see “Appendix: Query Options Reference” on page 816.

8.1 Introduction

Query options enable you to control many aspects of content and values searches, including limiting the scope of a search, customizing the string search grammar, defining sort order, and specifying the contents and format of search results.

MarkLogic Server defines a set of default query options that are applied when you do not include custom query options in a search. You can modify the default query options if you are using the REST or Java API. You can override the default options by defining custom query options and apply them to individual searches.

Query options can be specified in XML for all APIs. The REST and Java APIs also support a JSON representation. XML query options are always expressed as a `search:options` element in the following namespace:

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

Most search operations in the XQuery, REST and Java APIs accept optional query options, including the following:

- XQuery: The functions `search:search`, `search:resolve`, `search:values`, and `search:parse`
- REST: The `/search`, `/values`, `/qbe` services
- Java: Searches performed using the `com.marklogic.client.query.QueryManager` class.

8.2 Getting the Default Query Options

If you do not include any query options in a search operation that accepts them, the default query options are used. You can retrieve the default query options definition using the XQuery or REST APIs.

To retrieve the default query options using XQuery, call `search:get-default-options`.

To retrieve the default query options using REST, make a GET request to `/config/query/default`. For details, see the REST Client API Reference.

8.3 Checking Query Options for Errors

Query options can be fairly complex. The XQuery, REST and Java APIs include mechanisms for checking your query options for errors.

In XQuery, use the function `search:check-options`. This function validates your options and reports any errors it finds. It returns empty if the options are valid. If it finds errors, they are returned in the form of one or more `search:report` nodes.

The REST API can perform an equivalent check when you persist query options through the `/config/query` service. The Java API performs this check when you call `com.marklogic.client.admin.QueryOptionsManager.writeOptions`.

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

A common MarkLogic XQuery design pattern is to add a `$debug` option to your code that defaults to `false`, and when `true`, runs `search:check-options` or adds the `debug` option your query options. Set the `$debug` variable to true for development and false for production.

8.4 Constraint Options

A *constraint* is a mechanism the Search API and Client APIs use to limit the scope of a search. For example, find a term only when it occurs in the value of a particular XML element or JSON property.

Constraints are designed to take advantage of range indexes, 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 a string query. For information on search parsing and grammar, see "The Default String Query Grammar" on page 68.
- 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 34.
- To enhance search suggestions. For example, when using `search:suggest`. For information on search suggestions, see "Search Term Completion" on page 36.

A constraint must have a name, and that name must be unique across all operators and constraints in your query options. The name may not contain whitespace.

You can use a constraint name in a string query with operators such as ":", "<", and ">=". For example, if you define a range constraint named "price" that limits the scope of the search to an XML element or JSON property named "price", then the following query text matches occurrences where the value in that element or property is less than 10:

```
price < 10
```

For more details about the search grammar, see "Automatic Query Text Parsing and Grammar" on page 32 and "The Default String Query Grammar" on page 68. For more details on defining constraints, see "constraint" on page 822.

Similarly, if your "price" constraint defines value range buckets with the names "under10", "10to20", and "20+", then the following query matches occurrences where the value of the price element or property is in the range of the "under10" bucket:

```
price:under10
```

For more details on bucketed constraints, see "Bucketed Range Constraint Example" on page 391.

The following table lists the types of constraints you can build with query options. For more details, see “constraint” on page 822 in the options reference.

<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 389 and “value” on page 831.</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 389 and “word” on page 835 in the options reference.</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 390 and “collection” on page 839 in the options reference.</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 391, “Buckets Example” on page 62, “Computed Buckets Example” on page 64. 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">container</td><td>Restricts qtext to a particular XML element or JSON property. Requires position indexes enabled on the database for the best performancce.</td><td>cts:element-query</td><td>No facets for container constraints.</td></tr>
<tr><td colspan="3">Example element-query constraint:

<pre>
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="sample-element-constraint">
    <container>
      <element name="title" ns="http://my/namespace" />
    </container>
  </constraint>
</options>
</pre></td></tr>
<tr><td rowspan="2">properties</td><td>Finds matches on the corresponding properties documents.</td><td>cts:properties-fragment-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
geo-path
geo-json-propty
geo-json-property-pair</td><td>These geospatial constraints find matches on geospatial data. To use as a facet, the <constraint> element requires a <heatmap> child; for details, see “Geospatial Constraint Example” on page 393.</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 42.</td><td>Depends on your custom code implementation</td><td>Depends on your custom code implementation</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.

WhenMarkLogic Server parses a constraint in a query (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 query options and the value is a single search term, then the joiner string is treated as part of the search term. 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 in your query options 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
- Geospatial Constraint Example

For an example of a custom constraint, see “Creating a Custom Constraint” on page 42.

8.4.1 Value Constraint Example

The following query options define 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 use string queries such as the following to use these constraints:

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

Both parts of the above queries 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>
```

For more details, see “value” on page 831 in the options reference.

8.4.2 Word Constraint Examples

The following query options define two word constraints: One for the `<name/>` element in the namespace `http://authors-r-us.com` and one for the field `my-field`. 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>
```

You can create string and structured queries that use these constraints. For example, the following string query uses the `name` constraint.

```
name:raymond
```

When parsed, it becomes a `cts:element-word-query`:

```
<cts:element-word-query>
  <cts:element xmlns:_1="http://authors-r-us.com">
    _1:name
  </cts:element>
  <cts:text>raymond</cts:text>
</cts:element-word-query>
```

This query matches 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>
```

Similarly, the following string query using the `description` constraint parses into a `cts:field-word-query`:

```
description:author
```

This query matches the above document if the `name` element is included in the definition of the field named `my-field`. For details on fields, see Fields Database Settings in the *Administrator's Guide*.

For more details, see "word" on page 835 in the query options reference.

Word constraints can also be used in structured queries. For example, the following structured query is equivalent to the `name:raymond` string query:

```
<search:query xmlns:search="http://marklogic.com/appservices/search">
  <search:word-constraint-query>
    <search:constraint-name>name</search:constraint-name>
    <search:text>raymond</search:text>
  </search:word-constraint-query>
</search:query>
```

For details, see "Searching Using Structured Queries" on page 74.

8.4.3 Collection Constraint Example

The following query options define a collection constraint, which allows you to constrain your search to documents that are in a specified collection.

Note: You must enable the collection lexicon in the database to use collection constraints. If the collection lexicon is not enabled, an exception is thrown when you use a query with a collection constraint.

If you include a `prefix` attribute in the definition of a collection 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 693.

You can also use collection constraints in a structured query, using `collection-constraint-query`. For details, see “Searching Using Structured Queries” on page 74.

8.4.4 Bucketed Range Constraint Example

Range constraints operate on typed element, element attribute, JSON property, field, or path values that have a corresponding range index in the database. Without the correct range index, queries using range constraints throw a runtime exception.

Range constraints can match on either all of the individual values in the constrained scope (element, property, field, etc.), or on ranges of values defined as “buckets”. You can define two types of buckets in a `range` constraint specification.

- Use the bucket portion of a range constraint to define a value range in terms of fixed values.

- Use the computed-bucket portion of a range constraint to define a value range in terms of a dynamic time that is computed at runtime. For more information about `computed-bucket` range constraints, see "Computed Buckets Example" on page 64.

The following example uses `search:parse` with query options that define a `bucket` range constraint. The constraint definition is based on the JSON data from "Preparing to Run the Examples" on page 375, but you can create a similar constraint on XML data.

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

search:parse('price:under10',
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="price">
    <range type="xs:float" facet="true">
      <bucket lt="10" name="under10">under $10</bucket>
      <bucket ge="10" lt="21" name="10to20">$10 to $20</bucket>
      <bucket ge="21" name="20+">$20+</bucket>
      <facet-option>limit=10</facet-option>
      <json-property>/edition/price</json-property>
    </range>
  </constraint>
</options>)
```

With the given constraint definition, the query text "price:under10" parses into the following cts:query:

```
<cts:and-query xmlns:cts="http://marklogic.com/cts">
  <cts:json-property-range-query operator="&gt;=">
    <cts:property>/edition/price</cts:property>
    <cts:value xsi:type="xs:float"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      10
    </cts:value>
  </cts:json-property-range-query>
  <cts:json-property-range-query operator="&lt;">
    <cts:property>/edition/price</cts:property>
    <cts:value xsi:type="xs:float"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      21
    </cts:value>
  </cts:json-property-range-query>
</cts:and-query>
```

For other range constraint examples, see "Buckets Example" on page 62 and "Computed Buckets Example" on page 64, and the following example.

For syntax details, see "bucket" on page 880 of the query options appendix.

8.4.5 Exact Match (Unbucketed) Range Constraint Example

The following example shows an exact match year range constraint. It returns results that match the `xs:gYear` value "1964" when it is the value of an XML element named "nominee", in the namespace "http://marklogic.com/wikipedia". The database configuration must include a range index matching the constraint definition.

```
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)
```

You can also try out these query options with the Java and REST APIs. For details, see Apply Dynamic Query Options to Document Searches in the *Java Application Developer's Guide* or Specifying Dynamic Query Options with Combined Query in the *REST Application Developer's Guide*.

8.4.6 Geospatial Constraint Example

The following example shows how to use a geospatial constraint to generate geospatial facets in the form of boxes. For details on these options, see "Appendix: Query Options Reference" on page 816. For details on the concept of geospatial facets, see "Creating Geospatial Facets" on page 525.

Suppose the database contains documents of the following form, describing earthquake events:

```
<event xmlns="http://quakeml.org/xmlns/bed/1.2">
  <time>2015-03-24T09:39:15.500Z</time>
  <latitude>36.5305</latitude>
  <longitude>-98.8456</longitude>
  <depth>8.72</depth>
  <mag>3.4</mag>
  <magType>mb_lg</magType>
  <id>us10001q5d</id>
  <place>33km ENE of Mooreland, Oklahoma</place>
  <type>earthquake</type>
</event>
```

If you define a geospatial element pair range index on the `/event/latitude` and `/event/longitude` elements, then you can use query options to define an associated constraint that can be used to generate geospatial facets by including a `heatmap` element in the constraint definition.

The `heatmap` element defines a region over which to generate facets, along with the number of latitude and longitude divisions to use in sub-divisions the region into boxes. When you perform a search with the constraint in scope, the search response includes a set of `search:box` elements that give you the frequency of matches in each sub-division. You can use this box data for faceting or heatmap generation.

For example, the following constraint includes a heat map that corresponds very roughly to the continental United States, and divides the region into a set of 20 boxes (5 latitude divisions and 4 longitude divisions):

```
<constraint name="qgeo">
  <geo-elem-pair facet="true">
    <parent ns="http://quakeml.org/xmlns/bed/1.2" name="event"/>
    <lat ns="http://quakeml.org/xmlns/bed/1.2" name="latitude"/>
    <lon ns="http://quakeml.org/xmlns/bed/1.2" name="longitude"/>
    <heatmap s="24.0" n="49.0" e="-67.0" w="-125.0"
             latdivs="5" londivs="4" />
    <facet-option>gridded</facet-option>
  </geo-elem-pair>
</constraint>
```

If you also define an element range index on `/event/mag`, then the following search finds earthquakes with a magnitude (`mag`) greater than 4.0 within the continental US.

```
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="qgeo">
    <geo-elem-pair>
      <parent ns="http://quakeml.org/xmlns/bed/1.2" name="event"/>
      <lat ns="http://quakeml.org/xmlns/bed/1.2" name="latitude"/>
      <lon ns="http://quakeml.org/xmlns/bed/1.2" name="longitude"/>
      <heatmap s="24.0" n="49.0" e="-67.0" w="-125.0"
               latdivs="5" londivs="4" />
      <facet-option>gridded</facet-option>
    </geo-elem-pair>
  </constraint>
  <constraint name="mag">
    <range type="xs:float" facet="false">
      <element ns="http://quakeml.org/xmlns/bed/1.2" name="mag"/>
    </range>
  </constraint>
  <return-facets>true</return-facets>
```

```
</options>
return search:search("mag GT 4", $options)
```

The search response includes the following frequency count per geographic region, based on the geo constraint in the options:

```
<search:boxes name="qgeo">
  <search:box count="3" s="-18.9346" w="-178.4936"
              n="-17.8151" e="-174.9279"/>
  <search:box count="2" s="-48.36" w="-87.3529"
              n="7.6115" e="-81.8738"/>
  <search:box count="8" s="-20.7243" w="-73.0949"
              n="6.8852" e="151.9563"/>
  <search:box count="2" s="36.2665" w="70.65"
              n="36.4086" e="140.0085"/>
  <search:box count="3" s="53.6477" w="160.0923"
              n="53.8233" e="161.7353"/>
</search:boxes>
```

By default, the returned boxes are the smallest box within each grid box that encompasses all the matched documents. To return boxes corresponding to the grid divisions instead, add `<facet-option>gridded</facet-option>` to the geo constraint definition. In this example it results in the following boxes:

```
<search:boxes name="qgeo">
  <search:box count="3" s="-90" w="-180" n="24" e="-125"/>
  <search:box count="2" s="-90" w="-96" n="24" e="-81.5"/>
  <search:box count="8" s="-90" w="-81.5" n="24" e="180"/>
  <search:box count="2" s="34" w="-81.5" n="39" e="180"/>
  <search:box count="3" s="44" w="-81.5" n="90" e="180"/>
</search:boxes>
```

For an example of expression a geospatial constraint in JSON, see “geo-attr-pair” on page 847 and “heatmap” on page 877.

Note that the generated facets will bucket all matched points, even if they’re outside the extent of the box defined in the heatmap. If you want to facet only within bounds of the heatmap, then your search results must be similarly constrained.

For example, you could add an `additional-query` option that constrains the search to the same box as defined by the heatmap using a `cts:element-pair-geospatial-query`.

8.5 Operator Options

Search operators enable you to define operators in your string query grammar that 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 a string query or query text. For details, see “operator” on page 908 in the query options appendix.

For example, the following options XML defines an operator named `sort` that enables you to sort by relevance or by date:

```
<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 definition in the query options allows you to add text like the following to a string query, and MarkLogic Server will parse the string and sort it according to the operator specification.

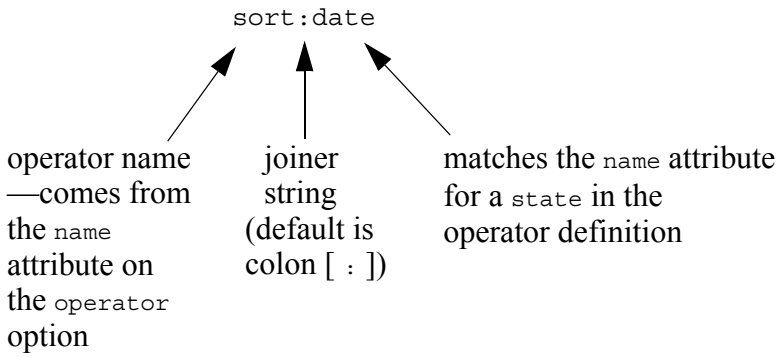
```
sort:date

sort:relevance
```

Each operator is named, and the name must be unique across all operators and constraints in your query options. When you specify an operator in a string query, 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`. You can also include operator settings in structured queries, using the `operator-state` element. For details, see “operator-state” on page 192.

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 32 and "The Default String Query Grammar" on page 68.

The `search:state` 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`
- `search-option`
- `searchable-expression`

  Note: Due to security and performance considerations, beginning in MarkLogic 9.0-10, the `searchable-expression` property/element in query options is deprecated. Please see Search API searchable-expression Deprecated in the Release Notes for more information.

- `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.

8.6 Return Options

You can specify a number of options that control what is include in a search response, such as the results returned by the XQuery Search API function `search:search`, a GET request to the `/search` service of the REST API, or the Java API method `com.marklogic.client.query.QueryManager.search`. These include the following boolean options:

- `return-aggregates`
- `return-constraints`
- `return-facets`
- `return-frequencies`
- `return-metrics`
- `return-plan`
- `return-qtext`
- `return-query`
- `return-results`
- `return-similar`
- `return-values`

For details on these and other options, see "Appendix: Query Options Reference" on page 816.

Setting one of these options to `true` includes the specified information in the `search:response` returned by a search. Setting to `false` omits the information 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*.

8.7 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*.

Note: Due to security and performance considerations, beginning in MarkLogic 9.0-10, the `searchable-expression` property/element in query options is deprecated. Please see Search API searchable-expression Deprecated in the Release Notes for more information.

By default, searches apply to 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.

For more details, see “searchable-expression” on page 921 in the query options appendix.

8.8 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 `<term>` or 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>)
```

Setting fragment scope in a `<term>` definition causes term queries to be evaluated in the specified scope. However, the local fragment scope is ignored under the following conditions:

- The `<term>` definition includes an inline word, value, or range constraint or a reference to another constraint. In this case, the fragment scope of the constraint definition applies.
- The `<term>` definition specifies a custom term processing function using `apply`/`ns`/`at`. In this case, the custom function controls scope.

For more details, see “fragment-scope” on page 899 in the query options appendix.

8.9 Searching Key-Value Metadata Fields

You can associate key-value metadata with a document using the “metadata” option during document insertion, or using builtin functions such as the `xdmp:document-set-metadata` XQuer function or the `xdmp.documentSetMetadata` Server-Side JavaScript function.

To make key-value metadata searchable, you must define a metadata field on the key, as described in Configuring a New Metadata Field in the *Administrator's Guide*. You might also need to enable field value searches on your database or configure a field range index, depending on the type of query you want to perform.

Once you define a field over a metadata key, you can include that key-value pair in searches using any of the field query capabilities.

The following example defines a constraint named “by” over a metadata field named “author”, and then uses the constraint in a string query. The search only matches occurrences of “twain” in the metadata key-value pair with the key “author”. It will not match occurrences in document content or under a different key.

```
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="by">
    <word>
      <field name="author"/>
    </word>
```

```
  </constraint>
</options>
return search:search("by:twain", $options)
```

For more details, see Metadata Fields in the *Administrator's Guide*.

8.10 Modifying Your Snippet Results

The `transform-results` option enables 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

8.10.1 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-matches/>
</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. For more details, see "transform-results" on page 936 in the query options appendix.

`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-matches`	Specify zero or more XML elements or JSON properties 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-matches>`
	`  <element ns="" name="TITLE"/>`
	`  </preferred-matches>`
	`</transform-results>`
	For JSON properties, use a `json-property` child element or property. For example:
	`<transform-results apply="snippet">`
	`  <preferred-matches>`
	`  <json-property>title</json-property>`
	`  </preferred-matches>`
	`</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 snippetting 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"` snippeting option is as follows:

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

The `apply="metadata-snippet"` option returns the specified preferred elements from the properties documents. If no `<preferred-matches>` 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.

8.10.2 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 36. 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>
```

You must generate an XML `<search:snippet/>` element, even when producing snippets for JSON documents. You can embed JSON in your generated snippet as text. If you include a `format="json"` attribute in your snippet, the REST, Java, and Node.js client APIs will treat the embedded text as JSON and unquote when returning search results as JSON. For example:

```
declare function my:snippeter(
   $result as node(),
   $ctsquery as schema-element(cts:query),
   $options as element(search:transform-results)?
) as element(search:snippet) {
    element search:snippet {
        attribute format { "json" },
        text {'{"MY":"CUSTOM SNIPPET"}'}
    }
};
```

You can optionally pass additional information into your custom snippeting function by adding extra children to the `transform-results` option. The Search API passes 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.

8.11 Extracting a Portion of Matching Documents

This sections explains how to use the `extract-document-data` option to project selected XML elements, XML attributes, and JSON properties out of documents matched by a search. For more details, see “extract-document-data” on page 895.

By default, a search returns only the `search:response` result summary. When you use `extract-document-data`, you can embed selected portions of each matching document in the search results or return the selected portions as documents instead in a `search:response`.

The projected contents are specified through absolute XPath expressions in `extract-document-data` and a `selected` attribute that specifies how to treat the selected content.

For example, suppose your database includes the following documents:

XML	JSON
URI: /extract/doc1.xml <root> <a>foo <body> <target>content</target> </body> bar </root>	URI: /extract/doc2.json {"root": { "a": "foo", "body": { "target":"content" }, "b": "bar" }}

Then, if you search for “content” both of the above documents match.

```
search:search("content")
```

If you add the following `extract-document-data` option, the search response includes the projected content in each search result, similar to the way sinippets are returned. Each projection contains only the `target` element or property specified by the option.

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

search:search("content",
  <options xmlns="http://marklogic.com/appservices/search">
    <extract-document-data>
      <extract-path>/root/body/target</extract-path>
    </extract-document-data>
    <search-option>filtered</search-option>
  </options>
)

==>

<search:response snippet-format="snippet" total="2" start="1" ...>
  <search:result index="1" uri="/extract/doc1.xml"
       path="fn:doc(&amp;quot;/extract/doc1.xml&amp;quot;)" ...>
    <search:snippet>...</search:snippet>
    <search:extracted context="fn:doc(&amp;quot;/extract/doc1.xml&amp;quot;)">
      <target>content</target>
    </search:extracted>
  </search:result>
  <search:result index="2" uri="/extract/doc2.json"
       path="fn:doc(&amp;quot;/extract/doc2.json&amp;quot;)" ...>
    <search:snippet>...</search:snippet>
```

```
    <search:extracted format="json" kind="object"
        context="fn:doc("/extract/doc2.json")">
      {"target":"content"}
    </search:extracted>
  </search:result>
  ...
</search:response>
```

Using `extract-document-data` with `search:resolve` has a similar effect. However, if you use the option with `search:resolve-nodes`, you get the projected content as sparse documents instead of a `search:response`. For example:

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

search:resolve-nodes(
  search:parse("content"),
  <options xmlns="http://marklogic.com/appservices/search">
    <extract-document-data>
      <extract-path>/root/body/target</extract-path>
    </extract-document-data>
    <search-option>filtered</search-option>
  </options>
)

==>

(
  <?xml version="1.0" encoding="UTF-8"?>
  <search:extracted context="fn:doc(&amp;quot;/extract/doc1.xml&amp;quot;)"
      xmlns:search="http://marklogic.com/appservices/search">
    <target>content</target>
  </search:extracted>,

  {"context":"fn:doc(\"/extract/doc2.json\")",
   "extracted":[{"target":"content"}]}
)
```

You can specify multiple `extract-path` elements. For paths to XML elements and attributes, namespaces in scope on the `search:options` (or `search:search` for a combined query) can be used for namespace prefix bindings on the path expressions.

For performance and security reasons, the path expression in `extract-path` is limited to a subset of XPath. For details, see The extract-document-data Query Option in the *XQuery and XSLT Reference Guide*.

Use the `selected` attribute to specify how to use the content selected by `extract-path` in the returned documents. You can set the attribute to `include` (default), `include-with-ancestors`, `exclude`, or `all`. If you choose anything except `include`, the output for each match is a sparse representation of the original document instead of a document with an `extracted` element or JSON property; see the examples in the table below.

The table below demonstrates how `extract-document-data/@selected` affects the content extracted from the two sample documents.

selected	XML	JSON
include	`<target>content</target>`	`{"target":"content"}`

include-with-ancestors	`<root>`
	`  <body>`
	`  <target>content</target>`
	`  </body>`
	`</root>`
		`{"root":{`
		`  "body":{`
		`  "target":"content"`
		`  }`
		`}}`

exclude	`<root>`
	`  <a>foo</a>`
	`  <body/>`
	`  <b>bar</b>`
	`</root>`
		`{"root":{`
		`  "a":"foo",`
		`  "body":{},`
		`  "b":"bar"`
		`}}`

all	`<root>`
	`  <a>foo</a>`
	`  <body>`
	`  <target>content</target>`
	`  </body>`
	`  <b>bar</b>`
	`</root>`
		`{"root":{`
		`  "a":"foo",`
		`  "body":{`
		`  "target":"content"`
		`  },`
		`  "b":"bar"`
		`}}`

When `selected` is `include` or `include-with-ancestors` and no content for a given search result is matched by the `extract-path` expressions, an `extracted-none` placeholder is returned to preserve the presence of the document in the result list. For example:

```
<search:extracted-none
  context="fn:doc("/extract/doc1.xml&")" ... />

{ "context":"fn:doc(\"/extract/doc2.json\")",
  "extracted-none":null}
```

The Node.js Client API supports `extract-document-data` through the `queryBuilder.extract` method. By default, `DatabaseClient.documents.query` is a multi-document read, so it returns the extracted content as individual documents. You can request a search result summary instead, including the extracted content by using `queryBuilder.withOptions({categories: 'none'})`. For details, see Extracting a Portion of Each Matching Document in the *Node.js Application Developer's Guide*.

The Java Client API supports `extract-document-data` via the `QueryManager.search` and `DocumentManager.search` interfaces. For details on embedding extracted content in the search results, see Extracting a Portion of Matching Documents in the *Java Application Developer's Guide*. For details on retrieving extracted content in document form, see Extracting a Portion of Each Matching Document in the *Java Application Developer's Guide*.

When you use `extract-document-data` with the REST Client API `/v1/search` service, whether the extracted content is returned in the search response or as separate documents depends on the Accept header. A multi-document read returns the extracted content as documents. A simple search returns the extracted content in the search response. For details, see Extracting a Portion of Each Matching Document in the *REST Application Developer's Guide*.

8.12 Customizing Search Results with a Decorator

This section describes how to implement a custom search result *decorator* function to add extra information to the search results for your application. The following topics are covered:

- Understanding Search Result Decorators
- Writing a Custom Search Result Decorator
- Installing a Custom Search Result Decorator
- Using a Custom Search Result Decorator

8.12.1 Understanding Search Result Decorators

When you perform a query, MarkLogic Server returns a `<search:response/>` containing a `<search:result>` for each document or fragment that satisfies your query. You can use a search result decorator to add additional information to the each `<search:result/>`, without changing the basic structure or default contents.

For example, the MarkLogic REST API uses an internal result decorator to add `href`, `mimetype`, and `format` attributes to search results. The following output shows the extended information added by the REST API default decorator:

```
<search:response snippet-format="snippet" total="1" ...>
  <search:result ...
      href="/v1/documents?uri=/docs/example.xml"
      mimetype="text/xml" format="xml">
  ...
```

```
  </search:result>
</search:response>
```

Applications using the XQuery Search API can use custom result decorators to similarly add attributes and elements to a `<search:result/>`.

> **Note:** Users of the REST API, Java, or Node.js Client API should use search result transformations to modify search results, rather than result decorators. It is possible to override the builtin REST API result decorator, but it is not recommended. If you use a custom result decorator in a Client API context, it completely replaces the default decorator that adds `href`, `mimetype`, and `format` data to results. Also, any data added by a decorator is returned as serialized XML, even when the client requests results in JSON.

To create and use a search result decorator, do the following:

1. Write an XQuery function that conforms to the decorator interface.
2. Install your function in the modules database or modules directory associated with your App Server.
3. Instruct MarkLogic Server to use your function by specifying it in a `result-decorator` query option that you supply with your search.

The rest of this section covers these steps in detail.

8.12.2 Writing a Custom Search Result Decorator

To create a custom decorator, implement an XQuery function that conforms to the following interface:

```
declare function your-name($uri as xs:string) as node()*
```

The `$uri` input parameter is the URI of a document containing search matches. The nodes returned by your function become attributes or child nodes of the `search:result` element on whose behalf it is called.

Your result decorator should always produce XML, even when working with JSON documents or producing output for a client expecting JSON search results.

The following example is a custom decorator function that returns the same information as the default REST API decorator (`href`, `mimetype`, and `format`), with the attribute names changed so you can see them in use. The example also adds a `<my-elem/>` element to the search results.

```
xquery version "1.0-ml";

module namespace my-lib = "http://marklogic.com/example/my-lib";
```

```
declare function my-lib:decorator($uri as xs:string) as node()*
{
  let $format   := xdmp:uri-format($uri)
  let $mimetype := xdmp:uri-content-type($uri)
  return (
    attribute my-href { concat("/documents/are/here?uri=", $uri) },

    if (empty($mimetype)) then ()
    else attribute my-mimetype {$mimetype},

    if (empty($format)) then ()
    else attribute my-format { $format }

    element my-elem { "Extra Goodness" }
  )
};
```

To use your function, install it as a module in your App Server, and then specify it in a `result-decorator` query option. For details, see "Installing a Custom Search Result Decorator" on page 410 and "Using a Custom Search Result Decorator" on page 410.

8.12.3 Installing a Custom Search Result Decorator

Install the XQuery library module containing your decorator function in the modules database or under the filesystem root associated with your App Server.

For example, running the following query in Query Console loads an XQuery module into the modules database with the URI `/my.domain/decorator.xqy`, assuming you select the modules database as the Content Source in Query Console:

```
xquery version "1.0-ml";
xdmp:document-load(
  "/space/rest/decorator.xqy",
  <options xmlns="xdmp:document-load">
    <uri>/my.domain/decorator.xqy</uri>
  </options>)
```

If you use the REST API or Java API, install the module in the modules database associated with your REST API instace.

8.12.4 Using a Custom Search Result Decorator

To use a custom search result decorator, specify it in a `result-decorator` query option that is included with your query.

For example, if you install the custom decorator from "Writing a Custom Search Result Decorator" on page 409 as `/my.domain/decorator.xqy`, then you can reference it in query options as follows:

```
<options xmlns="http://marklogic.com/appservices/search">
  <result-decorator apply="decorator"
      ns="http://marklogic.com/example/my-lib"
      at="/my.domain/decorator.xqy"/>
</options>
```

The following example uses the above query options in a search:

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

search:search(
  "cat AND dog",
  <options xmlns="http://marklogic.com/appservices/search">
    <result-decorator apply="decorator"
        ns="http://marklogic.com/example/my-lib"
        at="/my.domain/decorator.xqy" />
  </options>
)
```

The decorator adds the `my-href`, `my-mimetype`, and `my-format` attributes and the `my-elem` element to the search results, similar to the following:

```
<search:response ...>
  <search:result index="1" uri="/docs/example.xml"
       path="fn:doc(&amp;quot;/docs/example.xml&amp;quot;)" ...
       my-href="/documents/are/here?uri=/docs/example.xml"
       my-mimetype="text/xml" my-format="xml">
    <my-elem>Extra Goodness!</my-elem>
    ...
  </search:result>
</search:response>
```

8.13 Other Search Options

There are many 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 a complete list, see "Appendix: Query Options Reference" on page 816.

8.14 Query Options Examples

This section includes the following additional query options examples:

- Example: Values and Tuples Query Options
- Example: Field Constraint Query Options
- Example: Collection Constraint Query Options
- Example: Path Range Index Constraint Query Options
- Example: Element Attribute Range Constraint Query Options
- Example: Geospatial Constraint Query Options

8.14.1 Example: Values and Tuples Query Options

This examples demonstrates how to create and use a values or tuples query option. For more details, see “values” on page 945 and “tuples” on page 941 in the query options appendix.

This example sets up the following configurations:

- A values tuple `/v1/values/pop`, which gets the values from the element range index from `xs:QName("popularity")` (in no namespace), and is typed as an `xs:int`.
- A values tuple /`v1/values/score`, which gets the values from the element range index from a QName with namespace `"http://test.aggr.com"` and local name `"score"`, typed as an `xs:decimal`.
- A tuple `/v1/values/pop-rate-tups`, which combines the range index from from `xs:QName("popularity")` (in no namespace), and typed as an `xs:int`, and the range index from a QName with namespace `"http://test.aggr.com"` and local name `"score"`, typed as an `xs:decimal`.

Format	Options
XML	<search:options xmlns:search="http://marklogic.com/appservices/search"> <search:values name="pop-aggr"> <search:range type="xs:int"> <search:element ns="" name="popularity"/> </search:range> </search:values> <search:values name="score-aggr"> <search:range type="xs:decimal"> <search:element ns="http://test.aggr.com" name="score"/> </search:range> </search:values> <search:tuples name="pop-rate-tups"> <search:range type="xs:int"> <search:element ns="" name="popularity"/> </search:range> <search:range type="xs:int"> <search:element ns="http://test.tups.com" name="rate"/> </search:range> </search:tuples> </search:options>

Format	Options

JSON	<pre>{ "options": {
	    "values": [{
	      "name": "pop-aggr",
	      "range": {
	        "type": "xs:int",
	        "element": {
	           "ns": "",
	           "name": "popularity"
	        }
	      }
	    },
	    {
	      "name": "score-aggr",
	      "range": {
	        "type": "xs:decimal",
	        "element": {
	          "ns": "http://test.aggr.com",
	          "name": "score"
	        }
	      }
	    }],
	    "tuples": [{
	      "name": "pop-rate-tups",
	      "range": [
	         {
	           "type": "xs:int",
	           "element": {
	             "ns": "",
	             "name": "popularity"
	           }
	         },
	         {
	           "type": "xs:int",
	           "element": {
	             "ns": "http://test.tups.com",
	             "name": "rate"
	           }
	         }
	       ]
	    }]
	} }</pre>

8.14.2 Example: Field Constraint Query Options

This example constructs a simple options node that sets up just one constraint, based on the field named `"bbqtext"`. It creates a word constraint, therefore when you search for

```
summary:"hot dog"
```

It constrains the search to documents that have the phrase `"hot dog"` in the field called `"bbqtext"`.

Format	Options

XML	`<search:options`
	`  xmlns:search="http://marklogic.com/appservices/search">`
	`  <search:constraint name="summary">`
	`  <search:word>`
	`  <search:field name="bbqtext"/>`
	`  </search:word>`
	`  </search:constraint>`
	`</search:options>`

JSON	`{ "options": {`
	`  "constraint": [ {`
	`  "name": "summary",`
	`  "word": {`
	`  "field": { "name": "bbqtext" }`
	`  }`
	`  } ]`
	`} }`

For more details, see “constraint” on page 822 in the query options appendix.

8.14.3 Example: Collection Constraint Query Options

The query options in this example do the following:

- Sets flags that modify the search results (`return-metrics` and `return-qtext`).
- Specifies a builtin `transform-results` function that will return raw document search results.
- Defines a collection constraint that uses the collection URIs.

The collection constraint named `"coll"` uses the collection URIs with `"http://test.com"` stripped off.

Format	Options

XML	`<search:options`
	`  xmlns:search="http://marklogic.com/appservices/search">`
	`  <search:debug>true</search:debug>`
	`  <search:constraint name="coll">`
	`  <search:collection prefix="http://test.com"/>`
	`  </search:constraint>`
	`  <search:return-metrics>false</search:return-metrics>`
	`  <search:return-qtext>false</search:return-qtext>`
	`  <search:transform-results apply="raw">`
	`  <search:preferred-matches/>`
	`  </search:transform-results>`
	`</search:options>`

JSON	`{ "options": {`
	`  "debug": true,`
	`  "constraint": [ {`
	`  "name": "coll",`
	`  "collection": { "prefix": "http://test.com" }`
	`  } ],`
	`  "return-metrics": false,`
	`  "return-qtext": false,`
	`  "transform-results": {`
	`  "apply": "raw",`
	`  "preferred-matches": ""`
	`  }`
	`} }`

For more details, see “collection” on page 839 in the query options appendix.

8.14.4 Example: Path Range Index Constraint Query Options

This example shows how to configure a constraint with a path range index. With this in place, searching for:

```
pindex:low
```

Searches for values less than 5 from the nodes at `/Employee/fn` (in no namespace). It is a string range index, faceted, scoped to documents, with a nice looking unicode label. The facet values are returned with any search.

Format	Options
XML	<search:options xmlns:search="http://marklogic.com/appservices/search"> <search:constraint name="pindex"> <search:range type="xs:string" facet="true" collation="http://marklogic.com/collation/"> <search:path-index>/Employee/fn</search:path-index> <search:fragment-scope>documents</search:fragment-scope> <search:bucket name="low" ge="5">0 to 5</search:bucket> <search:bucket name="medium" lt="10" ge="5" >5 to 10</search:bucket> <search:bucket name="high" lt="15" ge="10" >10 to 15</search:bucket> </search:range> </search:constraint> </search:options>

JSON	{ "options": {
	"constraint": [{
	"name": "pindex",
	"range": {
	"type": "xs:string",
	"facet": true,
	"collation": "http://marklogic.com/collation/",
	"path-index": { "text": "/Employee/fn" },
	"fragment-scope": "documents",
	"bucket": [
	{
	"name": "low",
	"lt": "5",
	"label": "0 to 5"
	},
	{
	"name": "medium",
	"lt": "10",
	"ge": "5",
	"label": "5 to 10"
	},
	{
	"name": "high",
	"lt": "15",
	"ge": "10",
	"label": "10 to 15"
	}
	]
	}
	}]
	} }

8.14.5 Example: Element Attribute Range Constraint Query Options

This example shows an element attribute range index, with computed buckets. When you search, the facets will be filled out depending on the values of `{http://example.com}entry/@date`.

Format	Options
XML	<search:options xmlns:search="http://marklogic.com/appservices/search"> <search:constraint name="date"> <search:range type="xs:dateTime" facet="true"> <search:attribute ns="" name="date"/> <search:element ns="http://example.com" name="entry"/> <search:fragment-scope>documents</search:fragment-scope> <search:computed-bucket name="older" lt="-P1Y" anchor="start-of-year">Older</search:computed-bucket> <search:computed-bucket name="year" lt="P1Y" ge="P0Y" anchor="start-of-year">This Year</search:computed-bucket> <search:computed-bucket name="month" lt="P0M" ge="P1M" anchor="start-of-month">This Month</search:computed-bucket> <search:computed-bucket name="today" lt="P0D" ge="P1D" anchor="start-of-day">Today</search:computed-bucket> <search:computed-bucket name="future" ge="P0D" anchor="now">Future</search:computed-bucket> </search:range> </search:constraint> </search:options>

Format	Options

JSON	{ "options": {
	"constraint": [{
	"name": "date",
	"range": {
	"type": "xs:dateTime",
	"facet": true,
	"attribute": {
	"ns": "",
	"name": "date"
	},
	"element": {
	"ns": "http://example.com",
	"name": "entry"
	},
	"fragment-scope": "documents",
	"computed-bucket": [
	{
	"name": "older",
	"lt": "-P1Y",
	"anchor": "start-of-year",
	"label": "Older"
	},
	{
	"name": "year",
	"lt": "P1Y",
	"ge": "P0Y",
	"anchor": "start-of-year",
	"label": "This Year"
	},
	{
	"name": "month",
	"lt": "P0M",
	"ge": "P1M",
	"anchor": "start-of-month",
	"label": "This Month"
	},
	{
	"name": "today",
	"lt": "P0D",
	"ge": "P1D",
	"anchor": "start-of-day",
	"label": "Today"
	},
	{
	"name": "future",
	"ge": "P0D",
	"anchor": "now",
	"label": "Future"
	}
	]
	}}]}}

8.14.6 Example: Geospatial Constraint Query Options

This example shows geospatial constraint query options. In addition to having a search results configuration setting (`page-length`), this sets up three geospatial constraints and three element constraints.

Format	Options
XML	<search:options xmlns:search="http://marklogic.com/appservices/search"> <search:debug>true</search:debug> <search:return-metrics>false</search:return-metrics> <search:page-length>25</search:page-length> <search:constraint name="geo-elem"> <search:geo-elem> <search:element ns="" name="g-elem-point"/> </search:geo-elem> </search:constraint> <search:constraint name="geo-elem-pair"> <search:geo-elem-pair> <search:lat ns="" name="lat"/> <search:lon ns="" name="long"/> <search:parent ns="" name="g-elem-pair"/> </search:geo-elem-pair> </search:constraint> <search:constraint name="geo-attr-pair"> <search:geo-attr-pair> <search:lat ns="" name="lat"/> <search:lon ns="" name="long"/> <search:parent ns="" name="g-attr-pair"/> </search:geo-attr-pair> </search:constraint> </search:options>

Format	Options

JSON	{ "options": {
	"debug": true,
	"return-metrics": false,
	"page-length": 25,
	"constraint": [
	{ "name": "geo-elem",
	"geo-elem": {
	"element": {
	"ns": "",
	"name": "g-elem-point"
	}
	}
	},
	{ "name": "geo-elem-pair",
	"geo-elem-pair": {
	"lat": {
	"ns": "",
	"name": "lat"
	},
	"lon": {
	"ns": "",
	"name": "long"
	},
	"parent": {
	"ns": "",
	"name": "g-elem-pair"
	}
	}
	},
	{ "name": "geo-attr-pair",
	"geo-attr-pair": {
	"lat": {
	"ns": "",
	"name": "lat"
	},
	"lon": {
	"ns": "",
	"name": "long"
	},
	"parent": {
	"ns": "",
	"name": "g-attr-pair"
	}
	}
	}
	]
	} }

9.0 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
- Boosting Relevance Score With a Secondary Query
- Including a Range or Geospatial Query in Scoring
- Interaction of Score and Quality
- Using cts:score, cts:confidence, and cts:fitness
- Relevance Order in cts:search Versus Document Order in XPath
- Exploring Relevance Score Computation
- Sample cts:search Expressions

9.1 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

You can use the `relevance-trace` option with `cts:relevance-info` to explore score calculations in detail. For details, see "Exploring Relevance Score Computation" on page 442.

9.1.1 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".

9.1.2 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.

9.1.3 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.

9.1.4 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).

9.1.5 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.

9.2 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*.

9.3 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 444.

9.4 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

9.4.1 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.

9.4.2 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 422. 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.

9.4.3 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).

9.5 Boosting Relevance Score With a Secondary Query

You can use `cts:boost-query` to modify the relevance score of search results that match a secondary (or "boosting") query. The following example returns results from all documents containing the term "dog", and assigns a higher score to results that also contain the term "cat". The relevance score of matches for the first query are boosted by matches for the second query.

```
cts:search(fn:doc(),
  cts:boost-query(
    cts:word-query("dog"),
    cts:word-query("cat"))
)
```

As discussed in "Understanding How Scores and Relevance are Calculated" on page 422, many factors affect relevance score, so the exact quantitative effect of a boosting query on relevance score varies. However, the effect is always proportional to the weighting of the boosting query.

For example, suppose the database includes two documents, `/example/dogs.xml` and `/example/llamas.xml` that have the following contents:

```
/example/dogs.xml:
  <data>This is my dog. I do not have a cat.</data>
/example/llamas.xml:
  <data>This is my llama. He likes to spit at dogs.</data>
```

Then an unboosted search for the word "dog" returns the following matches:

```
cts:search(fn:doc(), cts:word-query("dog"))

<data>This is my dog. I do not have a cat.</data>
<data>This is my llama. He likes to spit at dogs.</data>
```

Assume these matches have the same relevance score. If you repeat the search as a boost query with default weight, the first match has a score that is roughly double that of the 2nd match. (The actual score values do not matter, only their relative values.)

```
for $n in (cts:search(fn:doc(),
    cts:boost-query(
      cts:word-query("dog"),
      cts:word-query("cat"))))
return fn:concat(fn:document-uri($n), " : ", cts:score($n))
```

```
==>
/example/dogs.xml : 22528
/example/llamas.xml : 11264
```

If you increase the weight on the boosting query to 10.0, the relevance score of the document containing both terms becomes roughly 10x that of the document that only contains `"dog"`.

```
for $n in (cts:search(fn:doc(),
    cts:boost-query(
      cts:word-query("dog"),
      cts:word-query("cat", (), 10.0))))
return cts:score($n)

==>
/example/dogs.xml : 22528
/example/llamas.xml : 2048
```

If the primary (or "matching") query returns no results, the boosting query is not evaluated. A boosting query is ignored in an XPath expression or any other context in which the score is zero or randomized.

The `BOOST` string query operator allows equivalent boosting in string search; for details, see "Query Components and Operators" on page 68. The `boost-query` structured query component also exposes the same functionality as `cts:boost-query`; for details, see "boost-query" on page 105.

9.6 Including a Range or Geospatial Query in Scoring

By default, range queries do not influence relevance score. However, you can enable range and geospatial queries score contribution using the `score-function` and `slope-factor` options. This section covers the following topics:

- How a Range Query Contributes to Score
- Use Cases for Range Query Score Contributions
- Enabling Range Query Score Contribution
- Understanding Slope Factor
- Performance Considerations
- Range Query Scoring Examples

9.6.1 How a Range Query Contributes to Score

By default, a range query makes no contribution to score. If you enable scoring for a given range query, it has the same impact as a word query. The contribution from a range query is just one of many factors influencing the overall score, especially in a complex query. As with any query, you can use weights to change the influence a range query has on score; for details, see "Using Weights to Influence Scores" on page 425.

The difference between a matching value and the reference value does not contribute directly to the score. A function is applied to the delta, with suitable scaling based on datatype, such that the resulting range is comparable to the term frequency (TF) contribution from a word query. You control the scaling using the slope factor of the function; for details, see "Understanding Slope Factor" on page 433.

The type of function (linear or reciprocal) determines whether values closest to or furthest from the reference value contribute more to the score. The reference value is the constraining value in the query. For example, if a range query expresses a constraint such as "> 5", then the reference value is 5. You cannot choose the function, but you can choose the type of function.

If a document contains multiple matching values, the highest contribution is used in the overall score computation.

9.6.2 Use Cases for Range Query Score Contributions

Range query score contributions are useful in cases such as the following:

- Boost the score of newer documents over similar older documents, where "newness" is a function of dateTime or another numeric element value. For example, boost the score of recently published documents.
- Boost the score based on how close some element value is to a reference value. For example, boost scores for documents containing prices closest to an ideal of $20.
- Boost the score based on how far away some element value is from a reference value. For example, boost scores for items with a price furthest below a maximum of $20.
- Boost the score based on geospatial distance. For example, find all hotels within 5 miles, boosting the scores for those closest to my current location.

For examples of how to realize these use cases, see "Range Query Scoring Examples" on page 436.

9.6.3 Enabling Range Query Score Contribution

Add the `score-function` option to a range or geospatial query constructor to enable score contributions. You can also use the `slope-factor` option to scale the contribution; for details, see "Understanding Slope Factor" on page 433.

For example, the following search boosts the score more for documents with high ratings (furthest from the reference value 0). Setting the slope factor to 10 decreases the range of values that make a distinct contribution and increases the difference between the amount of contribution.

```
(: Scoring for positive ratings in range 1 to 100 :)
cts:search(doc(),
  cts:element-range-query(xs:QName("ratings"), ">", 0,
    ("score-function=linear","slope-factor=10")))
```

For examples of constructing a similar query with other MarkLogic Server APIs, see "Range Query Scoring Examples" on page 436.

You can set the value of `score-function` to one of the following function types:

Score Function	Description
`zero`	Default. The score contribution of the range query is zero.
`reciprocal`	Use a reciprocal function to calculate the scoring contribution. Document values nearer to the reference value receive higher scores.
`linear`	Use a linear function to calculate the scoring contribution. Document values further from the reference value receive higher scores.

You can specify a score function and slope factor with the following XQuery query constructors, or the equivalent structured or QBE range query constructs.

- `cts:element-range-query`
- `cts:element-attribute-range-query`
- `cts:field-range-query`
- `cts:path-range-query`
- `cts:element-geospatial-query`
- `cts:element-child-geospatial-query`
- `cts:element-pair-geospatial-query`
- `cts:element-attribute-pair-geospatial-query`
- `cts:path-geospatial-query`
- `cts:triple-range-query`

9.6.4 Understanding Slope Factor

In addition to specifying a score function for a range query, you can use the `slope-factor` option to specify a multiplier on the slope of the scoring function applied to a range query. The slope factor affects how the range of differences between a matching value and the reference value affect the score contribution. You should experiment with your application to determine the best slope factor for a given range query. This section provides details to guide your experimentation.

The *delta* for a given range query match is the difference between the matching value and the reference value in a range query:

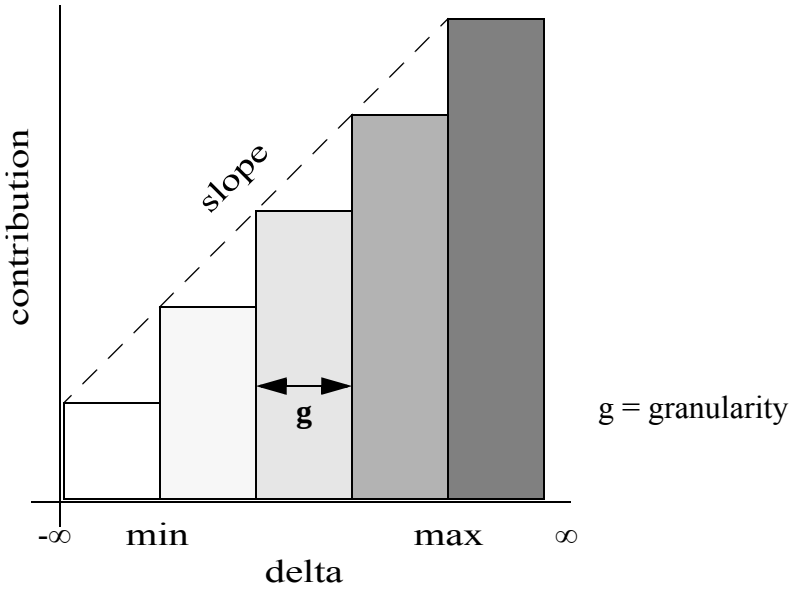
```
delta = reference_value - matching_value
```

For example, if a range query expresses “greater than 5” and the matching value is 3, then the delta is 2. This delta is the basis of the score contribution for a given match, though it is not the actual score contribution.

Each possible delta value does not make a different score contribution because contribution is bucketed. The range of delta values is bounded by a min and max delta value, beyond which all deltas make the same contribution. The granularity represents the size of each bucket within that range. All deltas that fall in the same bucket make the same score contribution, so granularity determines the range of deltas that make a distinct score contribution.

The number of buckets does not change as you vary the slope factor, so changing the slope factor affects the min, max, and granularity of the score function.

The figure below shows the relationship between slope, minimum delta, maximum delta, and granularity for a linear score function.



A slope factor greater than 1 results in finer granularity, but a more narrow range of delta values. A slope factor less than 1 gives a coarser granularity, but a greater range of delta values. Doubling the slope factor with a linear function gives you half the range and half the granularity.

The minimum delta, maximum delta, and granularity for a given slope factor depend upon the type of the range index. The table below shows minimum delta, maximum delta, and granularity for each range index type with the default slope factor (1.0). The granularity is not linear for a reciprocal score function.

Range Index Type	Lower Bound	Upper Bound	Granularity
`integer`	1	1024	4
`float`	1.0	1024.0	~3.98
`double`	1.0	1024.0	~3.98
`decimal`	1.0	1024.0	~3.98
`string`	1	64	1
`point (wgs84)`	1.0 mile ~0.87 deg	100.0 miles ~1.45 degrees	0.39 miles ~0.34 min.
`point (raw)`	1.0	100.0	~0.39
`date`	1 day	1 year	~1.5 days
`time`	1 min	24 h ours	~5.5 min
`dateTime`	1 min	30 days	~2.6 hours
`dayTimeDuration`	1 min	24 hours	~5.5 min
`yearMonthDuration`	1 month	25 years	1 month
`gYear`	1 year	100 years	1 year
`gMonth`	1 month	1 year	1 month
`gDay`	1 day	1 month	1 day
`gYearMonth`	1 month	25 years	1 month

For example, the table contains the following information about range queries over `dateTime` with the default slope factor:

```
Min delta: 1 minute
Max delta: 30 days
Granularity: ~2.6 hours
```

From this, you can deduce the following for a slope factor of 1.0:

- Any delta smaller than 1 minute makes the same contribution as 1 minute
- Any delta greater than 30 days makes the same contribution as 30 days.
- Deltas within ~2.6 hours of each other can make the same contribution. For example, a delta of 5 minutes and a delta of 2 hours make the same contribution because they both fall into the bucket for "1 min. < delta ≤ 2.6 hours".

In a `dateTime` range query where the deltas are on the order of hours, the default slope factor provides a good spread of contributions. However, if you need to distinguish between deltas of a few minutes or seconds, you would increase the slope factor to provide a finer granularity. When you do this, the minimum and maximum delta values get closer together, so the overall range of distinguishable delta values becomes smaller.

Another way to look at slope factor is based on the target minimum or maximum delta. For example, if the default maximum delta for your datatype is 1024 and the range of "interesting" delta values for your range query is only 1 to 100, you probably want to set slope-factor to 10, which lowers the maximum delta to 100 (`1024 div 10`).

9.6.5 Performance Considerations

The performance impact of enabling range query score contributions depends on the nature of your query. The cost is highest for queries that return many matches and queries on strings.

The number of matches affects cost because the scoring calculation is performed for each match. The value type affects the cost because the score calculation is significantly more complex for string values.

Range query score contribution calculations are skipped (and therefore have no negative performance impact) if any of the following conditions apply:

- The `score-function` option is not set or is set to `zero`.
- The range query has a weight of 0.
- The scoring method does not use term frequency. That is, the scoring method is not `score-logtfidf` or `score-logtf`.

9.6.6 Range Query Scoring Examples

This section contains examples that illustrate the use cases outlined in "Use Cases for Range Query Score Contributions" on page 431, plus examples of how to use the feature with additional APIs, such as structured query and QBE.

The following examples are included:

- Example: Most Recently Published
- Example: Closest to a Target Price
- Example: Best Price Below a Maximum
- Example: Closest to a Location
- Example: Use in a Structured Query
- Example: Use in Query By Example

9.6.6.1 Example: Most Recently Published

Boost the score of newer documents over similar older documents, where "newness" is a function of `dateTime` or another numeric element value. The following example boosts the score of recently published documents, where the publication date is stored in a `pubdate` element:

```
cts:element-range-query(
  xs:QName("pubdate"), "<=", current-dateTime(),
  "score-function=reciprocal")
```

The example uses a reciprocal score function so that `pubdate` values closest to "now" contribute the most to the score. That is, the smallest deltas make the biggest contribution.

9.6.6.2 Example: Closest to a Target Price

Boost the score based on how close some element value is to a reference value. The following example boost scores for documents containing prices closest to an ideal of $20, assuming the `price` is an attribute of the `item` element:

```
cts:element-attribute-range-query(
  xs:QName("item"), xs:QName("price"), ">=", 20.0,
  "score-function=reciprocal")
```

The example uses a reciprocal score function so that the smallest deltas between actual and ideal price ($20) make the highest contribution.

9.6.6.3 Example: Best Price Below a Maximum

Boost the score based on how far away some element value is from a reference value. For example, boost scores for items with a price furthest below a maximum of $20:

```
cts:element-attribute-range-query(
  xs:QName("item"), xs:QName("price"), "<=", xs:decimal(20.0),
  ("score-function=linear","slope-factor=51.2"))
```

The example uses a linear function so that the largest deltas between the actual price and the maxiumum price ($20) make the highest contribution.

The slope factor is increased to bring the range of interesting delta values down. As shown in "Understanding Slope Factor" on page 433, the default maximum delta for `xs:decimal` is 1024.0. However, in this example, the interesting deltas are all in the range of 0 to 20.0. To bring the upper bound down to ~20.0, we calculate the slope factor as follows:

```
slope-factor = 1024.0 / 20.0 = 51.2
```

Increasing the slope factor also reduces the granularity, so smaller price differences make different score contributions. With the default slope factor, the granularity is ~3.98, which is very coarse for a delta range of 0-20.0.

9.6.6.4 Example: Closest to a Location

Boost the score based on geospatial distance. For example, find all hotels within 10 miles, boosting the scores for those closest to my current location:

```
cts:and-query(("hotel",
  cts:element-geospatial-query(
    xs:QName("pt"), cts:circle(10, $current-location),
    ("score-function=reciprocal", "slope-factor=10.0"))))
```

The example uses a reciprocal score function so that points closest to the reference location (the smallest deltas) make the greatest score contribution.

The slope factor is increased because the range of interesting delta values is only 0 to 10 ("within 10 miles"). As shown in "Understanding Slope Factor" on page 433, the default maximum delta for a point is 100.0 miles. To bring the maximum delta down to 10.0, slope factor is computed as follows:

```
slope-factor = 100.0 / 10.0 = 10.0
```

9.6.6.5 Example: Use in a Structured Query

The following example is a structured query containing a range query for ratings greater than zero, boosting the score more as the rating increases. Documents with a higher rating receive a higher range query score contribution.

Format	Query
XML	<search:query xmlns:search="http://marklogic.com/appservices/search"> <search:range-query type="xs:integer"> <search:element ns="" name="rating"/> <search:range-operator>GT</range-operator> <search:value>0</value> <search:range-option>score-function=linear</range-option> <search:range-option>slope-factor=10</range-option> </search:range-query> </search:query>

JSON	{ "query": {
	  "queries": [{
	  "range-query": {
	  "type": "xs:integer",
	  "element": {
	  "ns": "",
	  "name": "rating"
	  },
	  "range-operator": "GT",
	  "value": [0],
	  "range-option": [
	  "score-function=linear",
	  "slope-factor=10"
	  ]
	  }
	  }]
	}}

For details, see “Searching Using Structured Queries” on page 74 and the following interfaces:

Interface	Interface	More Information
Search API	`search:resolve`	*XQuery and XSLT Reference Guide*
REST API	GET/POST methods of the `/search` service	Querying Documents and Metadata in *REST Application Developer's Guide*
Java API	`RawStructuredQueryDefinition` or `StructuredQueryBuilder` in `com.marklogic.client.query`	Search Documents Using Structured Query Definition in *Java Application Developer's Guide*

9.6.6.6 Example: Use in Query By Example

The following example is a QBE that contains a range query for ratings greater than zero, boosting the score more as the rating increases. Documents with a higher rating receive a higher range query score contribution.

This query is suitable for use with the REST API `/qbe` service or the Java API `RawQueryByExampleDefinition` interface.

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <rating>`
	`  <q:gt score-function="linear" slope-factor="10">0</q:gt>`
	`  </rating>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{ "$query": {`
	`  "rating": {`
	`  "$gt": 0,`
	`  "$score-function": "linear",`
	`  "$slope-factor": 10`
	`  }`
	`} }`

For details, see “Searching Using Query By Example” on page 195 and the following interfaces:

Interface	Interface	More Information
Search API	`search:resolve`	*XQuery and XSLT Reference Guide*
REST API	GET/POST methods of the `/qbe` service	Using Query By Example to Prototype a Query in *REST Application Developer’s Guide*
Java API	`RawQueryByExampleDefinition` in `com.marklogic.client.query`	Prototype a Query Using Query By Example in *Java Application Developer’s Guide*

9.7 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.

9.8 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.

9.9 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.

9.10 Exploring Relevance Score Computation

You can use the `relevance-trace` search option to explore how the relevance scores are computed for a query. For example, you can use this feature to explore the impact of varying query weight and document quality weight.

> **Note:** Collecting score computation information during a search is costly, so you should only use the `relevance-trace` option when you intend to generate a score computation report from the collected trace.

When you use the `relevance-trace` option on a search, MarkLogic Server collects detailed information about how the relevance score is computed. You can access the information in one of the following ways:

- If you search using `cts:search`, call `cts:relevance-info` on your search results to generate an XML report.
- If you search using the Search API (`search:search` or `search:resolve`), REST API, or Java API, an XML report is automatically returned in the relevance-info section of each search result. (The REST and Java APIs can also return a JSON report.)

The following example generates a score computation report from the results of cts:search.

```
for $x in cts:search(fn:doc(), "example", "relevance-trace")
return cts:relevance-info($x)
```

The resulting score computation report looks similar to the following:

```
<qry:relevance-info xmlns:qry="http://marklogic.com/cts/query">
  <qry:score
    formula="(256*scoreSum/weightSum)+(256*qualityWeight*documentQuality)"
    computation="(256*208/1)+(256*1*0)">53248</qry:score>
  <qry:confidence formula="sqrt(score/(256*8*maxlogtf*maxidf))"
    computation="sqrt(53248/(256*8*18*log(848)))">0.462837</qry:confidence>
  <qry:fitness formula="sqrt(score/(256*8*maxlogtf*avgidf))"
    computation="sqrt(53248/(256*8*18*(3.13196/1)))">0.679113</qry:fitness>
  <qry:uri>/example.xml</qry:uri>
  <qry:path>fn:doc("/example.xml")</qry:path>
  <qry:term weight="3.25">
    <qry:score formula="8*weight*logtf" computation="26*8">208</qry:score>
    <qry:key>16979648098685758574</qry:key>
    <qry:annotation>word("example")</qry:annotation>
  </qry:term>
</qry:relevance-info>
```

Each `qry:score` element contains a `@formula` describing the computation, and a `@computation` showing the values plugged into the formula. The data in the `score` element is the result of the computation. For example:

```
<qry:score
    formula="(256*scoreSum/weightSum)+(256*qualityWeight*documentQuality)"
    computation="(256*154/2)+(256*1*0)">
  19712
</qry:score>
```

The following example generates a score computation report using the XQuery Search API:

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

search:search("example",
  <search:options xmlns="http://marklogic.com/appservices/search">
    <search-option>relevance-trace</search-option>
  </search:options>
)
```

The query generates results similar to the following:

```
<search:response snippet-format="snippet" total="1" start="1" ...>
  <search:result index="1" uri="/example.xml"
      path="fn:doc(&amp;quot;/example.xml&amp;quot;)" score="14336"
      confidence="0.749031" fitness="0.749031">
    <search:snippet>...</search:snippet>
    <qry:relevance-info xmlns:qry="http://marklogic.com/cts/query">
      <qry:score
        formula="(256*scoreSum/weightSum)+(256*qualityWeight*documentQuality)"
        computation="(256*56/1)+(256*1*0)">14336</qry:score>
      <qry:confidence formula="sqrt(score/(256*8*maxlogtf*maxidf))"
        computation="sqrt(14336/(256*8*18*log(2)))">0.749031</qry:confidence>
      <qry:fitness formula="sqrt(score/(256*8*maxlogtf*avgidf))"
        computation="sqrt(14336/(256*8*18*(0.693147/1)))">
        0.749031
       </qry:fitness>
      <qry:uri>/example.xml</qry:uri>
      <qry:path>fn:doc("/example.xml")</qry:path>
      <qry:term weight="0.875">
        <qry:score formula="8*weight*logtf" computation="7*8">56</qry:score>
        <qry:key>16979648098685758574</qry:key>
        <qry:annotation>word("example")</qry:annotation>
      </qry:term>
    </qry:relevance-info>
  </search:result>
  <search:qtext>example</search:qtext>
  ...
</search:response>
```

The REST and Java APIs use the same query options as the above Search API example, and return a report in the same way, inside each `search:result`.

9.11 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

9.11.1 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.

9.11.2 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.

10.0 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

10.1 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 804. 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 30.

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*.

10.2 Creating Lexicons

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

10.3 Word Lexicons

There are several types of word lexicons:

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

10.3.1 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`

10.3.2 Element/Element-Attribute Word Lexicons

An XML element word lexicon or an XML 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.

Create element and element-attribute word lexicons in the Admin Interface with the `Element Word Lexicons` and `Attribute Word Lexicons` links under the database in which you want to create the lexicons. You can also use the following Admin API functions:

- `admin:database-add-element-word-lexicon`
- `admin:database-add-element-attribute-word-lexicon`

Use the following functions to query element and element attribute word lexicons:

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

10.3.3 JSON Property Word Lexicons

A JSON property word lexicon contains all of the unique terms in the specified JSON property, with uniqueness determined by the specified collation. A JSON property word lexicon only contains words occurring in string values of the specified JSON property.

Create a JSON property word lexicon using the interfaces for XML element word lexicons. To create a lexicon with the Admin Interface, use the `Element Word Lexicons` section under the database in which you want to create the lexicon. To create a lexicon with the Admin API, use the function `admin:database-add-element-word-lexicon`.

Use the following functions to query JSON property word lexicons:

- `cts:json-property-words`
- `cts:json-property-word-match`

10.3.4 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`

10.4 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*.

10.5 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*.

10.6 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 453.

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 with a document with the URI `/george.xml` that looks as follows:

```
<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>
```

Before creating this document, 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`.

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
:)
```

10.7 Geospatial Lexicons

The following APIs use geospatial point 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 point index to use its corresponding geospatial lexicon. For example, to use `cts:element-geospatial-values`, you must first create a geospatial element point index. Use the Admin Interface (Databases > *database_name* > Geospatial Point 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.

To learn more about geospatial features in MarkLogic, see “Geospatial Search Applications” on page 476.

10.8 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*.

10.9 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`

10.10 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

10.10.1 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*.

10.10.2 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*.

10.10.3 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 683.

10.10.4 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>
```

11.0 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

11.1 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

11.1.1 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.

11.1.2 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 809.

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*.

11.1.3 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 248.

11.2 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.

11.3 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>
```

For queries against a dateTime index, when `$value` is an `xs:dayTimeDuration` or `xs:yearMonthDuration`, the query is executed as an age query. `$value` is subtracted from `fn:current-dateTime()` to create a `xs:dateTime` used in the query. If there is more than one item in `$value`, they must all be the same type.

For example, given a `dateTime` index on element `startDateTime`, queries `cts:element-range-query(xs:QName ("startDateTime"), ">", xs:dayTimeDuration("P1D"))` and `cts:element-range-query(xs:QName ("startDateTime"), ">", fn:current-dateTime() - xs:dayTimeDuration("P1D"))` are the same; both match values within the last day.

12.0 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

12.1 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*.

12.2 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());`

Interface	Mechanism	Example

Node.js	Specify the aggregate name using `valuesBuilder.aggregates` and use `DatabaseClient.values.read` to perform the computation. For details, see Analyzing Lexicons and Range Indexes with Aggregate Functions in the *Node.js Application Developer's Guide*.	`const vb = marklogic.valuesBuilder;`
		
		`db.values.read(`
		`  vb.fromIndexes('Amount')`
		`  .aggregates('sum')`
		`  .slice(0)`
		`)...`

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 30, the *Java Application Developer's Guide*, the *REST Application Developer's Guide*, or the *Node.js Application Developer's Guide*.

12.3 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 Aggregate 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());`

Node.js	Set the aggregate name and path using `valuesBuilder.udf` and `valuesBuilder.aggregates`, then use `DatabaseClient.values.read` to perform the computation. For details, see Analyzing Lexicons and Range Indexes with Aggregate Functions in the *Node.js Application Developer's Guide*.	`const vb =`
		`marklogic.valuesBuilder;`
		
		`db.values.read(`
		`  vb.fromIndexes('Amount')`
		`  .aggregates(`
		`  vb.udf(`
		`  '/native/samplePlugin,`
		`  'myAvg'))`
		`  .slice(0)`
		`)...`

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 30, the *Java Application Developer’s Guide*, or the *REST Application Developer’s Guide*.

13.0 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*.

13.1 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.

13.1.1 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 652.

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.

13.2 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.

13.3 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

13.3.1 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>
```

13.3.2 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>
```

13.3.3 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>
```

13.3.4 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`).

13.3.5 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.

13.4 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()
```

13.5 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`.

13.6 Common Usage Notes

This section shows some common usage patterns to be aware of 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

13.6.1 Input Must Be a Single Node

The input to `cts:highlight` must be a single node. If you want to highlight query hits from a `cts:search` operation that returns multiple nodes, you must iteratively apply `cts:highlight` to each result.

> **Note:** The input node to `cts:highlight` must be a document node or an element node; it cannot be a text node.

For example, the following query matches all documents that contain MarkLogic, and then highlights each query match by enclosing it in a `b` element. Each result is bound to a variable (`$x`) so `cts:highlight` can be applied to it.

```
for $x in cts:search(fn:doc(), "MarkLogic")
return
cts:highlight($x, "MarkLogic", <b>{$cts:text}</b>)
```

13.6.2 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.

13.6.3 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.

14.0 Geospatial Search Applications

This chapter describes how to use the geospatial features of MarkLogic and describes the type of applications that might use these functions. MarkLogic supports geospatial data represented in either XML or JSON, and supports geospatial search in several languages, include XQuery, Server-Side JavaScript, Java, and Node.js.

This chapter includes the following sections:

- Terms and Definitions
- Licensing Requirements for Geospatial Features
- Geospatial Features Overview
- Understanding Coordinate Systems
- Understanding MarkLogic Geospatial Region Types
- Understanding Geospatial Query and Index Types
- Searching for Matching Points
- Searching for Matching Regions
- Controlling Coordinate System and Precision
- Understanding Tolerance
- Summary of Other Geospatial Operations
- Converting To and From Common Geospatial Representations
- Constructing Geospatial Point and Region Values
- Geospatial Query Support in Other APIs
- Preparing to Run the Examples

14.1 Terms and Definitions

You should be familiar with the following terms and definition before using geospatial features of MarkLogic Server:

Term	Definition
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 longitude coordinate, and the horizontal axis is represented by a latitude coordinate. Together they make up a coordinate system that is used to map places on the Earth. For more details, see "Understanding Geodetic Coordinates" on page 483.
distance	The *distance* between two geospatial objects refers to the geographical closeness of those geospatial objects.
ETRS89	*ETRS89*, or European Terrestrial Reference System 1989, is an earth-centered geodetic coordinate system. This is one of the coordinate systems you can use for computations, search and indexing of geospatial data. For details, see "Multiple Coordinate Systems" on page 480.
point	A geospatial *point* is a discrete location, identified by two coordinates. In a geodetic coordinate system, a point is identified by its latitude and longitude coordinates. For more details, see "Understanding Points" on page 483.
point query	A *point query* matches points in documents against point or other region search criteria. When the criteria are expressed as points, a document matches if a point in the document is equal to the input criteria. When the criteria are expressed as other region types, a document matches if a point in the document is within the input region. Use a region query to match non-point regions in documents.
proximity	The *proximity* of search results is how close matches are to each other in a document. Proximity can apply to any type of search terms, including geospatial search terms. For example, you might want to find the term "dog" within 10 words of a point in a given zip code.
raw	A Euclidean coordinate system. This is one of the coordinate systems you can use for computations, search and indexing of geospatial data. For details, see "Multiple Coordinate Systems" on page 480.

Term	Definition
region	A *region* is a set of points that describe a point, box, circle, polygon, or linestring. For details, see "Understanding Coordinate Systems" on page 483.
region query	A *region query* matches regions in documents against region search criteria. A document matches if a region in the document satisfies a specified relationship with the input regions, such as overlaps, intersects, contains, or within. When searching for matching points, you should usually use a point query instead of a region query. For details, see "Searching for Matching Regions" on page 528.
tolerance	A distance within which two points are considered equal, a point is considered "on" an edge, or two edges are considered "touching", even when the coordinate values do match exactly. For details, see "Understanding Tolerance" on page 558.
WGS84	*WGS84*, or World Geodetic System version 1984, is an earth-centered geodetic coordinate system. This is one of the coordinate systems you can use for computations, search and indexing of geospatial data. For details, see "Multiple Coordinate Systems" on page 480.
WKT	*WKT*, or Well Known Text, is a common string representation of geospatial data. You can convert to and from WKT and the internal MarkLogic representation of a region or point. For details, see "Converting To and From Common Geospatial Representations" on page 562.
WKB	*WKB*, or Well Known Binary, is a common binary representation of geospatial data. You can convert to and from WKT and the internal MarkLogic representation of a region or point. For details, see "Converting To and From Common Geospatial Representations" on page 562.
governing coordinate system	The coordinate system/precision combination in effect during a geospatial operation. For details, see "The Governing Coordinate System" on page 486.

14.2 Licensing Requirements for Geospatial Features

You must have an Advanced Geospatial License Option to use the following geospatial features:

- The functions `geo:complex-polygon-contains`, `geo:complex-polygon-intersects`, `geo.complexPolygonContains`, `geo.complexPolygonIntersects`.
- Double precision coordinates, including wgs84/double, etrs89/double, and raw/double.

- `cts:reverse-query` or `cts.reverseQuery` with geospatial constraints. This is sometimes called “geo alerting”.

No other geospatial features or capabilities in MarkLogic require the Advanced Geospatial License Option.

14.3 Geospatial Features Overview

This section provides a brief overview of key features of the geospatial capabilities of MarkLogic Server. Each topic includes pointers to deeper discussion of the feature. The following topics are covered:

- Search for Points, Polygons, and Other Regions
- Geospatial Type System
- Multiple Coordinate Systems
- Support for Common Geospatial Representations
- Flexible Data Layout
- Support for Single and Double Precision Coordinates
- Geospatial Computational Utility Functions
- Geospatial Format Conversion Functions
- Support in Multiple APIs

14.3.1 Search for Points, Polygons, and Other Regions

In MarkLogic, you can construct searches based on either points (discrete locations) or regions (areas). A geospatial query can match points, polygons, and other regions in your documents against points, boxes, circles, polygons, complex polygons, and linestrings search criteria.

You can compare points for equality to other points or for containment in regions. You can compare polygons and other regions using a rich set of topological operators that includes containment, overlap, and intersection.

For example, you can use geospatial search in MarkLogic to find documents matching criteria such as the following:

- Match points against other points. For example, find documents containing this point.
- Match points within regions. For example, find documents containing a point within this circle.
- Match regions against each other: For example, find documents containing a polygon that overlaps this polygon, or find documents containing a region that intersects this linestring.

Notice that the first two query types match points in documents. These are called point queries. You can only use a point query to test for equality to another point or containment within a region.

To search for regions satisfying relationships such as intersection, containment, and overlap, use a region query.

For more details, see:

- "Understanding Geospatial Query and Index Types" on page 493
- "Searching for Matching Points" on page 509
- "Searching for Matching Regions" on page 528

14.3.2 Geospatial Type System

The geospatial interfaces in MarkLogic operate on a geospatial type hierarchy based on point and region primitive types. The type system includes region "subtypes" for specific region types, such as circle, box, and linestring.

The `cts:point` XQuery type and `cts.point` JavaScript type represents a point. Points are used as building blocks for the region types. The `cts:region` XQuery type and the `cts.region` JavaScript is the base type for all regions.

The geospatial interfaces include constructors for creating points and all supported region types. For example, you can create a polygon value using the `cts:polygon` XQuery constructor or the `cts.polygon` JavaScript constructor.

For more details, see "Constructing Geospatial Point and Region Values" on page 567.

14.3.3 Multiple Coordinate Systems

The geospatial data can be expressed in one of several coordinate systems, including WGS84, ETRS89, and raw. WGS84 and ETRS89 are earth-centered geodetic coordinate systems. Raw is a flat plane, cartesian coordinate system. For more details, see "Supported Coordinate Systems" on page 485.

MarkLogic also supports both single and double precision coordinates for each coordinate system. The precision is coupled with the coordinate system in most contexts. For example, when constructing a geospatial point index, your choice of coordinate system includes both "wgs84" (single precision) and "wgs84/double" (double precision).

For details, see the following topics:

- "Supported Coordinate Systems" on page 485
- "Controlling Coordinate System and Precision" on page 549

14.3.4 Support for Common Geospatial Representations

Many MarkLogic interfaces work with geospatial data in common formats, such as Well Known Text (WKT), Well Know Binary (WKB), KML, GML, and GeoJson.

For more details, see:

- "Converting To and From Common Geospatial Representations" on page 562

14.3.5 Flexible Data Layout

Geospatial data in MarkLogic is stored in XML elements and/or attributes, and JSON properties. The coordinates of a point or region thus stored can be represented in several different ways. You can also identify the location of your geospatial data in several different ways.

For point queries, you can specify the location of coordinates in your documents by XPath expression, XML element name, XML element attribute name, or JSON property name. In addition, the coordinates of a point can be either a single, compound value ("10.5 32.7") or separate latitude and longitude values.

For region queries, specify the location of the region coordinates using an XPath expression. Region coordinates must be stored as WKT or serialized `cts:region` values.

Coordinates can also be stored, indexed, and interpreted as either single or double precision values. The original precision is always preserved in your documents, but the configured precision determines the precision at which coordinates are indexed and interpreted during computations.

For more details, see:

- "Understanding Geospatial Query and Index Types" on page 493
- "Understanding Coordinate Systems" on page 483

14.3.6 Support for Single and Double Precision Coordinates

You can evaluate geospatial queries and create geospatial indexes that interpret coordinates as either single (float) or double precision values. You should usually choose single precision, unless your application requires fine-grained accuracy (less than 1 meter).

The default precision depends on your evaluation context. If you do nothing to explictly configure the precision of your App Server or evaluation context, then single precision is used.

For details, see the following topics:

- "Understanding Coordinate Systems" on page 483

- "Controlling Coordinate System and Precision" on page 549

14.3.7 Geospatial Computational Utility Functions

MarkLogic provides a rich set of geospatial utility functions, including the following:

- Computing distance and bearing computations
- Counting region vertices
- Finding the point at which two arcs intersect
- Generating a set of bounding boxes that cover a region

For more details, see "Summary of Other Geospatial Operations" on page 560.

14.3.8 Geospatial Format Conversion Functions

MarkLogic provides XQuery and JavaScript library modules to translate Metacarta, GML, KML, GeoRSS, and GeoJSON formats to MarkLogic primitive geospatial types.

The functions in these libraries are designed to convert geospatial data in supported formats and convert it into primitive MarkLogic geospatial primitive types for use with geospatial query constructors and other geospatial operations.

For more details, see the following topics:

- "Converting To and From Common Geospatial Representations" on page 562

14.3.9 Support in Multiple APIs

This chapter focuses on performing geospatial queries in MarkLogic Server using the `cts:search` XQuery function or `cts.search` Server-Side JavaScript function. You can also configure and use geospatial search with the following MarkLogic APIs:

- Search API (XQuery or Server-Side JavaScript); see "Appendix: Query Options Reference" on page 816 and "Searching Using Structured Queries" on page 74.
- JSearch API (Server-Side JavaScript); see "Creating JavaScript Search Applications" on page 289 and the examples in this chapter.
- Client APIs for Node.js, Java, and REST; see "Creating Point Queries with the Client APIs" on page 522 and "Creating Region Queries Using the Client APIs" on page 542
- REST Management API (for creating and managing geospatial indexes); see the *Monitoring MarkLogic Guide* and *MarkLogic REST API Reference*.

14.4 Understanding Coordinate Systems

In its most basic form, geospatial data is a set of coordinates. The interpretation of the coordinates is based on a coordinate system. For example, a geodetic coordinate system inteprets the coordinates as latitude and longitude values applying to the surface of the earth.

MarkLogic supports both geodetic and Euclidean coordinate systems.

This section covers the following topics:

- Understanding Points
- Understanding Geodetic Coordinates
- Understanding Euclidean Coordinates
- Supported Coordinate Systems
- Understanding MarkLogic Geospatial Region Types
- The Governing Coordinate System
- How Precision Affects Geospatial Operations

14.4.1 Understanding Points

A point represents a discrete location. In a geodetic coordinate system such as WGS84, a point represents a discrete location on the earth. In a Euclidean coordinate system such as raw, a point represents a discrete location in the Euclidean space.

A point is represented by an ordered pair of numbers called coordinates. In a geodetic coordinate system, these numbers represent latitude and longitude values on the earth; for more details, see “Understanding Geodetic Coordinates” on page 483. In a 2-dimensional Euclidean coordinate system, these numbers represent horizontal (x) and verical (y) values; for more details, see “Understanding Euclidean Coordinates” on page 484.

The `cts:point` XQuery type and `cts.point` JavaScript type represent a point in MarkLogic Server. Use the `cts:point` or `cts.point` constructor to construct a point from a pair of coordinates.

Points are also to used define the other regions in MarkLogic Server, and constructor functions are available for these regions, such `cts:box` in XQuery or `cts.polygon` in JavaScript. To learn about supported region types, see “Understanding MarkLogic Geospatial Region Types” on page 487.

14.4.2 Understanding Geodetic Coordinates

A geodetic coordinate system maps points to locations on the Earth. MarkLogic supports geodetic coordinate systems such as WGS84 and ETRS89.

The coordinates of a point in a geodetic coordinate system represent latitude and longitude positions on the Earth. A point has one latitude coordinate and one longitude coordinate. The latitude coordinate represents the north/south position of the point on the Earth. The longitude coordinate represents the east/west position of the point on the Earth.

Point coordinates are expressed in decimal degrees. Distance is measured in units such as miles, feet, kilometers, and meters.

In a geodetic coordinate system, the shortest distance between two points is a curve called a "geodesic arc" or simply a "geodesic". (In a spherical coordinate system, a geodesic is the same as a "great circle".) The edges of a polygon in a geodetic coordinate system are geodesics, not straight lines.

Latitude values are in the range -90 to 90 degrees. The equator has latitude zero. Negative latitude values are south of the equator, with -90 at the south pole. Positive latitude values are north of the equator, with 90 at the north pole.

Longitude values are in the range -180 to 180 degrees. The Prime Meridian has longitude 0. Negative longitude values span the 180 degrees west of the Prime Meridian. Positive longitude values span the 180 degrees east of the Prime Meridian.

14.4.3 Understanding Euclidean Coordinates

A Euclidean coordinate system maps points to locations on a two-dimensional Euclidean plane. The "raw" coordinate system in MarkLogic is a Euclidean coordinate system; for more details, see "Raw Coordinate System" on page 485.

A Euclidean coordinate can be used to represent non-Earth spatial data in local coordinate systems, such as for mathematical modeling or when projecting geographic points on to a flat plane.

A point in the raw coordinate system is represented by an (x,y) value pair, where x represents the horizontal position on the plane and y represents the vertical position. The interpretation of the coordinates is application specific, as is the range of values.

Point coordinates and distances in a Euclidean coordinate system are interpreted in an application-specific way. The units for x, y, and distance are assumed to be the same. The edges of a polygon are straight lines in a Euclidean coordinate system.

Most of the geospatial interfaces and documentation in MarkLogic refer to the coordinates of a point or region using "latitude" and "longitude" terminology. However, when working with a raw coordinate system, the coordinates do not actually represent latitude and longitude values. Instead, latitude refers to the x coordinate and longitude refers to the y coordinate.

14.4.4 Supported Coordinate Systems

MarkLogic Server supports the following coordinate systems for geospatial data:

- WGS84 Coordinate System
- ETRS89 Coordinate System
- Raw Coordinate System

14.4.4.1 WGS84 Coordinate System

By default, MarkLogic Server uses the World Geodetic System version 1984 (WGS84) as the basis for geocoding. WGS84 is a widely accepted standard for global point representation. WGS84 is an earth-centered geodetic coordinate system with a coordinate system origin at the Earth's center of mass.

WGS84 is widely used for mapping locations on the Earth, and is used by a wide range of services, including satellite services such as Global Positioning System (GPS) and Google Maps. There are other coordinate systems, some of which have advantages or disadvantages over WGS84. For example, some are more accurate in a given region, while others may be used historically in legacy data.

For details on WGS84, see http://en.wikipedia.org/wiki/World_Geodetic_System.

14.4.4.2 ETRS89 Coordinate System

The European Terrestrial Reference System (ETRS89) is an earth-centered, earth-fixed geodetic coordinate system, designed primarily for mapping locations in Europe.

This coordinate system is fixed to the stable part of the Eurasian tectonic plate. As such, it is not subject to continental drift. ETRS89 and WGS84 coordinates are not interchangeable because of this difference in the handling of continental drift.

For more details, see http://en.wikipedia.org/wiki/European_Terrestrial_Reference_System_1989.

14.4.4.3 Raw Coordinate System

The "raw" coordinate is a Euclidean coordinate system.

The coordinates of a point in the raw coordinate system represent a position on a two-dimensional Euclidean plane. For details, see "Understanding Euclidean Coordinates" on page 484.

The raw coordinate system is a simple cartesian coordinate system, best suited for working with non-geospatial data. However, you can use the raw coordinate system to represent geographical points projected on to a flat plane.

14.4.5 The Governing Coordinate System

The *governing coordinate system* is the coordinate system/precision combination in effect during a geospatial operation. It affects the handling of input values, calculations, comparisons, and return values.

A precision is always implied by the coordinate system name. For example, "wgs84" implies single precision, while "wgs84/double" implies double precision. However, some operations accept a `precision` option that enables you to override the precision implicit in the coordinate system name. For details, see "Specifying a Per-Operation Coordinate System and Precision" on page 556.

The governing coordinate system is based on a precedence ordering of the coordinate system and precision specified in the App Server configuration, a main module prolog (XQuery only), and the parameters or options of a geospatial function (from lowest to highest precedence). For details, see "How MarkLogic Selects the Governing Coordinate System" on page 551.

14.4.6 How Precision Affects Geospatial Operations

The governing coordinate system always has an associated precision, either float (single) or double. Some operations allow you to override the precision implied by the coordinate system through an option.

- The original precision of your data is always preserved in your documents.
- Geospatial data is indexed using the precision configured for the index.
- Geospatial points and regions are serialized at the precision of the governing coordinate system.
- Comparison operations on geospatial regions use the precision of the governing coordinate system.
- Functions operating on geospatial data interpret their input, perform their calculations, and return their results using the governing coordinate system.
- Functions that return geospatial points or regions return either single or double precision coordinates, depending on the governing coordinate system.
- Accessor functions for geospatial points or region return either a single or double precision value, depending on the governing coordinate system. This applies to XQuery functions such as `cts:point-latitude`, `cts:circle-radius`, `cts:box-west`, and their Server-Side JavaScript equivalents.
- Latitude and longitude bounds on box functions are not truncated to the single precision range if the governing coordinate system is double precision. This applies to XQuery functions such as `cts:geospatial-boxes` and `cts:element-geospatial-boxes`, and their Server-Side JavaScript equivalents.

- Geospatial operations perform calculations using the precision of the governing coordinate system. This applies to functions such as `cts:distance`, `cts:polygon-contains`, and `cts:bounding-boxes`, and their Server-Side JavaScript equivalents.
- The input pattern parameter to value-match functions can be either single or double precision, depending on the governing coordinate system. This applies to XQuery functions such as `cts:element-geospatial-value-match` and their Server-Side JavaScript equivalents.
- Searches involving geospatial queries use the precision of the governing coordinate system for determining matches and calculating scores.

14.5 Understanding MarkLogic Geospatial Region Types

This section provides a conceptual overview of the types of regions supported by MarkLogic. Points are the building block of most regions; to learn more about points, see "Understanding Points" on page 483.

Most geospatial interfaces in MarkLogic work with geospatial data represented as a `cts:region` (XQuery) or `cts.region` (Server-Side JavaScript), or an equivalent serialization. The cts region type is an abstraction that can represent any of the following concrete geospatial types:

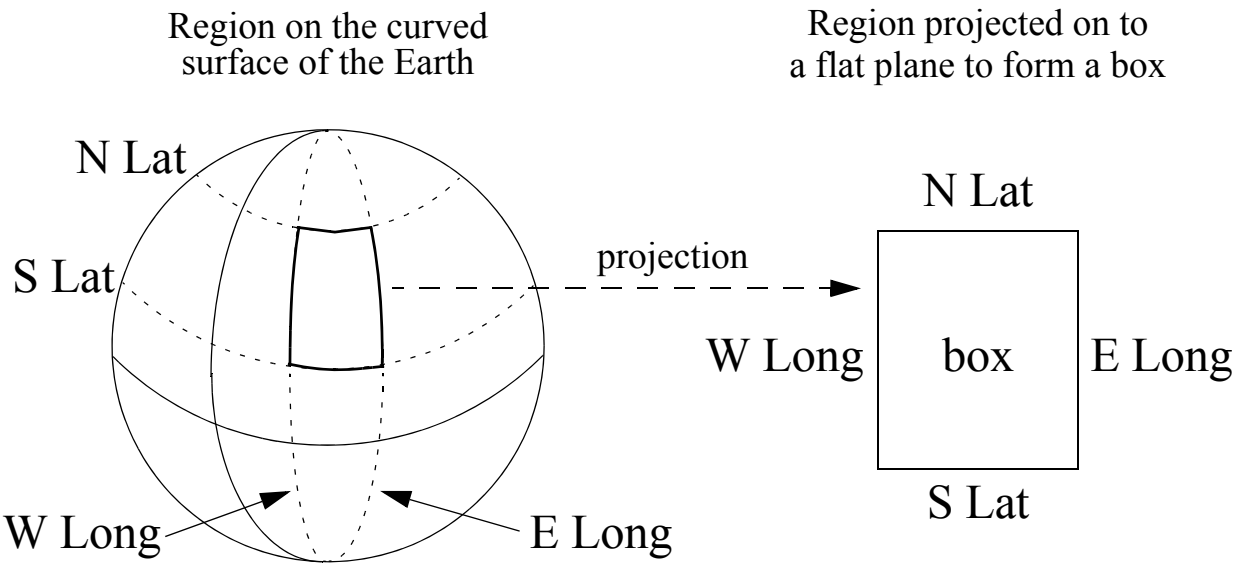
- Boxes
- Polygons
- Complex Polygons
- Linestrings
- Circles

14.5.1 Boxes

A geospatial box is a rectangular region consisting of all the points whose latitude and longitude coordinates are within the region bounds.

In a geodetic coordinate system, a box is a projection from the three-dimensional Earth onto a flat surface. On the surface of the Earth, the edges of a box are arcs. When you project the edges onto a flat plane, they become two-dimensional latitude and longitude lines, and the space defined by those lines forms a rectangle.

The following diagram uses a plate caree projection to illustrate the difference between the region defined by a box on the surface of the Earth and its projection into a rectangular region on a flat plane.



Projecting coordinates from the curved earth into a flat box

Warning In a geodetic coordinate system, the north and south edges of a box are latitude lines, not geodesic arcs. The east and west edges of a box are longitude lines, which are geodesic arcs. A box is not equivalent to a polygon with the same four vertices.

A point is contained in a box if its latitude coordinate is between the north and south latitude coordinates of the box, and its longitude coordinate is between the west and east longitude coordinates of the box.

In a Euclidean coordinate system, a box is simply a rectangle with boundaries defined by north, south, east, and west coordinates. In a Euclidean coordinate system, a box is equivalent to a polygon with the same four vertices.

The following assumptions and restrictions only apply to boxes in a geodetic coordinate system:

- In a geodetic coordinate system, the west/east extent of a box is determined by starting at the western longitude coordinate and heading east toward the eastern longitude coordinate. If the west coordinate is less than the east coordinate, the box will not cross the anti-meridian. If the east coordinate is less than the west coordinate, the box crosses the anti-meridian.
- In a geodetic coordinate system, the south/north extent of a box is determined by starting at the southern latitude coordinate and heading north to the northern latitude coordinate.

However, you cannot cross the pole: The northern coordinate must be greater than the southern coordinate.

The following assumptions and restrictions apply to boxes under both geodetic and Euclidean coordinate systems:

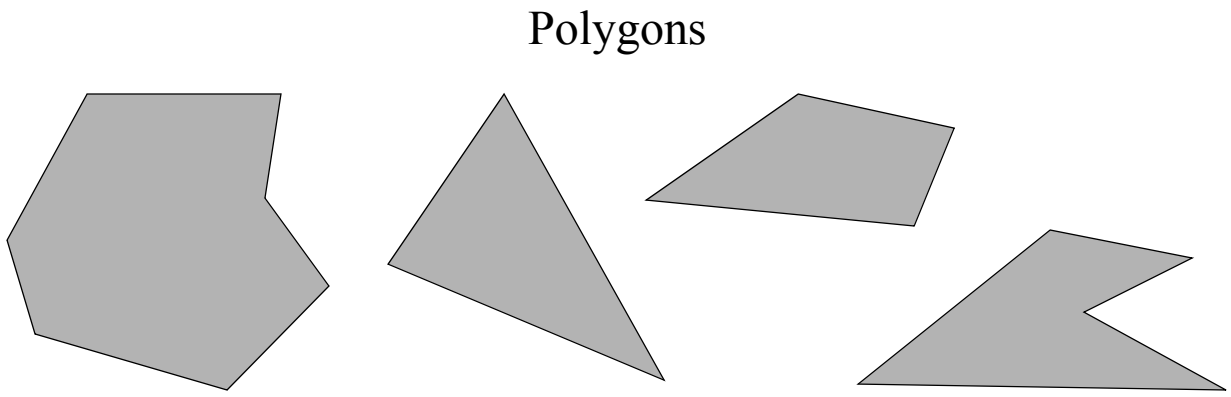
- If the western and eastern coordinates are the same, the box is a meridian line segment between the southern and northern coordinates passing through that longitude coordinate.
- If the southern and northern coordinates are the same, the box is a latitude line segment between the western and eastern coordinates passing through that longitude coordinate.
- If the western and eastern coordinates are the same, and the southern and northern coordinates are the same, then the box is a point specified by those coordinates.
- During a search, the query options determine whether the boundaries of a box are included in or excluded from the box. Various boundary options on the geospatial query constructors control this behavior).

In the "raw" coordinate system, the western coordinates are always less than or equal to the eastern coordinates, and the southern coordinates are always less than or equal to the northern coordinates.

The `cts:box` XQuery type and `cts.box` JavaScript type represent a box in MarkLogic Server. You can create a box using the `cts:box` XQuery constructor or the `cts.box` JavaScript constructor. You can also create a box using one of the conversion utility functions such as `geogml:box` (XQuery) or `geojson.box` (JavaScript). For more details, see "Constructing Geospatial Point and Region Values" on page 567.

14.5.2 Polygons

A geospatial polygon is a region with three or more sides. The following diagram illustrates several polygons.



In a geodetic coordinate system, a polygon can represent any area on the Earth (with the exceptions described below). For example, you might create a polygon to represent a country or a geographical region.

Polygons offer a large degree of flexibility compared to circles or boxes. In exchange for the flexibility, operations on geospatial polygons are not quite as fast or accurate as geospatial box and circle operations.

The efficiency of polygon operations 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 assumptions and restrictions apply to polygons only under a geodetic coordinate system in MarkLogic:

- A geodetic coordinate system treats the earth as an ellipsoid. In such a system, the edges of a polygon are geodesic arcs, not latitude lines.
- A polygon cannot include both poles and cannot have both poles as a boundary (regardless of whether the boundaries are included). Thus, a polygon cannot encompass the full 180 degrees of latitude.
- The span of the arc described by a polygon edge in a geodetic coordinate system must be between 0 and 180 degrees and cannot cross a pole. If you need to span more than 180 degrees, define multiple edges that cover the desired span.

  Note: Latitude lines are distinct from geodesic arcs. Except for the equator, the shortest distance between two points at the same latitude does not follow the latitude line. The edges of polygons are geodesic arcs, not latitude lines. You can approximate a latitude line by adding vertices evenly spaced along the latitude line. The north and south edges of a box are latitude lines; if the region to be described is a box, use a `cts:box` or `cts.box` instead of a polygon.

The following assumptions and restrictions apply to polygons under either a geodetic or Euclidean coordinate system in MarkLogic.

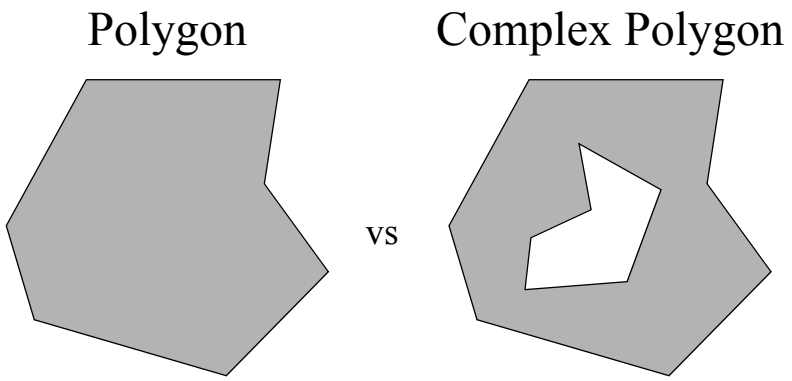
- No two edges of a polygon or complex polygon may overlap or cross.
- Coordinate system is considered at search time rather than when you construct a polygon value. Therefore, a search will throw a runtime exception if a polygon is not valid for the governing coordinate system.
- The boundaries of a polygon are either in or out of the polygon, depending on the operation and query options. The DE9IM operators include specific boundary behaviors; for other operations, you can use query constructor options to control the boundary behavior.

You can construct a 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.

For details, see the `cts:polygon` XQuery function or the `cts.polygon` JavaScript function.

14.5.3 Complex Polygons

A complex polygon is a polygon with one more holes. For example, the following graphic illustrates the difference between polygon and a complex polygon. The complex polygon is the shaded region in the region on the right. The unshaded region, or inner polygon, represents a hole in the outer polygon.



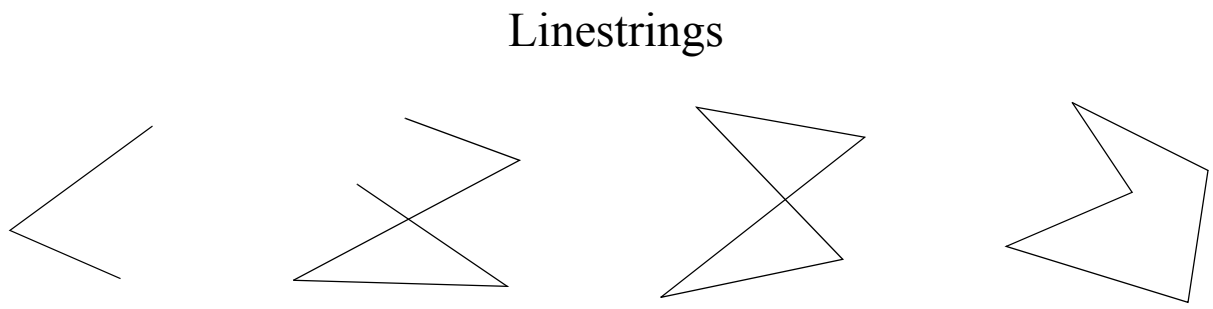
You can construct a complex polygon by constructing an outer polygon with zero or more outer polygons. All inner polygons must be completely contained in the interior of the outer polygon. No two edges can cross or overlap. Use the `cts:complex-polygon` XQuery function or the `cts.complexPolygon` JavaScript function to construct a complex polygon.

You can also cast a `cts:complex-polygon` or `cts.complexPolygon` with no holes (that is, with no inner polygons) to a `cts:polygon` or `cts.polygon`. If you specify multiple inner polygons, none of them should overlap each other.

14.5.4 Linestrings

A linestring is a connected sequence of edges. In a geodetic coordinate system, edges are geodesic arcs. In a Euclidean coordinate system such as “raw”, the edges are straight lines.

A linestring does not necessarily form a closed loop as the boundary of a polygon does, although it is permissible for a linestring to form a closed loop. The following diagram demonstrates some examples of linestrings.

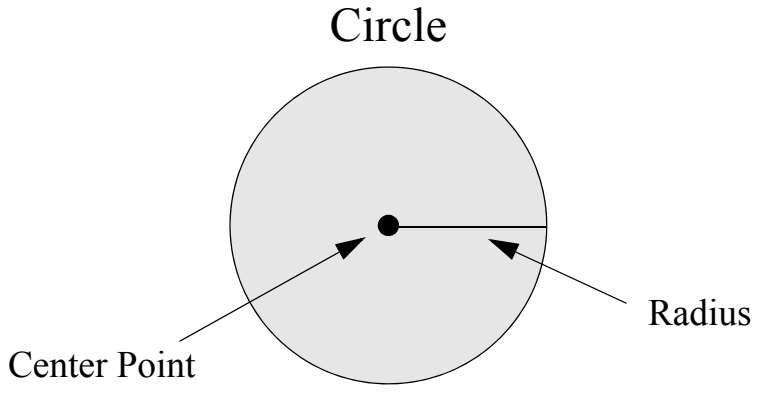


You can compare linestrings for equality or inequality. Two linestrings are equal if all of their vertices are equal, or if they are both empty.

To construct a linestring, use the `cts:linestring` XQuery function or the `cts.linestring` JavaScript function.

14.5.5 Circles

A geospatial circle consists of all the points within a certain distance (the radius) of a given center point. A geospatial region that represents a circle is defined by its center point and radius. The points that are the distance of the radius from the center define the boundary of the region.



Use the `cts:circle` XQuery function or the `cts.circle` JavaScript function to construct a circle.

14.6 Understanding Geospatial Query and Index Types

This topic discusses the types of geospatial query you can create, the index types that support each query type, and the data layout expected by each query and index type. The following topics are covered:

- Introduction to Geospatial Query and Index Types
- Geospatial Query Creation
- Geospatial Index Creation
- Geospatial XML Element Point Queries and Indexes
- Geospatial XML Element Child Point Queries and Indexes
- Geospatial XML Element Pair Point Queries and Indexes
- Geospatial XML Attribute Pair Point Queries and Indexes
- Geospatial Path Point Queries and Indexes
- Geospatial JSON Property Point Queries and Indexes
- Geospatial JSON Property Child Point Queries and Indexes
- Geospatial JSON Property Pair Point Queries and Indexes
- Geospatial Region Queries and Indexes
- Geospatial Index Positions
- Geospatial Lexicons
- Index Reference Resolution

14.6.1 Introduction to Geospatial Query and Index Types

MarkLogic supports several types of query for searching geospatial data contained in documents. In general, geospatial queries fall into the following two categories, based on the kind of geospatial document content to be matched:

- Point query: Match points in documents against points or other regions specified as input criteria. For example, “Find all documents containing a point within this circle.”
- Region query: Match other regions in documents that satisfy one of a number of relationships when compared to regions specified as input criteria. For example, “Find all documents containing polygons that intersect with this polygon.”

For best performance, a point query should be supported by a corresponding geospatial index. A region query always requires a backing geospatial region index. MarkLogic supports several types of geospatial index, corresponding to the different geospatial query types.

Select a geospatial point query or index type based on the layout of your data. The query or index type varies depending on whether the data is represented in XML or JSON, and whether the point coordinates are represented as a single compound value (“lat lon”) or as distinct latitude and longitude values. For example, you might use a `cts:element-geospatial-query` and a geospatial element index for points represented as a single compound XML element value.

The data layout for a region query or region index must be WKT or a serialized cts region, such as a `cts:polygon`. The region data is located within a document using an XPath expression when creating a query or index. Therefore, you use a `cts:geospatial-region-query` and a geospatial region path index for querying by region.

The following table summarizes the query and index types MarkLogic supports, based on the axes of geospatial content type (point or other region) and layout.

Geo Content to Search	Identified By	Example Data Layout	More Information
Point	XPath Expression	Any coordinate pair addressable with an indexable XPath expression. `/Placemark/Point/coordinates`	Geospatial Path Point Queries and Indexes
	XML Layout	`<coords>1.0 2.0</coords>`	Geospatial XML Element Point Queries and Indexes

		`<container>`
		`  <coords>1.0 2.0</coords>`
		`<container>`
			Geospatial XML Element Child Point Queries and Indexes

		`<coords>`
		`  <lat>1.0</lat>`
		`  <lon>2.0</lon>`
		`</coords>`
			Geospatial XML Element Pair Point Queries and Indexes

		`<coords lat="1.0" lon="2.0" />`	Geospatial XML Attribute Pair Point Queries and Indexes

	JSON Layout	`{"coords": "1.0 2.0" }`
		
		`{"coords": [1.0, 2.0] }`
			Geospatial JSON Property Point Queries and Indexes

		`{"container": {`
		`  {"coords": "1.0 2.0" }`
		`}}`
		
		`{"container": {`
		`  {"coords": [1.0, 2.0] }`
		`}}`
			Geospatial JSON Property Child Point Queries and Indexes

		`{"coords": {`
		`  "lat": 1.0,`
		`  "lon": 2.0`
		`}}`
			Geospatial JSON Property Pair Point Queries and Indexes

Other Region	XPath Expression	Any serialized cts region or WKT value addressable with an indexable XPath expression. `/envelope/cts-region`	Geospatial Region Queries and Indexes

14.6.2 Geospatial Query Creation

You can create a geospatial cts query in the following ways.

- Using an XQuery or Server-Side JavaScript query constructor, such as `cts:element-geospatial-query` (XQuery) or `cts.pathGeospatialQuery` (JavaScript).
- Parsing query text containing a geospatial search term. For details, see "Constructing a Point Query in XQuery" on page 518 or "Constructing a Region Query from Query Text" on page 540.

You can also create a geospatial structured query or Query By Example for use with the Search API or the Client APIs. The Java and Node.js Client APIs include builder interfaces for creating structured queries.

For more details, see the sections on each query/index type elsewhere in this section and the following topics:

- "Searching for Matching Points" on page 509
- "Searching for Matching Regions" on page 528

14.6.3 Geospatial Index Creation

Region queries require a region index, but an index is optional for some point queries. For best performance, you should usually create a geospatial index for both query types.

> **Note:** You must have a valid geospatial license key to create or use any geospatial indexes.

Use a geospatial region path index when matching regions in your documents. Use a geospatial point index when matching points in your documents; the type of point index depends on the layout of your content . For details, see "Introduction to Geospatial Query and Index Types" on page 493.

When creating a point index, you can specify the coordinate system, coordinate value precision, and point type (long-lat or lat-long). When creating a region index, you can specify the coordinate system and geohash precision. The default coordinate system is WGS84. The default coordinate precision is float (single precision), and the default point type is "point" (lat-long).

When you create an index using the Admin API, index properties such as coordinate system and precision are specified through the index reference constructor function, such as `admin:database-geospatial-element-index` or `admin:database-geospatial-region-path-index`. For an example of index creation using the Admin API, see "Configuring the Indexes" on page 573.

You can create a geospatial index using the following methods:

- Interactively, using the Admin Interface. See the Geospatial Point Indexes or Geospatial Region Indexes section under Database > *database_name* in the Admin Interface.
- Programmatically, using the server-side Admin API functions. For example, to create a geospatial element index, use the XQuery function `admin:database-add-geospatial-element-index` or the JavaScript function `admin.databaseAddGeospatialElementIndex`.
- Programmatically, using the REST Management API. For details, see the `PUT:/manage/v2/databases/{id|name}/properties` method.

For more details, see the sections on each query/index type, below.

14.6.4 Geospatial XML Element Point Queries and Indexes

Use a geospatial element query when the point coordinates in your documents are represented as the value of a single XML element, with the latitude and longitude values separated by whitespace or punctuation (except `+`, `-`, or .). For example:

```
<coords>37.52  -122.25</coords>
```

By default, the first coordinate is the latitude value, and the second coordinate is the longitude value. You can override the default order by specifying a longitude-first ordering when creating queries and indexes.

If the element value contains other coordinates, they are ignored. For example, KML data can include an additional altitude coordinate. The altitude can be present but is ignored.

When you use a geospatial element query, you should also create a corresponding geospatial element index for best performance.

For JSON documents with similar layout, see “Geospatial JSON Property Point Queries and Indexes” on page 503.

You can use the following interfaces to create a geospatial element query:

Interface	Query Constructor
XQuery	`cts:element-geospatial-query`
Server-Side JavaScript	`cts.elementGeospatialQuery`
Structured Query	geo-elem-query
Java Client API	`com.marklogic.client.query.StructuredQueryBuilder.geoElement` plus `com.marklogic.client.query.StructuredQueryBuilder.geospatial`
Node.js Client API	queryBuilder.geoElement

You can use the following interfaces to create a geospatial element index:

Interface	Index Construction Method
Admin Interface	`Databases >...> Geospatial Point Indexes > Geospatial Element Indexes`
XQuery Admin API (also usable with JavaScript)	`admin:database-add-geospatial-element-index`
REST Management API	`PUT:/manage/v2/databases/{id|name}/properties`

14.6.5 Geospatial XML Element Child Point Queries and Indexes

Use a geospatial element child index for geospatial point data when the coordinates are contained in an XML element value, separated by whitespace or punctuation (except +, -, or .), and you want to identify the container element as a child of another specific element. For example:

```
<parent-name>
  <child-name>37.52  -122.25</child-name>
</parent-name>
```

By default, the first coordinate is the latitude value, and the second coordinate is the longitude value. You can override the default order by specifying a longitude-first ordering when creating queries and indexes.

If the element value contains other coordinates, they are ignored. For example, KML data can include an additional altitude coordinate. The altitude can be present but is ignored.

When you use a geospatial element child query, you should also create a corresponding geospatial element child index for best performance.

For JSON documents with similar layout, see "Geospatial JSON Property Child Point Queries and Indexes" on page 504.

You can use the following interfaces to create a geospatial element child query:

Interface	Query Constructor
XQuery	`cts:element-child-geospatial-query`
Server-Side JavaScript	`cts.elementChildGeospatialQuery`
Structured Query	geo-elem-query
Java Client API	`com.marklogic.client.query.StructuredQueryBuilder.geoElement` and `com.marklogic.client.query.StructuredQueryBuilder.geospatial`
Node.js Client API	queryBuilder.geoElement

You can use the following interfaces to create a geospatial element child index:

Interface	Index Construction Method
Admin Interface	`Databases > ... > Geospatial Point Indexes > Geospatial Element Child Indexes`
XQuery Admin API (also usable with JavaScript)	`admin:database-add-geospatial-element-child-index`
REST Management API	`PUT:/manage/v2/databases/{id|name}/properties`

14.6.6 Geospatial XML Element Pair Point Queries and Indexes

Use a geospatial element pair index for geospatial point data when the longitude and latitude are values in two different elements that are children of the same parent element. For example:

```
<container-name>
  <latitude>37.52</latitude>
  <longitude>-122.25</longitude>
</container-name>
```

For JSON data requirements, see “Geospatial JSON Property Pair Point Queries and Indexes” on page 505.

You can use the following interfaces to create a geospatial element pair query:

Interface	Query Constructor
XQuery	`cts:element-pair-geospatial-query`
Server-Side JavaScript	`cts.elementPairGeospatialQuery`
Structured Query	geo-elem-pair-query
Java Client API	`com.marklogic.client.query.StructuredQueryBuilder.geoElementPair` and `com.marklogic.client.query.StructuredQueryBuilder.geospatial`
Node.js Client API	queryBuilder.geoElementPair

You can use the following interfaces to create a geospatial element pair index:

Interface	Index Construction Method
Admin Interface	`Databases > ... > Geospatial Point Indexes > Geospatial Element Pair Indexes`
XQuery Admin API (also usable with JavaScript)	`admin:database-add-geospatial-element-child-index`
REST Management API	`PUT:/manage/v2/databases/{id|name}/properties`

14.6.7 Geospatial XML Attribute Pair Point Queries and Indexes

Use a geospatial attribute pair index for geospatial point data when the longitude and latitude are values in two different attributes of the same parent XML element. For example:

```
<element-name latitude="37.52" longitude="-122.25"/>
```

When you use a geospatial attribute pair query, you should also create a corresponding geospatial attribute pair index for best performance.

You can use the following interfaces to create a geospatial element attribute pair query:

Interface	Query Constructor
XQuery	`cts:element-attribute-pair-geospatial-query`
Server-Side JavaScript	`cts.elementAttributePairGeospatialQuery`
Structured Query	geo-attr-pair-query
Java Client API	`com.marklogic.client.query.StructuredQueryBuilder.geoElement and com.marklogic.client.query.StructuredQueryBuilder.geospatial`
Node.js Client API	queryBuilder.geoElement

You can use the following interfaces to create a geospatial element attribute pair index:

Interface	Index Construction Method
Admin Interface	`Databases > ... > Geospatial Point Indexes > Geospatial Attribute Pair Indexes`
XQuery Admin API (also usable with JavaScript)	`admin:database-add-geospatial-element-attribute-pair-index`
REST Management API	`PUT:/manage/v2/databases/{id|name}/properties`

14.6.8 Geospatial Path Point Queries and Indexes

Use a geospatial path query and index for matching points when you want to express the location of the points using an XPath expression. The data layout must be one of the following:

- A single XML element value with the latitude and longitude coordinates separated by whitespace or punctuation, as for a geospatial element query.
- A single JSON property value with the latitude and longitude coordinates separated by whitespace or punctuation, as for a geospatial JSON property query.
- A JSON array value containing a latitude element and a longitude element, as for a geospatial JSON property query.

By default, the first coordinate is the latitude value, and the second coordinate is the longitude value. You can override the default order by specifying a longitude-first ordering when creating queries and indexes.

The path expression with which you define the index is limited to a subset of XPath for performance reasons. For details, see Path Field and Path-Based Range Index Configuration in the *XQuery and XSLT Reference Guide*.

The following table demonstrates the XPath expression to use when creating a path range index for several forms of example geospatial data.

Document Type	Example Data	Indexing Path Expression

XML	`<a:data>`
	`  <a:geo>37.52 -122.25</a:geo>`
	`</a:data>`
		`/a:data/a:geo`

XML	`<a:data>`
	`  <a:geo data="37.52 -122.25"/>`
	`</a:data>`
		`/a:data/a:geo/@data`

JSON	`{ "geometry" : {`
	`  "type": "Point",`
	`  "coordinates": [37.52,-122.25]`
	`  }`
	`}`
		`/geometry[type="Point"]/array-node("coordinates")`

Note: Once you create a geospatial path range index, you cannot change the path expression. To change the path, you must remove the existing geospatial path range index and create a new one.

You can use the following interfaces to create a geospatial path query:

Interface	Query Constructor
XQuery	`cts:path-geospatial-query`
Server-Side JavaScript	`cts.pathGeospatialQuery`
Structured Query	geo-path-query
Java Client API	`com.marklogic.client.query.StructuredQueryBuilder.geoPath` and `com.marklogic.client.query.StructuredQueryBuilder.geospatial`
Node.js Client API	queryBuilder.geoPath and queryBuilder.geospatial

You can use the following interfaces to create a geospatial path index:

Interface	Index Construction Method
Admin Interface	`Databases > ... > Geospatial Point Indexes > Geospatial Path Indexes`
XQuery Admin API (also usable with JavaScript)	`admin:database-add-geospatial-path-index`
REST Management API	`PUT:/manage/v2/databases/{id|name}/properties`

14.6.9 Geospatial JSON Property Point Queries and Indexes

Use a geospatial element index to index geospatial data in JSON documents when the point coordinates are contained in a single JSON property. The geospatial data must be represented in the property value as either whitespace/punctuation separated values in a string, or as an array of values. For example:

```
"prop-name": "37.52 -122.25"

"prop-name": [37.52, -122.25]
```

By default, the first coordinate is the latitude value, and the second coordinate is the longitude value. You can override the default order by specifying a longitude-first ordering when creating queries and indexes. The property value can include other entries, but they are ignored (for example, KML has an additional altitude coordinate, which can be present but is ignored).

You can use the following interfaces to create a geospatial JSON property query:

Interface	Query Constructor
XQuery	`cts:json-property-geospatial-query`
Server-Side JavaScript	`cts.jsonPropertyGeospatialQuery`
Structured Query	geo-json-property-query
Java Client API	`com.marklogic.client.query.StructuredQueryBuilder.geoJSONProperty and com.marklogic.client.query.StructuredQueryBuilder.geospatial`
Node.js Client API	queryBuilder.geoProperty

You can use the following interfaces to create an index for a geospatial JSON property query. Note you should create a geospatial element index, even though you are indexing JSON content.

Interface	Index Construction Method
Admin Interface	`Databases > ... > Geospatial Point Indexes > Geospatial Element Indexes`
XQuery Admin API (also usable with JavaScript)	`admin:database-add-geospatial-element-index`
REST Management API	`PUT:/manage/v2/databases/{id|name}/properties`

14.6.10 Geospatial JSON Property Child Point Queries and Indexes

Use a geospatial element child index to index geospatial data in JSON when you want to limit the index to coordinate properties contained in a specific property. The geospatial data must be represented in the child property value as either whitespace/punctuation separated values in a string, or as an array of values.

For example, if your data looks like one of the following, you could create a geospatial element child index specifying "`theParent`" as the parent element (property) and "`theChild`" as the child element (property).

```
"theParent": {
  "theChild": "37.52 -122.25"
}

"theParent": {
  "theChild": [37.52, -122.25]
}
```

By default, the first coordinate is the latitude value, and the second coordinate is the longitude value. You can override the default order by specifying a longitude-first ordering when creating queries and indexes. The property value can include other entries, but they are ignored (for example, KML has an additional altitude coordinate, which can be present but is ignored).

You can use the following interfaces to create a geospatial JSON property child query:

Interface	Query Constructor
XQuery	`cts:json-property-child-geospatial-query`
Server-Side JavaScript	`cts.jsonPropertyChildGeospatialQuery`
Structured Query	geo-json-property-query
Java Client API	`com.marklogic.client.query.StructuredQueryBuilder.geoJSONProperty` and `com.marklogic.client.query.StructuredQueryBuilder.geospatial`
Node.js Client API	queryBuilder.geoProperty

You can use the following interfaces to create an index for a geospatial JSON property child query. Note you should create a geospatial element child index, even though you are indexing JSON content.

Interface	Index Construction Method
Admin Interface	`Databases > ... > Geospatial Point Indexes > Geospatial Element Child Indexes`
XQuery Admin API (also usable with JavaScript)	`admin:database-add-geospatial-element-child-index`
REST Management API	`PUT:/manage/v2/databases/{id|name}/properties`

14.6.11 Geospatial JSON Property Pair Point Queries and Indexes

Use a geospatial element pair index to index geospatial data in JSON when the point coordinates are contained in sibling JSON properties. For example, use this type of index when working with data similar to the following:

```
"theParent" : {
  "latitude": 37.52,
  "longitude": -122.25
}
```

You can use the following interfaces to create a geospatial JSON property pair query:

Interface	Query Constructor
XQuery	`cts:json-property-pair-geospatial-query`
Server-Side JavaScript	`cts.jsonPropertyPairGeospatialQuery`
Structured Query	geo-json-property-pair-query
Java Client API	`com.marklogic.client.query.StructuredQueryBuilder.geoJSONPropertyPair` and `com.marklogic.client.query.StructuredQueryBuilder.geospatial`
Node.js Client API	queryBuilder.geoPropertyPair

You can use the following interfaces to create an index for a geospatial JSON property pair query. Note you should create a geospatial element pair index, even though you are indexing JSON content.

Interface	Index Construction Method
Admin Interface	`Databases > ... > Geospatial Point Indexes > Geospatial Element Pair Indexes`
XQuery Admin API (also usable with JavaScript)	`admin:database-add-geospatial-element-pair-index`
REST Management API	`PUT:/manage/v2/databases/{id|name}/properties`

14.6.12 Geospatial Region Queries and Indexes

Use a geospatial region path index to index geospatial regions, such as polygons, rather than points. A geospatial region path index supports operations such as the `cts:geospatial-region-query` XQuery function and the `cts.geospatialRegionQuery` JavaScript function. These functions enable you to test for relationships between regions, such as overlaps and contains.

Note: Region indexes over geodetic coordinate systems are based on geohashing. Geohashes of circles are calculated by approximating the circle by a polygon. The approximation is accurate to within 0.001% of the radius of the circle. If you require more precision, use `geo:circle-polygon` to convert circles in your data.

Note: When working with large circular regions, you might need to adjust the tolerance in your geospatial operations. For details, see "Understanding Tolerance" on page 558.

The path expression with which you define a region index is limited to a subset of XPath for performance reasons. For details, see Path Field and Path-Based Range Index Configuration in the *XQuery and XSLT Reference Guide*.

The content referenced by the path expression in a geospatial region index must be a region represented as either WKT or a serialized cts:region. For example:

Format	Example Data	Indexing Path Expression

XML	`<a:data>`
	`  <a:region>POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))</a:region>`
	`</a:data>`
		`/a:data/a:region`

XML	`<a:data>`
	`  <a:loc region="POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))"/>`
	`</a:data>`
		`/a:data/a:loc/@region`

JSON	`{ "location" : {`
	`  "region": "POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))"`
	`  }`
	`}`
		`/location/region`

If your data is not in the expected format, you can use an "envelope pattern" to encapsulate your original data along with a supported format. For more details, see "Example: Using the Envelope Pattern to Encode Regions" on page 548.

You can use the following interfaces to create a geospatial region path query. For more details, see "Searching for Matching Regions" on page 528.

Interface	Query Constructor
XQuery	`cts:geospatial-region-query`
Server-Side JavaScript	`cts.geospatialRegionQuery`
Structured Query	geo-region-path-query and geo-region-constraint-query

Interface	Query Constructor
Java Client API	`com.marklogic.client.query.StructuredQueryBuilder.geoRegionPath` and `com.marklogic.client.query.StructuredQueryBuilder.geospatial`
Node.js Client API	queryBuilder.geoPath and queryBuilder.geospatialRegion

You can use the following interfaces to create a geospatial region path index.

Interface	Index Construction Method
Admin Interface	`Databases > ... > Geospatial Region Indexes`
XQuery Admin API (also usable with JavaScript)	`admin:database-add-geospatial-region-path-index`
REST Management API	`PUT:/manage/v2/databases/{id|name}/properties`

14.6.13 Geospatial Index Positions

Each geospatial point index has a range value positions option. Enabling range value positions speeds up queries that constrain a search by the distance between geospatial data and other search terms in a document, such as when using `cts:near-query` in XQuery or `cts.nearQuery` in Javascript.

Additionally, enabling element positions improves index resolution (more accurate estimates) for XML element and JSON property queries that involve geospatial point queries (with a geospatial index with positions enabled for the geospatial data).

14.6.14 Geospatial Lexicons

Geospatial point 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 453.

14.6.15 Index Reference Resolution

Many geospatial operations either require or will take advantage of available geospatial indexes. Depending on the operation, the index reference might be explicit or implicit. For example, if you supply a `cts:reference` to an operation, the index reference is explicit. By contrast, when you supply an XPath expression, XML element QName, or JSON property name to a query constructor, the index reference is implicit.

Often, an index reference doesn't fully specify the characteristics of an index. For example, if you create a region path query and specify no options, you've only supplied the type of index (geospatial region path index) and the path. You have not explicitly specified the coordinate system, precision, or point type. Thus, they implicitly default to "wgs84", single, and "point", respectively.

MarkLogic attempts to resolve an index reference from the information in the call, including options, plus the defaults. If this is sufficient to identify a unique index, that index will be used. If it is not, an error is raised.

For example, suppose you create a geospatial region index on the path `/coordinates`, with coordinate-system and precision "wgs84/double". If you then construct a region query on the path `/coordinates` and specify the option "coordinate-system=wgs84", the precision is implicitly single precision, which will not match the only available index. You will get a `XDMP-GIDXNOTFOUND` error.

Similarly, suppose you create one geospatial region index on the path `/coordinates`, with coordinate-system and precision "wgs84/double" and another on the same path with "wgs84" (single precision). If you then create a region path query on /coordinates and do not specify the coordinate system, the index reference is ambiguous and you will get a `XDMP-GIDXAMBIGUOUS` error.

14.7 Searching for Matching Points

This section describes how to use a point query to find documents containing specific points or documents containing points in specific regions. You should use a point query rather than a region query when searching for points because point queries are usually faster than region queries.

This section covers the following topics:

- Point Search Overview
- Example: Point Query Using XQuery
- Example: Point Query Using JavaScript
- Constructing a Point Query in XQuery
- Constructing a Point Query in JavaScript
- Constructing a Point Query from Query Text
- Creating Point Queries with the Client APIs
- Creating Geospatial Facets

14.7.1 Point Search Overview

A point query finds documents containing one or more points that match search criteria regions. The search criteria regions can be points, circles, linestrings, polygons, or any other cts region type. (To find matching regions, rather than points, see "Searching for Matching Regions" on page 528).

The following are key features of searching with point queries:

- A point matches a criteria region if it is contained in the region.
- You can use options to control whether or not the criteria region boundaries should be considered in the match. Boundaries are included by default.
- You can use point queries with the same search framework as other kinds of queries, such as `cts:search`, `cts.search`, `jsearch.documents`, `search:search`, or the Client APIs.
- You can use a point query by itself or as a component of a more complex query, such as a `cts:and-query` (XQuery) or `cts.andQuery` (JavaScript).
- You can construct a geospatial point query using an XQuery or JavaScript query constructor, by parsing query text, or using the REST, Java, or Node.js Client APIs.
- Creating appropriate geospatial point indexes can improve speed and accuracy.

  Indexes are required for certain kinds of queries, such as range queries. Indexes are optional for queries such as value queries, but only if you use unfiltered search. For details, see Fast Pagination and Unfiltered Searches in the *Query Performance and Tuning Guide*.

For example, the following search uses an element child geospatial query to match documents containing at least one point in the circle with center (37.5073428,-122.2465038) and radius 1 mile. The circle criteria region is constructued using the `cts:circle` XQuery function or `cts.circle` JavaScript function.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`cts:search(fn:collection("geo-example"),`
	`  cts:element-child-geospatial-query(`
	`  fn:QName("http://www.opengis.net/kml/2.2", "Point"),`
	`  fn:QName("http://www.opengis.net/kml/2.2", "coordinates"),`
	`  cts:circle(1, cts:point(37.5073428,-122.2465038)),`
	`  ("type=long-lat-point")`
	`  )`
	`)//*:name/fn:data()`
	
	`(: returns just the names of the matched places :)`

JavaScript	`import * as jsearch from '/MarkLogic/jsearch.mjs';`
	`const geoSamples = jsearch.collections('geo-example');`
	`const matchedPlaces = [];`
	
	`geoSamples.documents()`
	`  .where(cts.elementChildGeospatialQuery(`
	`  fn.QName('http://www.opengis.net/kml/2.2', 'Point'),`
	`  fn.QName('http://www.opengis.net/kml/2.2', 'coordinates'),`
	`  cts.circle(1, cts.point(37.5073428,-122.2465038)),`
	`  ['type=long-lat-point']`
	`  ))`
	`  .result().results.forEach(function(result) {`
	`  // extract just the names of the matched places`
	`  matchedPlaces.push(`
	`  result.document.xpath('//*:name/fn:data()'))`
	`  });`
	`matchedPlaces`

(The above queries were written for sample documents containing KML geospatial data, so an element child query is used to confine matches to coordinates in KML `<Point/>` elements. The long-lat point type is used because KML coordinates are expressed in longitude-first order.)

The MarkLogic APIs also include geospatial utility functions useful for constructing criteria and analyzing search matches. For example, you can use the `geo:region-contains` XQuery function or the `geo.regionContains` JavaScript function to test whether one region contains another. The utility functions are usable with in-memory geospatial data, as well as data in documents in the database. For details, see “Summary of Other Geospatial Operations” on page 560.

14.7.2 Example: Point Query Using XQuery

This example uses XQuery to demonstrate the following type of point queries:

- Find documents containing this point
- Find documents containing points in this region

For an equivalent Server-Side JavaScript example, see "Example: Point Query Using JavaScript" on page 514. The example assumes the data and database configuration from "Preparing to Run the Examples" on page 569.

The sample data is XML documents containing KML data of the following form. For more details on the sample documents, see "Overview of the Sample Data" on page 569.

```
<envelope>
  <Placemark xmlns="http://www.opengis.net/kml/2.2">
    <name>MarkLogic HQ</name>
    <Point>
      <coordinates>-122.2465038,37.5073428</coordinates>
    </Point>
  </Placemark>
</envelope>
```

The example uses `cts:element-child-geospatial-query` to find matches in `coordinates` element of a KML `Point` element. Limiting the scope to coordinates in a `Point` element prevents false positives from the documents containing other kinds of regions. For example:

```
cts:element-child-geospatial-query(
      fn:QName("http://www.opengis.net/kml/2.2", "Point"),
      fn:QName("http://www.opengis.net/kml/2.2", "coordinates"),
      cts:point(37.5073428, -122.2465038),
      ("type=long-lat-point")
```

The query includes the "type=long-lat-point" option because KML uses longitude-first coordinate order while the default in MarkLogic is latitude-first (`"type=point"`).

The database configuration includes a corresponding geospatial element child index on `kml:Point/kml:coordinates` with long-lat point type.

The following code performs one search for documents containing the coordinates of the MarkLogic headquarters (`cts:point(37.5073428, -122.2465038)`) and one search for documents containing points in the "MarkLogic Neighborhood" polygon. The polygon coordinates are extracted from one of the sample documents, but you could also construct them inline.

```
xquery version "1.0-ml";

(: Find docs containing a point that matches another point.
 : The criteria point corresponds to the MarkLogic HQ feature :)
```

```
let $point-matches :=
  cts:search(fn:collection("geo-xml-examples"),
    cts:element-child-geospatial-query(
      fn:QName("http://www.opengis.net/kml/2.2", "Point"),
      fn:QName("http://www.opengis.net/kml/2.2", "coordinates"),
      cts:point(37.5073428, -122.2465038),
      ("type=long-lat-point")
    )
  )
(: Find docs containing a point contained in a region. The
 : MarkLogic Neighborhood polygon is used as the criteria region. :)
let $region-matches :=
  cts:search(fn:collection("geo-xml-examples"),
    cts:element-child-geospatial-query(
      fn:QName("http://www.opengis.net/kml/2.2", "Point"),
      fn:QName("http://www.opengis.net/kml/2.2", "coordinates"),
      fn:doc("/geo-examples/MarkLogic-Neighborhood.xml")//cts-region,
      ("type=long-lat-point")
    )
  )
(: Format results for display in QC :)
return (
  fn:concat("Features containing the cirteria point: ",
            fn:string-join($point-matches//*:name/data(), ", ")),
  fn:concat("Features containing points in the criteria region: ",
            fn:string-join($region-matches//*:name/data(), ", "))
)
```

If you run this query in Query Console, it produces output similar to the following:

```
Features containing the cirteria point: MarkLogic HQ
Features containing points in the criteria region:
   Restaurant, Museum, MarkLogic HQ
```

You can compose complex queries by combining geospatial queries with other query types. For example, the following code matches documents that contains points within a circle and that also contain the word “MarkLogic”:

```
cts:search(fn:collection("geo-xml-examples"),
  cts:and-query((
    cts:word-query("MarkLogic"),
    cts:element-child-geospatial-query(
      fn:QName("http://www.opengis.net/kml/2.2", "Point"),
      fn:QName("http://www.opengis.net/kml/2.2", "coordinates"),
      cts:circle(1, cts:point(37.5073428,-122.2465038)),
      ("type=long-lat-point")
    )
  ))
)//*:name/data()
```

Though the previous examples only searched the XML sample documents, you can apply a geospatial query to either XML or JSON documents, or both. For example, the following code searches both the XML and JSON sample documents by combining two geospatial queries in an OR query. (The point search criteria matches the "MarkLogic HQ" feature in the sample documents.)

```
let $matches :=
  cts:search(fn:collection("geo-examples"),
    cts:or-query((
      cts:path-geospatial-query(
        'geometry[type = "Point"]/array-node("coordinates")',
        cts:point(37.5073428, -122.2465038),
        ('type=long-lat-point')
      ),
      cts:element-child-geospatial-query(
        fn:QName('http://www.opengis.net/kml/2.2', 'Point'),
        fn:QName('http://www.opengis.net/kml/2.2', 'coordinates'),
        cts:point(37.5073428, -122.2465038),
        ('type=long-lat-point')
      )
    ))
  )
return
  for $match in ($matches)
  return xdmp:node-uri($match)
```

Running this query in Query Console produces the following output:

```
/geo-examples/MarkLogic-HQ.json
/geo-examples/MarkLogic-HQ.xml
```

14.7.3 Example: Point Query Using JavaScript

This example uses Server-Side JavaScript to demonstrate the following type of point queries:

- Find documents containing this point
- Find documents containing points in this region

For an equivalent XQuery example, see "Example: Point Query Using XQuery" on page 512. This example assumes the data and database configuration from "Preparing to Run the Examples" on page 569.

The sample data includes JSON documents containing GeoJSON data of the following form. For more details on the sample documents, see "Overview of the Sample Data" on page 569.

```
{ "envelope": {
  "feature": {
    "type": "Feature",
    "geometry": {
      "type": "Point",
```

```
      "coordinates": [ -122.2465038, 37.5073428 ]
    },
    "properties": { "name": "MarkLogic HQ" }
} } }
```

The example uses `cts.pathGeospatialQuery` to find matching documents. You must use a path query for point queries on GeoJSON for the reasons described in Geospatial Data in the *Application Developer's Guide*. The following path addresses the `coordinates` array of a point feature in the sample documents:

```
geometry[type = "Point"]/array-node("coordinates")
```

Thus, the core of the search is a path query of the following form. The query includes the "type=long-lat-point" option because GeoJSON uses longitude-first coordinate order while the default in MarkLogic is latitude-first (`"type=point"`).

```
cts.pathGeospatialQuery(
  'geometry[type = "Point"]/array-node("coordinates")',
  cts.circle(0.25, cts.point(37.5073428, -122.2465038)),
  ("type=long-lat-point")
)
```

The database configuration must include a corresponding geospatial path index. The instructions in "Preparing to Run the Examples" on page 569 include creating a suitable index.

The following code uses the JSearch API to perform 2 searches: one search for documents containing a point (`cts.point(37.5073428, -122.2465038)`, and one for documents containing points in a region. The region coordinates are extracted from one of the sample documents for convenience, but you could also construct the region inline using a geospatial constructor such as `cts.polygon`.

```
'use strict';
import * as jsearch from '/MarkLogic/jsearch.mjs';
const geoSamples = jsearch.collections('geo-examples');

// Find docs containing a point that matches another point.
// The criteria point corresponds the MarkLogic HQ feature.
const pointMatches =
  geoSamples.documents().where(
    cts.pathGeospatialQuery(
      'geometry[type = "Point"]/array-node("coordinates")',
      cts.point(37.5073428, -122.2465038),
      ("type=long-lat-point")
    )
  ).map(desc => desc.document.envelope.feature.properties.name)
  .result();
const regionMatches =
  geoSamples.documents().where(
    cts.pathGeospatialQuery(
      'geometry[type = "Point"]/array-node("coordinates")',
      fn.head(fn.doc('/geo-examples/MarkLogic-Neighborhood.json'))
```

```
        .toObject().envelope.ctsRegion,
      ["type=long-lat-point"]
    )
  ).map(desc => desc.document.envelope.feature.properties.name)
  .result();

// Format the results for display.
const results = 'Features containing the cirteria point: '
  + pointMatches.results.join(", ")
  + '\nFeatures containing points in the criteria region: '
  + regionMatches.results.join(', ');
results;
```

If you run this query in Query Console, it produces output similar to the following:

```
Features containing the cirteria point: MarkLogic HQ
Features containing points in the criteria region:
   Restaurant, Museum, MarkLogic HQ
```

Note that a lambda expression and the `map` method are used to extract just the feature names from the matched documents:

```
geoSamples.documents()
    .where(...)
    .map(desc => desc.document.envelope.feature.properties.name)
    .result();
```

This is a contrivance used to keep the example output brief. If you remove the `map` call, the search returns a `Sequence` of document descriptors that include the full document. For more details, see "Creating JavaScript Search Applications" on page 289.

You can also use `cts.search` to perform an equivalent search. For example:

```
// Find docs containing a point that matches another point.
// The criteria point corresponds the MarkLogic HQ feature.
const pointMatches =
  cts.search(
    cts.andQuery([
      cts.collectionQuery('geo-json-examples'),
      cts.pathGeospatialQuery(
        'geometry[type = "Point"]/array-node("coordinates")',
        cts.point(37.5073428, -122.2465038),
        ('type=long-lat-point')
    )
    ])
  );
// Find docs containing points contained in a region. The MarkLogic
// Neighborhood polygon is used as the criteria to be matched.
const regionMatches =
  cts.search(
    cts.andQuery([
      cts.collectionQuery('geo-json-examples'),
```

```
      cts.pathGeospatialQuery(
        'geometry[type = "Point"]/array-node("coordinates")',
        fn.head(fn.doc('/geo-examples/MarkLogic-Neighborhood.json'))
          .toObject().envelope.ctsRegion,
        ['type=long-lat-point']
      )
    ])
  );

// Format the results for display.
const results = 'Features containing the cirteria point: ';
const featureNames = [];
for (let doc of pointMatches) {
  featureNames.push(doc.toObject().envelope.feature.properties.name);
}
results += featureNames.join(', ');

results += '\nFeatures containing points in the criteria region: ';
featureNames = [];
for (let doc of regionMatches) {
  featureNames.push(doc.toObject().envelope.feature.properties.name);
}
results + featureNames.join(', ');
```

You can include multiple criteria in a single query; when you do so, encapsulate the criteria in an array. For example, you could search for matches to both the point and the region with a query such as the following. A document matches if it matches any one of the criteria

```
cts.andQuery([
  cts.collectionQuery('geo-json-examples'),
  cts.pathGeospatialQuery(
    'geometry[type = "Point"]/array-node("coordinates")',
    [cts.point(37.5073428, -122.2465038),
     fn.head(fn.doc('/geo-examples/MarkLogic-Neighborhood.json'))
       .toObject().envelope.ctsRegion],
    ['type=long-lat-point']
) ])
```

Since the MarkLogic HQ feature document satisfies both cirteria and the Museum and Restaurant feature documents satisfy the region criteria, the above query matches the Restaurant, Museum, and MarkLogic HQ features.

You can compose complex queries by combining geospatial queries with other query types. Notice that the above `cts.search` example search uses a `cts.andQuery` to combine the geospatial path query with a collection query that constrains the search to the JSON documents in the sample set.

To include the XML sample documents in the search, add a `cts.elementChildGeospatialQuery` on the KML data. For example, the following query finds documents containing the MarkLogic HQ coordinates in either the XML or JSON sample documents, and prints out the URIs of the matched documents:

```
'use strict';
import * as jsearch from '/MarkLogic/jsearch.mjs';
const geoSamples = jsearch.collections('geo-examples');

geoSamples.documents().where(
  cts.orQuery([
    cts.pathGeospatialQuery(
      'geometry[type = "Point"]/array-node("coordinates")',
      cts.point(37.5073428, -122.2465038),
      ['type=long-lat-point']
    ),
    cts.elementChildGeospatialQuery(
      fn.QName('http://www.opengis.net/kml/2.2', 'Point'),
      fn.QName('http://www.opengis.net/kml/2.2', 'coordinates'),
      cts.point(37.5073428, -122.2465038),
      ['type=long-lat-point']
    )
  ])
).map(desc => desc.uri)
 .result().results;
```

Running the above query in Query Console, produces the following output:

```
["/geo-examples/MarkLogic-HQ.json",
 "/geo-examples/MarkLogic-HQ.xml"]
```

14.7.4 Constructing a Point Query in XQuery

This section is a quick reference of available XQuery geospatial point query constructors. These functions create a cts:query object. For an equivalent JavaScript reference, see "Constructing a Point Query in JavaScript" on page 519. To create geospatial queries from query text, see "Constructing a Point Query from Query Text" on page 520.

Use the following functions to construct a point query. Select the query constructor that corresponds to the type of region and layout of the data to be searched, as described in "Understanding Geospatial Query and Index Types" on page 493. You can use these constructors with each other and with other `cts:query` constructors to build up complex queries.

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

- `cts:json-property-child-geospatial-query`
- `cts:json-property-geospatial-query`
- `cts:json-property-pair-geospatial-query`
- `cts:path-geospatial-query`

Every query constructor includes parameters that identify the content to search, either by path, name, or index reference; and one or more geospatial values to match. For example:

```
cts:element-geospatial-query(
  xs:QName("feature"),                          (: element to search :)
  cts:circle(20, cts:point(37.65, -122.42))     (: criteria :)
)
```

A geospatial query is constrained to the XML elements, XML attributes, and JSON properties identified in the query constructor. To cross multiple formats in a single search, use `cts:or-query` to combine multiple geospatial queries.

For a complete example, see “Example: Point Query Using XQuery” on page 512. For more details about constructing geospatial search criteria, see “Constructing Geospatial Point and Region Values” on page 567 and “Converting To and From Common Geospatial Representations” on page 562.

14.7.5 Constructing a Point Query in JavaScript

This section is a quick reference of available Server-Side JavaScript geospatial point query constructors. These functions create a cts.query object. For an equivalent XQuery reference, see “Constructing a Point Query in XQuery” on page 518. To create geospatial queries from query text, see “Constructing a Point Query from Query Text” on page 520.

The following JavaScript geospatial query constructors are available. You can use these constructors with each other and with other `cts:query` constructors to build up complex queries. Select the query constructor that corresponds to the type of region and layout of the data to be searched, as described in “Understanding Geospatial Query and Index Types” on page 493.

- `cts.elementAttributePairGeospatialQuery`
- `cts.elementChildGeospatialQuery`
- `cts.elementGeospatialQuery`
- `cts.elementPairGeospatialQuery`
- `cts.jsonPropertyChildGeospatialQuery`
- `cts.jsonPropertyGeospatialQuery`
- `cts.jsonPropertyPairGeospatialQuery`
- `cts.pathGeospatialQuery`

Every query constructor includes parameters that identify the content to search, either by path, name, or index reference; and one or more geospatial values to match. For example:

```
cts.elementGeospatialQuery(
  "feature",                                  // element to search
  cts.circle(20, cts.point(37.65, -122.42))   // criteria
)
```

A geospatial query is constrained to the XML elements, XML attributes, and JSON properties identified in the query constructor. To cross multiple formats in a single search `cts.orQuery` to combine multiple geospatial queries.

For a complete example, see “Example: Point Query Using XQuery” on page 512. For more details about constructing geospatial search criteria, see “Constructing Geospatial Point and Region Values” on page 567.

14.7.6 Constructing a Point Query from Query Text

You can use the `cts:parse` XQuery function or the `cts.parse` JavaScript function to create a geospatial point query from query text. The parse creates a cts query object. This grammar is only supported by the cts parser; the grammar used by `search:search` or `search:resolve` does not support geospatial terms.

The cts parse grammar supports search terms expressing points, circles, boxes, polygons, and other regions, bound a geospatial index reference. For details, see “Binding to a Geospatial Index Reference” on page 269.

The following example queries create a geospatial element child query over KML point coordinates. The bindings define the interpretation of the "poi" (point of interest) tag as a reference to a geospatial element child index. The query text "@1 -122.2465038,37.5073428" represents a circle with radius 1 mile (the default units) and center (37.5073428, -122.2465038). The query includes the option "type=long-lat-point" because KML uses longitude-first ordering for points, while the MarkLogic default ordering is latitude-first.

Language	Example

XML	`xquery version "1.0-ml";`
	`let $bindings := map:map()`
	`let $_ := map:put(`
	`  $bindings, "poi",`
	`  cts:geospatial-element-child-reference(`
	`  fn:QName("http://www.opengis.net/kml/2.2", "Point"),`
	`  fn:QName("http://www.opengis.net/kml/2.2", "coordinates"),`
	`  ("type=long-lat-point")`
	`  ) )`
	`return`
	`  cts:parse('poi:"@1 -122.2465038,37.5073428"', $bindings)`

JavaScript	`const bindings = {`
	`  'poi': cts.geospatialElementChildReference(`
	`  fn.QName("http://www.opengis.net/kml/2.2", "Point"),`
	`  fn.QName("http://www.opengis.net/kml/2.2",`
	`  "coordinates"),`
	`  ["type=long-lat-point"])`
	`};`
	`cts.parse('poi:"@1 -122.2465038,37.5073428"', bindings);`

The parse produces a cts query similar to the following:

Language	Example Output

XQuery	`cts:element-child-geospatial-query(`
	`  fn:QName("http://www.opengis.net/kml/2.2","Point"),`
	`  fn:QName("http://www.opengis.net/kml/2.2","coordinates"),`
	`  cts:circle("@1 -122.2465,37.507343"),`
	`  ("type=long-lat-point"), 1)`

JavaScript	`cts.elementChildGeospatialQuery(`
	`  [fn.QName("http://www.opengis.net/kml/2.2","Point")],`
	`  [fn.QName("http://www.opengis.net/kml/2.2","coordinates")],`
	`  cts.circle("@1 -122.2465,37.507343"),`
	`  ["type=long-lat-point"], 1)`

For more details, see "Creating a Query From Search Text With cts:parse" on page 253.

14.7.7 Creating Point Queries with the Client APIs

The REST, Java, and Node.js Client APIs expose geospatial queries through the use of structured queries and query builders, rather than through standalone query constructor functions.

The following topics provide examples of using a point query from the Client APIs:

- Java Client API
- Node.js Client API

You can also use a serialized cts query or a structured query with the Node.js, Java, or REST Client APIs and Search API functions such as `search:resolve`. See the following topics for details on including a point query in a structured query:

- “geo-elem-query” on page 130
- “geo-elem-pair-query” on page 134
- “geo-attr-pair-query” on page 138
- “geo-path-query” on page 142
- “geo-json-property-query” on page 146
- “geo-json-property-pair-query” on page 150

14.7.7.1 Java Client API

This topic assumes you are already familiar with the search features of the Java Client API. If you are not, see the *Java Application Developer's Guide*.

You are most likely to construct a geospatial point query with the Java Client API using a StructuredQueryBuilder object. You could also embed a structured point query in a `RawCombinedQuery`; this technique is not covered here. You cannot create a geospatial point query in Java using query text or QBE.

Each geospatial point query can only reference a single point index. To search more than one index, construct multiple point queries and combine them with an OR query.

Use `StructuredQueryDefinition.geospatial` to create a point query. Choose an overload that accepts a `GeospatialIndex` as input. A `GeospatialIndex` object identifies the point index to be searched.

To construct a `GeospatialIndex` object, use one of the geospatial index builders of StructuredQueryBuilder, such as `StructuredQueryBuilder.geoElement`. Choose the index builder that matches your index and data layout; for details, see “Understanding Geospatial Query and Index Types” on page 493.

For example, the following code snippet identifies a geospatial element child point index corresponding to the KML Point features in the data from “Preparing to Run the Examples” on page 569.

```
DatabaseClient client = ...;
QueryManager qm = client.newQueryManager();
StructuredQueryBuilder sqb = qm.newStructuredQueryBuilder();
...
sqb.geoElement(
  sqb.element(new QName("http://www.opengis.net/kml/2.2", "Point")),
  sqb.element(
    new QName("http://www.opengis.net/kml/2.2", "coordinates"))),
...
```

The following example uses the Java Client API to find XML documents containing the feature named “MarkLogic HQ”.

```
package examples;

import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.io.SearchHandle;
import com.marklogic.client.query.MatchDocumentSummary;
import com.marklogic.client.query.QueryManager;
import com.marklogic.client.query.StructuredQueryBuilder;
import
com.marklogic.client.query.StructuredQueryBuilder.FragmentScope;
import com.marklogic.client.query.StructuredQueryDefinition;

import javax.xml.namespace.QName;


public class GeoPointQuery {
    public static void main(String[] args) {
public static void main(String[] args) {
    // MODIFY THIS CALL TO MATCH YOUR ENV
    DatabaseClient client = DatabaseClientFactory.newClient(
        hostname, port, databaseName,
        new DatabaseClientFactory.DigestAuthContext(
            username, password));

    QueryManager qm = client.newQueryManager();
    StructuredQueryBuilder sqb = qm.newStructuredQueryBuilder();
    SearchHandle results = new SearchHandle();
    StructuredQueryDefinition query = sqb.geospatial(
        sqb.geoElement(
            sqb.element(new QName("http://www.opengis.net/kml/2.2",
                        "Point")),
            sqb.element(new QName("http://www.opengis.net/kml/2.2",
                        "coordinates"))),
        FragmentScope.DOCUMENTS,
        new String[] {"type=long-lat-point"},
        sqb.circle(37.507, -122.246, 0.25));
```

```
        qm.search(query, results);
        for (MatchDocumentSummary match : results.getMatchResults()) {
            System.out.println(match.getUri());
        }

        client.release();
    }
    }
```

If you run the above program against the sample data and database configuration from "Preparing to Run the Examples" on page 569, you should see output similar to the following:

```
/geo-examples/MarkLogic-HQ.xml
```

The example as written will only match the XML sample documents from "Preparing to Run the Examples" on page 569. You can match the JSON sample documents by changing the index builder to use `StructuredQueryBuilder.geoPath` and the path `geometry[type = "Point"]/array-node("coordinates")`, similar to the example in "Example: Point Query Using JavaScript" on page 514.

For more details, see Searching in the *Java Application Developer's Guide*.

14.7.7.2 Node.js Client API

This topic assumes you are familiar with the search features of the Node.js Client API. If you are not, you should review the *Node.js Application Developer's Guide*.

To construct a geospatial point query, use queryBuilder.geospatial. Use one of the geospatial index reference builders such as `queryBuilder.geoPath` or `queryBuilder.geoElement` to construct the point index specification. Choose the builder that corresponds to your index and data layout, as described in "Understanding Geospatial Query and Index Types" on page 493. Use helper functions such as `queryBuilder.point` to construct the criteria point(s) or regions(s).

Each geospatial point query can only reference a single index. To search more than one index, construct multiple region queries and combine them with an OR query.

The following example performs the same search as "Example: Simple Intersection Region Query" on page 530. The example relies on the sample documents and database configuration from "Preparing to Run the Examples" on page 569. Before running the example, modify the connection information in `connInfo`.

```
const marklogic = require('marklogic');

// MODIFY THIS VAR TO MATCH YOUR ENV
const connInfo = {
    host: 'localhost',
    port: 8000,
    user: username,
    password: password,
```

```
    database: 'Documents'
  };
const db = marklogic.createDatabaseClient(connInfo);
const qb = marklogic.queryBuilder;

db.documents.query(
  qb.where(
    qb.geospatial(
      qb.geoElement(
        qb.qname('http://www.opengis.net/kml/2.2', 'Point'),
        qb.qname('http://www.opengis.net/kml/2.2', 'coordinates')),
      qb.fragmentScope('documents'),
      qb.geoOptions('type=long-lat-point'),
      qb.circle(0.25, qb.latlon(37.507, -122.246))
   ))
).result( function(results) {
  for (let result of results) {
    console.log(result.uri);
  }
});
```

If you run the example against the sample data from "Preparing to Run the Examples" on page 569, you should see output similar to the following:

```
/geo-examples/MarkLogic-HQ.xml
```

As written, the sample code will only match the XML documents in the sample data. To match the JSON documents, use queryBuilder.geoPath instead of queryBuilder.geoElement and the path `geometry[type = "Point"]/array-node("coordinates")`, similar to the example in "Example: Point Query Using JavaScript" on page 514.

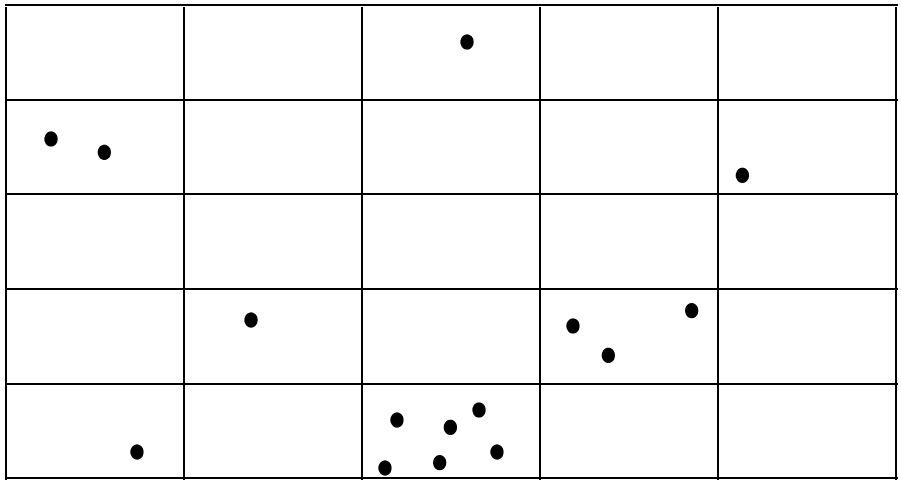
For more details, see Querying Documents and Metadata in the *Node.js Application Developer's Guide*.

14.7.8 Creating Geospatial Facets

Faceted navigation of search results enables users to filter large or complex search results by properties of the data. For example, filter a list of clothing items by size, color, and material. One technique for faceting the results of a point query is to define geospatial boxes that enclose the matched points.

If you divide a geospatial box into a grid of boxes, you can bucket matched points by the sub-divisions. Each subdivision represents a facet.

For example, the following diagram plots a series of matched points on a 5x5 grid.



You can use the number of points in each box and the box extent with mapping APIs like Google Maps to generate a heat map from the search results.

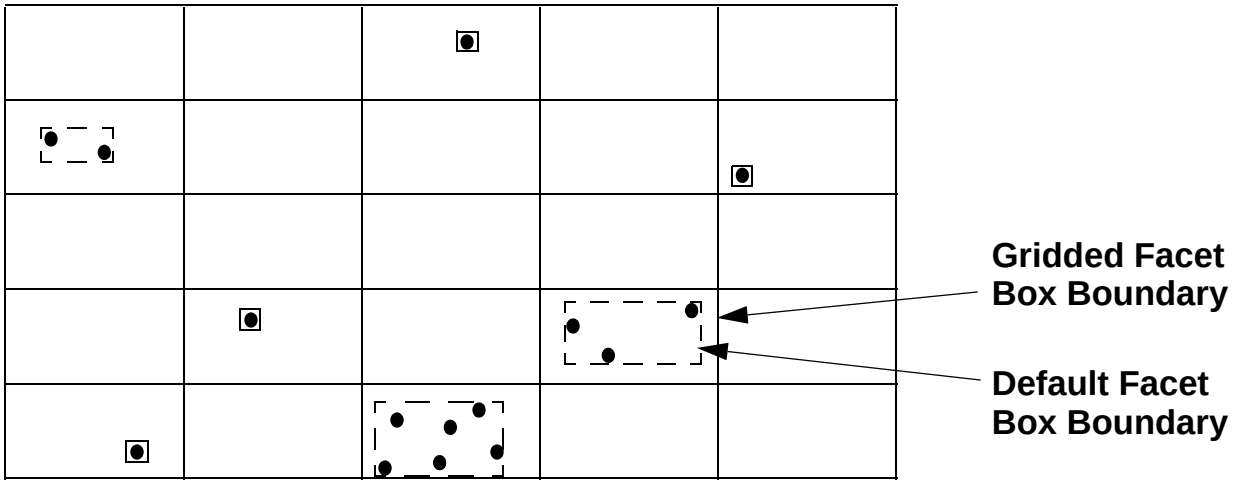
You can generate such geospatial facet data using the `cts:*-geospatial-boxes` XQuery functions and the `cts.*GeospatialBoxes` Server-Side JavaScript functions, such as `cts:element-geospatial-boxes` or `cts.elementGeospatialBoxes`. You can use the `cts:frequency` XQuery function or the `cts.frequency` JavaScript function to compute the number of points in each box.

The XQuery Search API, JavaScript jsearch API, and the Client APIs an equivalent capability at a higher level of abstraction through the heatmap component of a geospatial constraint definition.

All of these interfaces enable you to define the extent of the box over which to generate facets, and the latitudes and longitudes of the subdivisions.

You can also control whether the returned facet box extents are based on an even grid or the minimum bounding box that encompasses the points in a given bucket, as shown in the following diagram. By default, MarkLogic generates the minimum bounding boxes. Use the “gridded” option to override the default.

The following diagram illustrates the difference between gridded and non-gridded faceting:



The following XQuery example generates geospatial facets and their counts `cts:element-pair-geospatial-boxes`:

```
xquery version "1.0-ml";

(: compute even divisions between two coordinates :)
declare function local:compute-divs($coord1, $coord2, $ndivs)
as xs:double*
{
  let $bound-size := abs(($coord1 - $coord2) div $ndivs)
  let $start := if ($coord2 < $coord1) then $coord2 else $coord1
  return
    for $count in (1 to $ndivs)
    return ($start + ($count - 1) * $bound-size)
};

(: box dimensions :)
let $n := 49.0
let $s := 24.0
let $e := -67.0
let $w := -125.0

(: number of latitude and longitude divisions for facet bucketing :)
let $n-lat-divs := 5
let $n-lon-divs := 5

(: query with which to constrain points to facet :)
let $query := cts:and-query((
  cts:element-range-query(xs:QName("magnitude"), ">", 7),
  cts:element-pair-geospatial-query(
      xs:QName("quake"), xs:QName("lat"), xs:QName("long"),
      cts:box($s, $w, $n, $e))
))
```

```
(: facet the matching points :)
let $boxes := cts:element-pair-geospatial-boxes(
  xs:QName("quake"), xs:QName("lat"), xs:QName("long"),
  local:compute-divs($n, $s, $n-lat-divs),
  local:compute-divs($e, $w, $n-lon-divs),
  "gridded",
  $query)

(: count the points in each facet :)
let $counts := cts:frequency($boxes)

return for $i in (1 to fn:count($boxes))
  return (fn:subsequence($boxes, $i, 1),
          fn:subsequence($counts, $i, 1))
```

For the Server-Side JavaScript jsearch API, use `jsearch.makeHeatMap` with `FacetDefinition.groupInto` to define geospatial facets. For details, see "Grouping Values and Facets Into Buckets" on page 367.

For the XQuery SearchAPI and the client APIs, use the heatmap component of a geospatial constraint query option. In these interfaces, you need only specify the faceting box extent and the number of latitude and longitude divisions in the heatmap component. The underlying API computes the grid, the facet boxes, and the counts for you. For example:

```
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="geo">
    <geo-elem-pair>
      <parent ns="" name="quake"/>
      <lat ns="" name="lat"/>
      <lon ns="" name="long"/>
      <heatmap s="24.0" n="49.0" e="-67.0" w="-125.0"
               latdivs="5" londivs="5" />
      <facet-option>gridded</facet-option>
    </geo-elem-pair>
  </constraint>
  <return-facets>true</return-facets>
</options>
```

For a more complete example, see "Geospatial Constraint Example" on page 393.

14.8 Searching for Matching Regions

This section describes how to use a region query to search for regions in your documents. You can match regions using topological operators such as as containment and intersection.

This section covers the following topics:

- Region Match Overview
- Example: Simple Intersection Region Query

- Example: Using Region Queries in a Composed Query
- Constructing a Region Query Using a Constructor
- Constructing a Region Query from Query Text
- Creating Region Queries Using the Client APIs
- Example: Using the Envelope Pattern to Encode Regions

14.8.1 Region Match Overview

A region query matches geospatial regions in your documents against one or more criteria regions. The relationship between the regions that must be satisfied for a "match" is determined by the operator configured into the query. For example, you can create a query that matches documents containing a region that overlaps or intersects your criteria region(s).

> **Note:** MarkLogic Server does not support region queries using multi-part (WKT/WKB) geometries. Multi-part geometries (MULTI*) include MULTIPOINT, MULTIPOLYGON, and so on. For a list of WKT/WKB geometries, see Mapping of WKT and WKB Types to MarkLogic Types.

To construct a region query, use the XQuery `cts:geospatial-region-query` function or the JavaScript `cts.geospatialRegionQuery` function. Region queries require a geospatial region path index.

For example, the following code snippet creates a query the matches documents containing a region the intersects a circle. For a complete example, see "Example: Simple Intersection Region Query" on page 530.

Language	Example

XQuery	`cts:geospatial-region-query(`
	`  cts:geospatial-region-path-reference("/envelope/cts-region"),`
	`  "intersects",`
	`  cts:circle(0.25, cts:point(37.507343,-122.2465))`
	`)`

JavaScript	`cts.geospatialRegionQuery(`
	`  cts.geospatialRegionPathReference('/envelope/ctsRegion'),`
	`  'intersects',`
	`  cts.circle(0.25, cts.point(37.507343,-122.2465))`
	`)`

The following are key points about using region queries:

- One region matches another if it satisfies the topological operator configured into the query. You can choose from the following operators: contains, covered-by, covers, crosses, disjoint, equals, intersects, overlaps, within. For details, see `cts:geospatial-region-query` in the *MarkLogic XQuery and XSLT Function Reference* or `cts.geospatialRegionQuery` in the *MarkLogic Server-Side JavaScript Function Reference*.
- You must define a geospatial region path index on any regions you want to search with a region query. For details, see "Geospatial Region Queries and Indexes" on page 506.
- The regions in your documents must be in WKT or serialized cts:region format. If your data is not in one of these formats, you must transform it to conform. For one possible solution, see "Example: Using the Envelope Pattern to Encode Regions" on page 548.
- You can use a region query with the same search framework as other kinds of queries, such as `cts:search`, `cts.search`, `jsearch.documents`, `search:search`, or the Client APIs.
- You can use a region query by itself or as a component of a more complex query, such as a `cts:and-query` (XQuery) or `cts.andQuery` (JavaScript).
- You can construct a geospatial region query using an XQuery or JavaScript query constructor, by parsing query text, or using the REST, Java, or Node.js Client APIs.
- For best performance, when matching against individual points in your documents, you should usually use a point query rather than a region query.
- For highest accuracy when using a region query to match regions in documents, include only one region per document.

The MarkLogic APIs also include geospatial utility functions useful for constructing criteria and analyzing search matches. For example, you can use the `cts:region-contains` XQuery function or the `cts.region-contains` JavaScript function to test whether one region contains another. The utility functions are usable with in-memory geospatial data, as well as data in documents in the database. For details, see "Summary of Other Geospatial Operations" on page 560.

14.8.2 Example: Simple Intersection Region Query

This example depends on the data and configuration in "Preparing to Run the Examples" on page 569.

Run one of the following queries in Query Console to find documents that contain a region that intersects with a polygon. The criteria polygon corresponds to the "MarkLogic Neighborhood" region in the sample data. The query produces the feature names from the matched documents.

Language	Example

XQuery	```xquery version "1.0-ml";
	declare namespace kml="http://www.opengis.net/kml/2.2";
	
	(: This region corresponds to the MarkLogic Neighborhood polygon :)
	let $criteria-region := cts:polygon((
	  cts:point(37.519087, -122.26346),
	  cts:point(37.521299, -122.24805),
	  cts:point(37.512279, -122.24462),
	  cts:point(37.50336, -122.24556),
	  cts:point(37.506185, -122.25981),
	  cts:point(37.513436, -122.26337),
	  cts:point(37.519087, -122.26346)
	))
	return cts:search(
	  fn:collection("geo-xml-examples"),
	  cts:geospatial-region-query(
	  cts:geospatial-region-path-reference("/envelope/cts-region"),
	  "intersects", $criteria-region
	  )
	)/envelope/kml:Placemark/kml:name/fn:data()```

Language	Example

JavaScript	`// This region corresponds to the MarkLogic Neighborhood polygon`
	`const criteriaRegion = cts.polygon([`
	`  cts.point(37.519087, -122.26346),`
	`  cts.point(37.521299, -122.24805),`
	`  cts.point(37.512279, -122.24462),`
	`  cts.point(37.50336,  -122.24556),`
	`  cts.point(37.506185, -122.25981),`
	`  cts.point(37.513436, -122.26337),`
	`  cts.point(37.519087, -122.26346)`
	`])`
	
	`// Perform the search`
	`const results = cts.search(cts.andQuery([`
	`  cts.collectionQuery('geo-json-examples'),`
	`    cts.geospatialRegionQuery(`
	`    cts.geospatialRegionPathReference('/envelope/ctsRegion'),`
	`    'intersects', criteriaRegion`
	`  )`
	`]))`
	
	`// Iterate over the results, accumulating the feature names in`
	`// an array for convenient display in Query Console.`
	`const matchedRegions = [];`
	`for (let result of results) {`
	`  matchedRegions.push(`
	`    result.toObject().envelope.feature.properties.name)`
	`}`
	`matchedRegions`

If you run one of these queries in Query Console, the following feature names should be displayed:

- Holly St
- Wildlife Refuge
- Hwy 101
- Airport
- MarkLogic Neighborhood

You can also work with XML documents in JavaScript and work with JSON documents in XQuery, as shown below. The following example performs the same region search against the "opposite" document type.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`declare namespace kml="http://www.opengis.net/kml/2.2";`
	
	`(: This region corresponds to the MarkLogic Neighborhood polygon :)`
	`let $criteria-region := cts:polygon((`
	`  cts:point(37.519087, -122.26346),`
	`  cts:point(37.521299, -122.24805),`
	`  cts:point(37.512279, -122.24462),`
	`  cts:point(37.50336,  -122.24556),`
	`  cts:point(37.506185, -122.25981),`
	`  cts:point(37.513436, -122.26337),`
	`  cts:point(37.519087, -122.26346)`
	`))`
	`return cts:search(`
	`  fn:collection("geo-json-examples"),`
	`  cts:geospatial-region-query(`
	`  cts:geospatial-region-path-reference("/envelope/ctsRegion"),`
	`  "intersects", $criteria-region`
	`  )`
	`)/envelope/feature/properties/name`

Language	Example

JavaScript	`// This region corresponds to the MarkLogic Neighborhood polygon`
	`const criteriaRegion = cts.polygon([`
	`  cts.point(37.519087, -122.26346),`
	`  cts.point(37.521299, -122.24805),`
	`  cts.point(37.512279, -122.24462),`
	`  cts.point(37.50336,  -122.24556),`
	`  cts.point(37.506185, -122.25981),`
	`  cts.point(37.513436, -122.26337),`
	`  cts.point(37.519087, -122.26346)`
	`])`
	
	`// Perform the search`
	`const results = cts.search(cts.andQuery([`
	`  cts.collectionQuery('geo-xml-examples'),`
	`    cts.geospatialRegionQuery(`
	`    cts.geospatialRegionPathReference('/envelope/cts-region'),`
	`    'intersects', criteriaRegion`
	`  )`
	`]))`
	
	`// Iterate over the results, accumulating the feature names in`
	`// an array for convenient display in Query Console.`
	`const matchedRegions = [];`
	`for (let result of results) {`
	`  matchedRegions.push(fn.head(result.xpath(`
	`    '/envelope/kml:Placemark/kml:name/fn:data()',`
	`    {kml: 'http://www.opengis.net/kml/2.2'})))`
	`}`
	`matchedRegions`

Notice that the result processing in JavaScript is significantly different because you cannot handle the matched XML documents as native javascript objects.

The following example searches the combined set of both XML and JSON documents. Notice that you can pass multiple region index references into the geospatial region query constructor. A document satisfies the query if a match is found using any of the indexes.

Language	Example

XQuery	`xquery version "1.0-ml";`
	
	`(: This region corresponds to the MarkLogic Neighborhood polygon :)`
	`let $criteria-region := cts:polygon((`
	`  cts:point(37.519087, -122.26346),`
	`  cts:point(37.521299, -122.24805),`
	`  cts:point(37.512279, -122.24462),`
	`  cts:point(37.50336,  -122.24556),`
	`  cts:point(37.506185, -122.25981),`
	`  cts:point(37.513436, -122.26337),`
	`  cts:point(37.519087, -122.26346)`
	`))`
	`let $matches := cts:search(`
	`  fn:collection("geo-examples"),`
	`  cts:geospatial-region-query(`
	`    (cts:geospatial-region-path-reference("/envelope/cts-region"),`
	`     cts:geospatial-region-path-reference("/envelope/ctsRegion")),`
	`    "intersects", $criteria-region`
	`  )`
	`)`
	`for $doc in $matches order by xdmp:node-uri($doc)`
	`return xdmp:node-uri($doc)`

Language	Example

JavaScript	`// This region corresponds to the MarkLogic Neighborhood polygon`
	`const criteriaRegion = cts.polygon([`
	`  cts.point(37.519087, -122.26346),`
	`  cts.point(37.521299, -122.24805),`
	`  cts.point(37.512279, -122.24462),`
	`  cts.point(37.50336,  -122.24556),`
	`  cts.point(37.506185, -122.25981),`
	`  cts.point(37.513436, -122.26337),`
	`  cts.point(37.519087, -122.26346)`
	`]);`
	
	`// Perform the search`
	`const results = cts.search(cts.andQuery([`
	`  cts.collectionQuery('geo-examples'),`
	`  cts.geospatialRegionQuery(`
	`  [cts.geospatialRegionPathReference('/envelope/cts-region'),`
	`  cts.geospatialRegionPathReference('/envelope/ctsRegion')],`
	`  'intersects', criteriaRegion`
	`  )`
	`]));`
	
	`// Accumulate the matched URIs in an array for`
	`// convenient display of brief results in Query Console.`
	`const matchedRegions = [];`
	`for (let result of results) {`
	`  matchedRegions.push(xdmp.nodeUri(result))`
	`}`
	`matchedRegions.sort()`

If you run one of these queries in Query Console, it emits the following list of URIs:

```
/geo-examples/Airport.json
/geo-examples/Airport.xml
/geo-examples/Holly-St.json
/geo-examples/Holly-St.xml
/geo-examples/Hwy-101.json
/geo-examples/Hwy-101.xml
/geo-examples/MarkLogic-Neighborhood.json
/geo-examples/MarkLogic-Neighborhood.xml
/geo-examples/Wildlife-Refuge.json
/geo-examples/Wildlife-Refuge.xml
```

14.8.3 Example: Using Region Queries in a Composed Query

This example depends on the data and configuration in “Preparing to Run the Examples” on page 569.

You can use geospatial region queries along with other query types to compose more complex queries. For example, the following queries find documents containing a region that intersects with the “MarkLogic Neighborhood” region, but that do not contain a region that is covered by the “MarkLogic Neighborhood” region.

Language	Example

XQuery	```xquery version "1.0-ml";
	declare namespace kml="http://www.opengis.net/kml/2.2";
	
	(: This region corresponds to the MarkLogic Neighborhood polygon :)
	let $criteria-region := cts:polygon((
	  cts:point(37.519087, -122.26346),
	  cts:point(37.521299, -122.24805),
	  cts:point(37.512279, -122.24462),
	  cts:point(37.50336, -122.24556),
	  cts:point(37.506185, -122.25981),
	  cts:point(37.513436, -122.26337),
	  cts:point(37.519087, -122.26346)
	))
	return cts:search(
	  fn:collection("geo-example"),
	  cts:and-not-query(
	  cts:geospatial-region-query(
	  cts:geospatial-region-path-reference("/envelope/cts-region"),
	  "intersects", $criteria-region
	  ),
	  cts:geospatial-region-query(
	  cts:geospatial-region-path-reference("/envelope/cts-region"),
	  "covered-by", $criteria-region
	  )
	  )
	)/envelope/kml:Placemark/kml:name/fn:data()```

Language	Example

JavaScript	`const criteriaRegion = cts.polygon([`
	`  cts.point(37.519087, -122.26346),`
	`  cts.point(37.521299, -122.24805),`
	`  cts.point(37.512279, -122.24462),`
	`  cts.point(37.50336,  -122.24556),`
	`  cts.point(37.506185, -122.25981),`
	`  cts.point(37.513436, -122.26337),`
	`  cts.point(37.519087, -122.26346)`
	`]);`
	
	`// Perform the search`
	`const results = cts.search(cts.andQuery([`
	`  cts.collectionQuery('geo-example'),`
	`  cts.andNotQuery(`
	`  cts.geospatialRegionQuery(`
	`  cts.geospatialRegionPathReference('/envelope/cts-region'),`
	`  'intersects', criteriaRegion`
	`  ),`
	`  cts.geospatialRegionQuery(`
	`  cts.geospatialRegionPathReference('/envelope/cts-region'),`
	`  'covered-by', criteriaRegion`
	`  )`
	`  )`
	`]));`
	
	`// Iterate over the results. We accumulate the feature names in`
	`// an array just for convenient display in Query Console.`
	`const matchedRegions = [];`
	`for (let result of results) {`
	`  matchedRegions.push(fn.head(result.xpath(`
	`  '/envelope/kml:Placemark/kml:name/fn:data()',`
	`  {kml: 'http://www.opengis.net/kml/2.2'})))`
	`}`
	`matchedRegions`

Query Console displays the following feature names if the query is successful:

```
Wildlife Refuge
Hwy 101
```

14.8.4 Constructing a Region Query Using a Constructor

This section demonstrates how to use the constructor functions `cts:geospatial-region-query` (XQuery) or `cts.geospatialRegionQuery` (JavaScript) to construct a region query on MarkLogic Server. You can also construct a region query from query text using `cts:parse` (XQuery) or `cts.parse` (JavaScript); for details, see "Constructing a Region Query from Query Text" on page 540.

A region query has the following form:

XQuery	JavaScript

`cts:geospatial-region-query(`
`  $region-index-references,`
`  $operator,`
`  $criteria-regions,`
`  $options,`
`  $weight)`
	`cts.geospatialRegionQuery(`
	`  [regionIndexReference, ...],`
	`  operator,`
	`  [criteriaRegion, ...],`
	`  [option, ...],`
	`  weight)`

The options and weight parameters are optional. You can specify multiple region indexes and multiple criteria regions, which is treated as an implicit OR query. That is, a document matches if it satisfies any of the comparisons.

> **Note:** A region query must be backed by a corresponding geospatial region path index. For more details, see "Geospatial Region Queries and Indexes" on page 506.

For example, the following constructor creates a region query that matches region data located at the XPath "/envelope/cts-region" the overlap with a circle with center (-122.2465,37.507343) and radius 1 mile.

Language	Example Output

XQuery	`cts:geospatial-region-query(`
	`  cts:geospatial-region-path-reference("/envelope/cts-region"),`
	`  "overlaps",`
	`  cts:circle("@1 -122.2465,37.507343"))`

JavaScript	`cts.geospatialRegionQuery(`
	`  [cts.geospatialRegionPathReference("/envelope/cts-region")],`
	`  "overlaps",`
	`  cts.circle("@1 -122.2465,37.507343"))`

The operator must be a string with one of the following values, corresponding to the DE9-IM predicates.

- contains
- covered-by
- covers
- crosses
- disjoint
- equals

- intersects
- overlaps
- touches
- within

You can construct your criteria regions using region constructors such as the `cts:polygon` (XQuery) or `cts.polygon` (JavaScript), the geospatial format conversion functions such as `geogml:linestring` (XQuery) or `geojson.circle` (JavaScript), or using WKT or WKB.

A geospatial query is constrained to the XML elements, XML attributes, and JSON properties identified in the query constructor. To cross multiple formats in a single search, use `cts:or-query` in XQuery or `cts.orQuery` in JavaScript to combine multiple geospatial queries.

For more details, see the following topics:

- "Converting To and From Common Geospatial Representations" on page 562
- "Constructing Geospatial Point and Region Values" on page 567
- `cts:geospatial-region-query` in the *MarkLogic XQuery and XSLT Function Reference*
- http://en.wikipedia.org/wiki/DE-9IM.

14.8.5 Constructing a Region Query from Query Text

You can use the `cts:parse` XQuery function or the `cts.parse` JavaScript function to create a geospatial region query from query text. If you bind a tag to a geospatial region path index, then you can use the tag in an expression of the following form in your query text:

```
boundTag operator criteriaRegion [options]
```

For example, if "reg" is the name of a tag bound to a geospatial region index, then the following expression parses to a geospatial region query than matches regions in your documents that overlap with the given polygon

```
reg DE9IM_OVERLAPS POLYGON((1 1,2 2,0 1,1 1))
```

The operator must be one of the following. For more details on the operators, see "Operators Usable with Geospatial Queries" on page 260 and http://en.wikipedia.org/wiki/DE-9IM.

- DE9IM_CONTAINS
- DE9IM_COVERED_BY
- DE9IM_COVERS
- DE9IM_CROSSES
- DE9IM_DISJOINT

- DE9IM_EQUALS
- DE9IM_INTERSECTS
- DE9IM_OVERLAPS
- DE9IM_TOUCHES
- DE9IM_WITHIN

The following example queries illustrate the tag binding and parsing necessary to search using this query text. The binding specifies the tag “reg” represents a geospatial region path index reference for the XPath expression “/envelope/cts-region”. The query text “@1 -122.2465038,37.5073428” represents a circle with radius 1 mile (the default units) and center (37.5073428, -122.2465038).

Language	Example

XML	`xquery version "1.0-ml";`
	`let $bindings := map:map()`
	`let $_ := map:put(`
	`  $bindings, "reg",`
	`  cts:geospatial-region-path-reference(`
	`  "/envelope/cts-region")`
	`)`
	`return cts:parse(`
	`  'reg DE9IM_OVERLAPS "@1 -122.2465038,37.5073428"',`
	`  $bindings`
	`)`

JavaScript	`const bindings = {`
	`  'reg': cts.geospatialRegionPathReference(`
	`  '/envelope/cts-region')`
	`};`
	`cts.parse(`
	`  'reg DE9IM_OVERLAPS "@1 -122.2465038,37.5073428"',`
	`  bindings);`

The parse produces a cts query similar to the following:

Language	Example Output

XQuery	`cts:geospatial-region-query(`
	`  (cts:geospatial-region-path-reference(`
	`  "/envelope/cts-region",("coordinate-system=wgs84"))`
	`  ), "overlaps",`
	`  cts:circle("@1 -122.2465,37.507343"),`
	`  (), 1)`

JavaScript	`cts.geospatialRegionQuery([`
	`  cts.geospatialRegionPathReference(`
	`  "/envelope/cts-region",`
	`  ["coordinate-system=wgs84"])`
	`  ],`
	`  "overlaps",`
	`  cts.circle("@1 -122.2465,37.507343"),`
	`  [], 1)`

For more details, see “Creating a Query From Search Text With cts:parse” on page 253.

14.8.6 Creating Region Queries Using the Client APIs

See the following topics for an overview and example of using a region query in a search in the Client APIs.

- JavaClient API
- Node.js Client API
- REST Client API

14.8.6.1 JavaClient API

This topic assumes you are already familiar with the search features of the Java Client API. If you are not, see the *Java Application Developer's Guide*.

You are most likely to construct a geospatial region query with the Java Client API using the StructuredQueryBuilder. You could also embed a structured region query in a `RawCombinedQuery`; this technique is not covered here. You cannot create a geospatial region query in Java using query text or QBE.

Each geospatial region query can only reference a single region index. To search more than one index, construct multiple region queries and combine them with an OR query.

Use `StructuredQueryDefinition.geospatial` to create a region query. Choose an overload that accepts a `GeospatialRegionIndex` as input. A `GeospatialRegionIndex` object identifies the region index to be searched.

To construct a `GeospatialRegionIndex` object, use `StructuredQueryBuilder.geoRegionPath`. When defining the index, you must include a `PathIndex` value, and you may also include coordinate system and precision information.

For example, the following code snippet identifies a region index on the path `/envelope/cts-region` with the coordinate system "wgs84".

```
DatabaseClient client = ...;
QueryManager qm = client.newQueryManager();
StructuredQueryBuilder sqb = qm.newStructuredQueryBuilder();
...
sqb.geoRegionPath(
    sqb.pathIndex("/envelope/cts-region"),
    StructuredQueryBuilder.CoordinateSystem.WGS84)
...
```

The following example uses the Java Client API to build a structured query equivalen to the query in "Example: Simple Intersection Region Query" on page 530. The example as written will only match the XML sample documents from "Preparing to Run the Examples" on page 569. You can match the JSON sample documents by changing the index path to `/envelope/ctsRegion`.

```
package examples;

import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.io.SearchHandle;
import com.marklogic.client.query.MatchDocumentSummary;
import com.marklogic.client.query.QueryManager;
import com.marklogic.client.query.StructuredQueryBuilder;
import
com.marklogic.client.query.StructuredQueryBuilder.GeoSpatialOperator;
import com.marklogic.client.query.StructuredQueryDefinition;


public class GeoRegionQuery {
    public static void main(String[] args) {
        // MODIFY THIS CALL TO MATCH YOUR ENV
        DatabaseClient client = DatabaseClientFactory.newClient(
            hostname, port, databaseName,
            new DatabaseClientFactory.DigestAuthContext(
                username, password));

        QueryManager qm = client.newQueryManager();
        StructuredQueryBuilder sqb = qm.newStructuredQueryBuilder();
        SearchHandle results = new SearchHandle();

        StructuredQueryDefinition query = sqb.geospatial(
```

```
                sqb.geoRegionPath(
                    sqb.pathIndex("/envelope/cts-region"),
                    StructuredQueryBuilder.CoordinateSystem.WGS84),
                GeospatialOperator.INTERSECTS,
                sqb.polygon(
                    sqb.point(37.519087, -122.26346),
                    sqb.point(37.521299, -122.24805),
                    sqb.point(37.512279, -122.24462),
                    sqb.point(37.50336,  -122.24556),
                    sqb.point(37.506185, -122.25981),
                    sqb.point(37.513436, -122.26337),
                    sqb.point(37.519087, -122.26346)
                            ));
            qm.search(query, results);
            for (MatchDocumentSummary match : results.getMatchResults()) {
                System.out.println(match.getUri());
            }

            client.release();
        }
    }
```

If you run the above program against the sample data and database configuration from "Preparing to Run the Examples" on page 569, you should see output similar to the following:

```
/geo-examples/Hwy-101.xml
/geo-examples/Holly-St.xml
/geo-examples/Wildlife-Refuge.xml
/geo-examples/Shopping-Center.xml
/geo-examples/MarkLogic-Neighborhood.xml
/geo-examples/Airport.xml
```

14.8.6.2 Node.js Client API

This topic assumes you are familiar with the search features of the Node.js Client API. If you are not, you may want to review the *Node.js Application Developer's Guide*.

To construct a geospatial region query, use queryBuilder.geospatialRegion. You cannot create a region query using `queryBuilder.parsedFrom` or `queryBuilder.byExample`. Use `queryBuilder.geoPath` to construct the region index specification, and helper functions such as `queryBuilder.polygon` to construct the criteria region(s).

Each geospatial region query can only reference a single region index. To search more than one index, construct multiple region queries and combine them with an OR query.

The following example performs the same search as "Example: Simple Intersection Region Query" on page 530. The example relies on the sample documents and database configuration from "Preparing to Run the Examples" on page 569. Before running the example, modify the connection information in `connInfo`.

```
const marklogic = require('marklogic');

// MODIFY THIS VAR TO MATCH YOUR ENV
const connInfo = {
    host: 'localhost',
    port: 8000,
    user: username,
    password: password,
    database: 'Documents'
  };
const db = marklogic.createDatabaseClient(connInfo);
const qb = marklogic.queryBuilder;

db.documents.query(
  qb.where(
    qb.geospatialRegion(
      qb.geoPath('/envelope/ctsRegion', qb.coordSystem('wgs84')),
      'intersects',
      qb.polygon(
        qb.point(37.519087, -122.26346),
        qb.point(37.521299, -122.24805),
        qb.point(37.512279, -122.24462),
        qb.point(37.50336,  -122.24556),
        qb.point(37.506185, -122.25981),
        qb.point(37.513436, -122.26337),
        qb.point(37.519087, -122.26346))
   ))
).result( function(results) {
  for (let result of results) {
    console.log(result.uri);
  }
});
```

If you run the example against the sample data from "Preparing to Run the Examples" on page 569, you should see output similar to the following:

```
/geo-examples/Hwy-101.json
/geo-examples/Wildlife-Refuge.json
/geo-examples/Holly-St.json
/geo-examples/Airport.json
/geo-examples/Shopping-Center.json
/geo-examples/MarkLogic-Neighborhood.json
```

For more details, see the *Node.js Application Developer's Guide*.

14.8.6.3 REST Client API

This topic assumes you are already familiar with the search features of the REST Client API. If you are not, refer to the *REST Application Developer's Guide*.

To evaluate a geospatial region using the REST Client API, you can use either a `cts:geospatial-region-query` or a structured query that contains a geo-region-path-query or a geo-region-constraint-query. Your cts or structured query can be standalone or part of a combined query. You cannot construct query text or a QBE that represents a region query.

Each structured region query can reference only one region index. To search more than one region index at a time, create multiple region queries and combine them with an `or-query`.

The following example uses the REST Client API and a structured query to perform the same search as the one in "Example: Simple Intersection Region Query" on page 530. The example as written will only match the XML sample documents from "Preparing to Run the Examples" on page 569. You can match the JSON documents by changing the `path-index` to `/envelope/ctsRegion`.

Copy the following query into a file. You will use it the file as the POST body of a search request. The example curl command below assumes the filename is `body.xml`.

```
<query xmlns="http://marklogic.com/appservices/search">
  <geo-region-path-query coord="wgs84">
    <path-index>/envelope/cts-region</path-index>
    <geospatial-operator>intersects</geospatial-operator>
    <polygon>
      <point>
        <latitude>37.519087</latitude><longitude>-122.26346</longitude>
      </point>
      <point>
        <latitude>37.521299</latitude><longitude>-122.24805</longitude>
      </point>
      <point>
        <latitude>37.512279</latitude><longitude>-122.24462</longitude>
      </point>
      <point>
        <latitude>37.50336</latitude><longitude>-122.24556</longitude>
      </point>
      <point>
        <latitude>37.506185</latitude><longitude>-122.25981</longitude>
      </point>
      <point>
        <latitude>37.513436</latitude><longitude>-122.26337</longitude>
      </point>
      <point>
        <latitude>37.519087</latitude><longitude>-122.26346</longitude>
      </point>
    </polygon>
  </geo-region-path-query>
</query>
```

Run a `curl` command similar the following to perform the search. Before running the command, change the username and password. If you are not using the Document database as your content database, you will need to add a `database` request parameter to the URL.

```
curl --anyauth --user user:password -X POST -i \
  -d @./body.xml -H "Content-type: application/xml" \
  'http://localhost:8000/v1/search'
```

The search should match the following documents:

```
/geo-examples/Hwy-101.xml
/geo-examples/Holly-St.xml
/geo-examples/Wildlife-Refuge.xml
/geo-examples/Shopping-Center.xml
/geo-examples/MarkLogic-Neighborhood.xml
/geo-examples/Airport.xml
```

The following is the equivalent structured query, expressed as JSON.

```
{"query": {
  "geo-region-path-query": {
    "path-index": { "text": "/envelope/cts-region" },
    "coord": "wgs84",
    "geospatial-operator": "intersects",
    "polygon": [
      { "point": [
        { "latitude": 37.519087, "longitude": -122.26346 },
        { "latitude": 37.521299, "longitude": -122.24805 },
        { "latitude": 37.512279, "longitude": -122.24462 },
        { "latitude": 37.50336,  "longitude": -122.24556 },
        { "latitude": 37.506185, "longitude": -122.25981 },
        { "latitude": 37.513436, "longitude": -122.26337 },
        { "latitude": 37.519087, "longitude": -122.26346 }
      ] }
    ]
  }
}}
```

You can use this query with a curl command similar to the XML example. Just change the request body content type and, potentially, name of the file containing the body. For example:

```
curl --anyauth --user user:password -X POST -i \
  -d @./body.json -H "Content-type: application/json" \
  'http://localhost:8000/v1/search'
```

For more details, see Using and Configuring Query Features in the *REST Application Developer's Guide* and "Searching Using Structured Queries" on page 74.

14.8.7 Example: Using the Envelope Pattern to Encode Regions

Content you search with a region query must be in WKT or serialized cts:region format. This example illustrates using the "envelope pattern" to encapsulate the searchable region format with original data in an incompatible format. This is not the only solution to this problem. For example, you can tranform your content before ingesting it into MarkLogic, or you can replace the unsupported original format entirely, rather than persisting both.

The example reads in a file from the file system that contains an aggregate XML element contains a several KML Placemark elements. The data is disaggregated into one file per Placemark, and then each Placmark is wrapped in an "envelope" that contains both the original data and the serialized representation of a cts:region that corresponds to the region in the original data.

For example, if the original input file has the following structure:

```
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Placemark>
    <name>Hwy 101</name>
    <description>...</description>
    <LineString>
      <extrude>0</extrude>
      <tessellate>1</tessellate>
      <altitudeMode>clampedToGround</altitudeMode>
      <coordinates>
        -122.2637558,37.5206187 -122.2428131,37.5020318
      </coordinates>
    </LineString>
  </Placemark>
  <Placemark>...</Placemark>
  ...
</kml>
```

Then the result is one document per Placemark, with the following structure. The `envelope` root element and the `cts-region` element are created by the ingest transformation.

```
<envelope>
  <cts-region>LINESTRING(-122.26376 37.520619,-122.24281
37.502032)</cts-region>
  <Placemark xmlns="http://www.opengis.net/kml/2.2">
    <name>Hwy 101</name>
    <description>...</description>
    <LineString>
      <extrude>0</extrude>
      <tessellate>1</tessellate>
      <altitudeMode>clampedToGround</altitudeMode>
      <coordinates>
        -122.2637558,37.5206187 -122.2428131,37.5020318
      </coordinates>
    </LineString>
  </Placemark>
<envelope>
```

The following example query ingests the original data and creates a document from the envelope it wraps around each Placemark:

```
xquery version "1.0-ml";
import module namespace geokml = "http://marklogic.com/geospatial/kml"
         at "/MarkLogic/geospatial/kml.xqy";
declare namespace kml="http://www.opengis.net/kml/2.2";

(: Convert the KML regions into cts regions :)
declare function local:region-convert(
  $nodes as node()*
) as cts:region*
{
  for $n in $nodes return
  typeswitch($n)
    case element(kml:Polygon) return geokml:parse-kml($n)
    case element(kml:LineString) return geokml:parse-kml($n)
    case element(kml:Point) return ()
    default return local:region-convert($n/node())
};

(: Create a doc for each KML Placemark, with a wrapper around
 : the KML that contains the cts region equiv of the KML region :)
let $file := xdmp:document-get("/space/geo/ml2.xml")
return
  for $place in $file//*:Placemark
  let $basename := fn:string-join(fn:tokenize($place/*:name, " "),"-")
  return xdmp:document-insert(
    fn:concat("/example/",$basename,".xml"),
    <envelope>{
     let $converted-region := local:region-convert($place)
     return if (fn:empty($converted-region))
       then ()
       else <cts-region>{$converted-region}</cts-region>
    }
    {$place}
    </envelope>,
    xdmp:default-permissions(), "geo-example")
```

Points are not translated by the above example simply because it was not necessary for this purpose. You can index and use point queries on in KML points without transformation. You only need to transform them if you want to use a region query on point data.

14.9 Controlling Coordinate System and Precision

- The Relationship Between Precision and Coordinate System
- Determining the Best Precision for Your Application
- How MarkLogic Selects the Governing Coordinate System
- Probing the Governing Coordinate System Name

- Specifying the App Server Default Coordinate System
- Specifying the Per-Module Coordinate System
- Specifying a Per-Operation Coordinate System and Precision
- Specifying Coordinate System During Index Creation

14.9.1 The Relationship Between Precision and Coordinate System

The coordinate system and precision are conflated in the coordinate system name in many operations that accept a coordinate system name as input.

For example, when you specify “wgs84” as the value of the “coordinate-system” option in a query constructor, it also implicitly specifies single precision. Similarly, a value of “wgs84/double” specifies both the WGS84 coordinate system and double precision.

In many interfaces, you can use a precision option or parameter to override the precision implied by the coordinate system name.

14.9.2 Determining the Best Precision for Your Application

MarkLogic always preserves the precision of geospatial data in your documents. For example, if you ingest documents containing double precision coordinate values, those values retain full precision, even if the governing coordinate system during ingestion specifies single precision.

However, the precision of values in a geospatial index is determined by the configuration of the index. Thus, you might have double precision values in your documents, but single precision values in the corresponding geospatial index.

A double precision index enables a greater degree of accuracy when computing geospatial search matches, but at the cost of increased memory requirements and some computational overhead.

> **Note:** Greater precision does not equate to greater accuracy. Most applications do not require double precision indexing.

For example, geospatial queries against single precision indexes are accurate to within 1 meter for geodetic coordinate systems. If your application does not require sub-meter accuracy, then there is no reason to incur the overhead of a double precision index.

The following are examples of geospatial applications that might require double precision:

- Tracking equipment moving around a facility.
- Tracking room-to-room movements within a building.
- Tracking slow-moving objects that move in sub-meter increments, such as fault lines and tectonic plates.

- Tracking assets that require high placement precision, such on which side of a street a fire hydrant is located.

Excessive precision can cause difficulty for geospatial operations. For example, when comparing two points at double precision, they will fail a test for equality if the coordinate values differ when compared at the level of microns. Most applications would consider such a difference "in the noise" and consider these points the same. Comparison of double-precision coordinates assumes a tolerance of zero by default, meaning they must match exactly, at all digits of precision. This affects operations such as comparison of points, testing whether a point is on a edge, and testing two edges for adjacency. You can use the tolerance option available on some operations to enable less precise comparisons. For more details, see "Understanding Tolerance" on page 558.

An application can use a mix of single and double precision geospatial indexes and operations. For example, you can define both a single and a double precision index over the same data. You can specify precision per operation.

You can control geospatial precision in the following ways:

- Specify float or double precision when creating a geospatial point or region index. This determines the precision of values stored in the index. For details, see "Determining the Best Precision for Your Application" on page 550.
- Configure an App Server default precision. This specifies the precision to use in geospatial queries and computations when no other precision override is in effect. The default is single (float) precision. For details, see "Specifying the App Server Default Coordinate System" on page 554.
- In XQuery, you can specify a default precision for a main module. This overrides the App Server default precision. For details, see "Specifying the Per-Module Coordinate System" on page 555.
- Specify precision on an operation, such as when constructing a geospatial query, computing a distance, or accessing the coordinates of a box. This overrides the App Server and module default precision. For details, see "Specifying a Per-Operation Coordinate System and Precision" on page 556.

You can specify precision in conjunction with the coordinate system name in most MarkLogic geospatial interfaces. For example, the "wgs84" and "raw" coordinate system names imply single precision, while the "wgs84/double" and "raw/double" coordinate system names specify double precision.

14.9.3 How MarkLogic Selects the Governing Coordinate System

When MarkLogic evaluates your XQuery or Server-Side JavaScript code, the governing coordinate system is the first of the following settings found. For more details, see "The Governing Coordinate System" on page 486.

- Per-operation coordinate system option or parameter

- Per-module coordinate system, as specified by the XQuery `xdmp:coordinate-system` prolog option. (This feature is only available in XQuery main modules.)
- App Server default coordinate system

If you specify a precision using the `precision` option of an operation, the specified precision always takes precedence over the precision implied by the governing coordinate system name.

The following examples illustrate the governing coordinate system applied in several calling contexts if the App Server default coordinate system is "wgs84".

Language	Example

XQuery
No prolog option
	`(: app server default is wgs84 :)`
	`xquery version "1.0-ml";`
	`<wrapper>`
	`  <wgs84>{  (: app server default coord-sys :)`
	`  geo:distance(cts:point(1.0,1.0), cts:point(2.0,2.0))`
	`  }</wgs84>`
	`  <wgs84d>{  (: per-operation coord-sys :)`
	`  geo:distance(cts:point(1.0,1.0), cts:point(2.0,2.0),`
	`  ("coordinate-system=wgs84/double"))`
	`  }</wgs84d>`
	`  <raw>{  (: per-op coord-sys, with precision override :)`
	`  geo:distance(cts:point(1.0,1.0), cts:point(2.0,2.0),`
	`  ("coordinate-system=raw/double",`
	`  "precision=float"))`
	`  }</raw>`
	`</wrapper>`
	
	`(: produces:`
	`  <wrapper>`
	`  <wgs84>97.4783192326097</wgs84>`
	`  <wgs84-d>97.4783199874495</wgs84-d>`
	`  <raw>1.4142135623731</raw>`
	`  </wrapper>`
	`:)`

Language	Example
XQuery Prolog option	(: app server default is wgs84 :) xquery version "1.0-ml"; **declare option xdmp:coordinate-system "raw";** <wrapper> <raw>{ (: per module coord-sys :) geo:distance(cts:point(1.0,1.0), cts:point(2.0,2.0)) }</raw> <wgs84d>{ (: per-operation coord-sys :) geo:distance(cts:point(1.0,1.0), cts:point(2.0,2.0), ("coordinate-system=wgs84/double")) }</wgs84d> <wgs84>{ (: per-op coord-sys, with precision override :) geo:distance(cts:point(1.0,1.0), cts:point(2.0,2.0), ("coordinate-system=wgs84/double", "precision=float")) }</wgs84> </wrapper> (: produces: <wrapper> <raw>1.4142135623731</raw> <wgs84d>97.4783199874495</wgs84d> <wgs84>97.4783192326097</wgs84> </wrapper> :)

Server-Side JavaScript	const result = {
	wgs84: // app server default coord-sys
	geo.distance(cts.point(1.0,1.0), cts.point(2.0,2.0)),
	wgs84d: // op specific coord-sys
	geo.distance(cts.point(1.0,1.0), cts.point(2.0,2.0),
	['coordinate-system=wgs84/double']),
	raw: // op specific coord-sys w precision override
	geo.distance(cts.point(1.0,1.0), cts.point(2.0,2.0),
	['coordinate-system=raw/double',
	'precision=float'])
	};
	result
	
	/* produces:
	{ "wgs84":97.4783192326097,
	"wgs84d":97.4783199874495,
	"raw":1.4142135623731
	}
	
	*/

See the following topics for instructions on setting the coordinate system and precision at various levels:

- "Specifying the App Server Default Coordinate System" on page 554
- "Specifying the Per-Module Coordinate System" on page 555
- "Specifying a Per-Operation Coordinate System and Precision" on page 556

14.9.4 Probing the Governing Coordinate System Name

You can probe the governing coordinate system using the XQuery function `geo:default-coordinate-system` or the JavaScript function `geo.defaultCoordinateSystem`. (This function can only account for the App Server default and per-module settings.)

The following examples illustrate how to retrieve the name of the governing coordinate system.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`geo:default-coordinate-system();`
	`(: returns the app server default coordinate system :)`
	
	`xquery version "1.0-ml";`
	`declare option xdmp:coordinate-system "wgs84/double";`
	`geo:default-coordinate-system();`
	`(: returns the per-module setting, wgs84/double :)`

Server-Side JavaScript	`geo.defaultCoordinateSystem();`
	
	`// returns the app server default coordinate system`

14.9.5 Specifying the App Server Default Coordinate System

You can use the following Admin library functions to set and get the default coordinate system/precision combination for an App Server. If you do not explicitly set the coordinate system and precision, it is "wgs84" (single precision).

- `admin:appserver-set-coordinate-system`
- `admin:appserver-get-coordinate-system`

For example, the following XQuery code sets the default coordinate system for the App Server named "MyAppServer" to "wgs84/double".

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

let $config := admin:get-configuration()
```

```
let $groupid := admin:group-get-id($config, "Default")
return admin:save-configuration(
  admin:appserver-set-coordinate-system(
    $config,
    admin:appserver-get-id($config, $groupid, "MyAppServer"),
    "wgs84/double")
)
```

You can also use the XQuery Admin library module from Server-Side JavaScript. The following example is equivalent to the previous XQuery code.

```
const admin = require('/MarkLogic/admin');
const config = admin.getConfiguration();
const groupId = admin.groupGetId(config, 'Default');
admin.saveConfiguration(
  admin.appserverSetCoordinateSystem(
    config,
    admin.appserverGetId(config, groupId, 'MyAppServer'),
    'wgs84/double')
)
```

To determine the canonical name for a coordinate system/precision combination, use the XQuery function `geo:coordinate-system-canonical` or the JavaScript function `geo.coordinateSystemCanonical`.

For more details, see the *XQuery and XSLT Reference Guide*.

14.9.6 Specifying the Per-Module Coordinate System

In XQuery, you can use the `xdmp:coordinate-system` prolog option to override the App Server default coordinate system module-wide. This option is only available in XQuery. For example:

```
declare option xdmp:coordinate-system "wgs84/double";
```

The override only takes effect when you declare the option in an XQuery main module, but it affects any library module functions subsequently invoked from that main module.

Note: REST, Java, and Node.js Client API resource extensions are library modules, so you cannot override the coordinate system in your extension implementation. Use the ad-hoc query (eval or invoke) of the Client APIs or a per-operation override if you need to override the App Server default with these APIs.

For example, if you create a library function that just returns the result of calling `geo:default-coordinate-system`, then the following main module will return “wgs84/double” for the coordinate system.

```
xquery version "1.0-ml";
import module namespace my = "http://marklogic.com/example/my-lib"
  at "/my/lib.xqy";
```

```
declare option xdmp:coordinate-system "wgs84/double";
my:get-coord-sys()
```

14.9.7 Specifying a Per-Operation Coordinate System and Precision

Many geospatial operations accept options for specifying the coordinate system and/or precision. A per-operation specification overrides the governing coordinate system. For example:

Language	Example

XQuery	`(: xquery :)`
	`geo:distance(`
	`  cts:point(1.0,1.0), cts:point(2.0,2.0),`
	`  ("coordinate-system=wgs84", "precision=double"))`

Server-Side JavaScript	`(: javascript :)`
	`geo.distance(`
	`  cts.point(1.0,1.0), cts.point(2.0,2.0),`
	`  ['coordinate-system=wgs84', 'precision=double'])`

Where both the coordinate-system and precision options are supported, you can specify the precision either as part of the coordinate system canonical name or independently. Where there is a conflict between the precision in the coordinate system name and the precision option, the precision option takes precedence.

The following example illustrates how the option settings interact:

Options	Resulting Coordinate System
`coordinate-system=wgs84/double`	`wgs84/double`
`coordinate-system=wgs84` `precision=double`	`wgs84/double`
`coordinate-system=wgs84/double` `precision=float`	`wgs84` (single precision)

You can get the canonical name for a coordinate system/precision combination using the XQuery function `geo:coordinate-system-canonical` or the JavaScript function `geo.coordinateSystemCanonical`.

14.9.8 Specifying Coordinate System During Index Creation

You can choose single or double precision when creating a geospatial index. This determines the precision of the values stored in the index. For example, you can create a single precision index over your geospatial data even if the data in your documents is double precision.

Specify precision during index creation through the coordinate system name. You can use the XQuery function `geo:coordinate-system-canonical` or the JavaScript function `geo.coordinateSystemCanonical` to generate the canonical name of the desired coordinate system and precision combination.

For example, the following code creates a geospatial element index for double precision wgs84 point values:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace admin =`
	`"http://marklogic.com/xdmp/admin"`
	`  at "/MarkLogic/admin.xqy";`
	
	`let $config := admin:get-configuration()`
	`let $dbid := xdmp:database("Documents")`
	`let $geo-index-spec := admin:database-geospatial-element-index(`
	`  "/my/namespace", "elementname",`
	`  **geo:coordinate-system-canonical("wgs84", "double")**,`
	`  fn:false() )`
	`return admin:save-configuration(`
	`  admin:database-add-geospatial-element-index(`
	`  $config, $dbid, $geo-index-spec)`
	`)`

Server-Side JavaScript	`const admin = require('/MarkLogic/admin');`
	
	`const config = admin.getConfiguration()`
	`const dbid = xdmp.database('Documents')`
	`const geoIndexSpec = admin.databaseGeospatialElementIndex(`
	`  '/my/namespace', 'elementName',`
	`  **geo.coordinateSystemCanonical('wgs84', 'double')**,`
	`  false)`
	`admin.saveConfiguration(`
	`  admin.databaseAddGeospatialElementIndex(config, dbid, geoIndexSpec)`
	`)`

For more details on geospatial indexes, see “Understanding Geospatial Query and Index Types” on page 493.

14.10 Understanding Tolerance

Tolerance is the largest allowable variation in geometry calculations. Tolerance is a distance within which two points are considered equal, a point is consider "on" an edge, or two edges are considered touching. Many geospatial functions in MarkLogic accept a "tolerance" option.

See the following topics for more details:

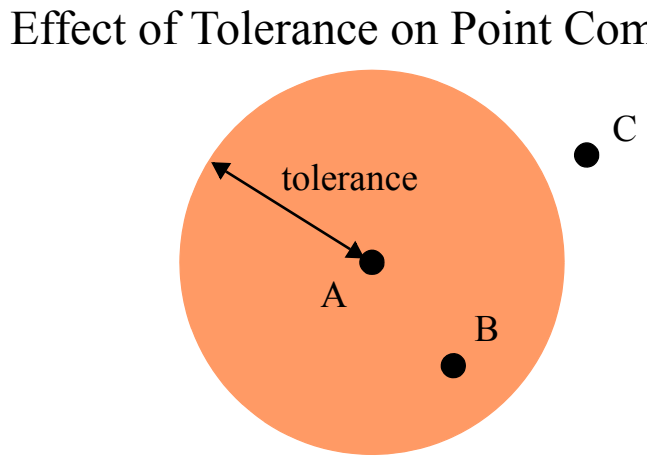
- How Tolerance Affects Geometric Comparisons
- Considerations for Tolerance Selection

14.10.1 How Tolerance Affects Geometric Comparisons

Tolerance defines the "largest acceptable error" when comparing two points for equality.

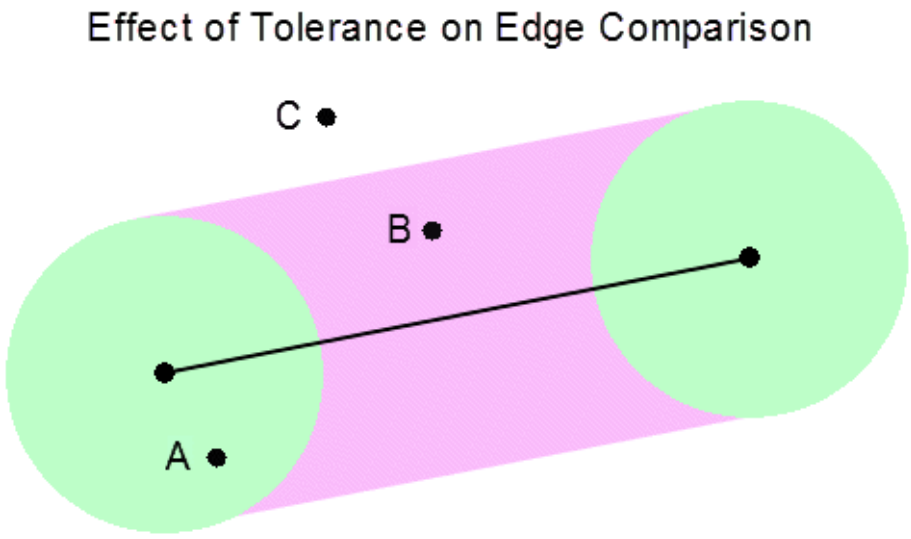
For example, a tolerance of zero means two points only match if they're exactly the same, out to the least significant digit. Thus, two points separated by a distance measurable in microns would not match. If you're trying to determine whether a truck is parked at the door of a building, such a high degree of precision is a hindrance. Use tolerance to filter out differences that are "in the noise".

The following diagram illustrates how tolerance affects point comparison. A, B, and C are points. The shaded circle describes the space within which points are considered "equal to" A, based on the tolerance. Point B falls within tolerance of A, so A and B are considered equal. Point C is further from A than the tolerance allows, so A and C are not considered equal.



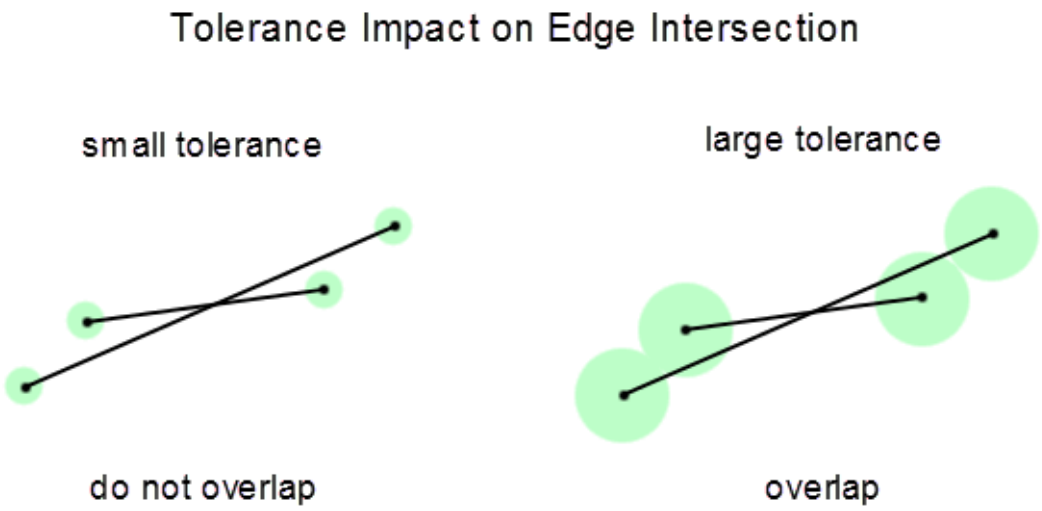
When comparing edges, a point is considered as lying "on the edge" if the distance from the point to the edge is within the tolerance.

For example, in the following diagram, any point in the two circles coincides with an endpoint of the edge. Any point within the center region lies on the edge. Thus, point A coincides with an endpoint and point B lies on the edge. Point C is outside the the tolerance range, so it is not considered to lie on the edge.



Operations such as computing whether two polygons intersect require comparing two edges. Two edges overlap if both endpoints of one edge lie on the other, or if an endpoint of each edge lies on the other.

For example, the following diagram illustrates the effect of two different tolerance values on determining overlap. The circles represent the tolerance of each endpoint. With the smaller tolerance, the edges do not overlap. With the larger tolerance, both endpoints of one edge are within tolerance of the other edge, so the edges overlap.



14.10.2 Considerations for Tolerance Selection

If you do not explicitly set tolerance, MarkLogic uses the default tolerance appropriate for the coordinate system.

To ensure accuracy, MarkLogic enforces a minimum tolerance for each coordinate system. If tolerance is too precise, then the calculation of distance might not be accurate to the specified level of precision.

You cannot choose a tolerance value less than zero.

For most operations, MarkLogic interprets a tolerance of zero as the minimum tolerance for the coordinate system. The only exceptions are the XQuery function `geo:bounding-boxes` and the JavaScript function `geo.boundingBoxes`, as follows:

When computing bounding boxes, a non-zero tolerance causes the bounding boxes to be "padded" by the tolerance amount. This ensures the bounding box covers the "thickened" boundary of the region under consideration. If you set tolerance to zero when computing bounding boxes, then the bounding boxes are not padded at all.

When considering a polygon, tolerance effectively "thickens" the boundary of the polygon. If you set the tolerance too high relative to the size of the polygon, the polygon degenerates. This can result in unexpected results or errors.

You can use the XQuery `geo:region-approximate` or the Server-Side JavaScript function `geo.regionApproximate` to simply your region(s) before performing geometric computations. The simplification can sometimes help you balance tolerance against polygon degneration.

Geospatial computational and comparison operations that do not accept a tolerance option behave as if tolerance is set to zero.

14.11 Summary of Other Geospatial Operations

The following APIs are used to perform various operations and calculations on geospatial data:

XQuery	JavaScript
`geo:polygon-contains`	`geo.polygonContains`
`geo:complex-polygon-contains`	`geo.complexPolygonContains`
`geo:region-contains`	`geo.regionContains`
`geo:arc-intersection`	`geo.arcIntersection`

XQuery	JavaScript
geo:box-intersects	geo.boxIntersects
geo:circle-intersects	geo.circleIntersects
geo:polygon-intersects	geo.polygonIntersects
geo:complex-polygon-intersects	geo.complexPolygonIntersects
geo:region-intersects	geo.regionIntersects
geo:region-approximate	geo.regionApproximate
geo:region-clean	geo.regionClean
geo:bounding-boxes	geo.boundingBoxes
geo:polygon-to-linestring	geo.polygonToLinestring
geo:linestring-reverse	geo.linestringReverse
geo:linestring-concat	geo.linestringConcat
geo:circle-polygon	geo.circlePolygon
geo:ellipse-polygon	geo.ellipsePolygon
geo:interior-point	geo.interiorPoint
geo:count-vertices	geo.countVertices
geo:count-distinct-vertices	geo.countDistinctVertices
geo:remove-duplicate-vertices	geo.removeDuplicateVertices
geo:distance	geo.distance
geo:shortest-distance	geo.shortestDistance
geo:destination	geo.destination
geo:bearing	geo.bearing
geo:approx-center	geo.approxCenter
geo:region-affine-transform	geo.regionAffineTransform
geo:region-relate	geo.regionRelate
geo:region-de9im	geo.regionDe9im

For signatures and more details, see the Library Module section of the *MarkLogic XQuery and XSLT Function Reference* or the *MarkLogic Server-Side JavaScript Function Reference*.

14.12 Converting To and From Common Geospatial Representations

MarkLogic provides interfaces for converting between MarkLogic geospatial primitive types and several common geospatial text, XML, and JSON representations. This section covers the following topics:

- Conversion Overview
- WKT and WKB Conversions in XQuery
- WKT and WKB Conversions in JavaScript
- Mapping of WKT and WKB Types to MarkLogic Types

14.12.1 Conversion Overview

You can use MarkLogic APIs to convert to and from the following common geospatial representations:

- Well-Known Text (WKT)
- Well-Known Binary (WKB)
- GML
- KML
- GeoJSON
- GeoRSS

For example, the following XQuery code uses `geogml:parse-gml` to convert a GML region into a `cts:region` (a polygon in this case). This function determines the output `cts:region` type from the kind of input GML region.

```
import module namespace geogml ="http://marklogic.com/geospatial/gml"
  at "/MarkLogic/geospatial/gml.xqy";

geogml:parse-gml(
  <gml:Polygon srsName="ML:wgs84"
      xmlns:gml="http://www.opengis.net/gml/3.2">
    <gml:exterior>
      <gml:LinearRing>
        <gml:posList srsDimension="2">
          5.0 1.0 8.0 1.0 8.0 6.0 5.0 7.0 5.0 1.0
        </gml:posList>
      </gml:LinearRing>
    </gml:exterior>
```

```
    </gml:Polygon>
)
```

If you know the input region type, you can also use one of the region-specific constructors to perform the equivalent conversion. For example, the above code could use `geogml:polygon` instead of `geogml:parse-gml`. For details, see "Constructing Geospatial Point and Region Values" on page 567.

The following Server-Side JavaScript code converts a GeoJSON polygon into a `cts.polygon`:

```
// Create a cts.polygon from a GeoJSON polygon
const geojson = require('/MarkLogic/geospatial/geojson.xqy');

geojson.polygon(
  { type: 'Polygon',
    coordinates: [
      [[100.0,0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0]]
  ] }
)
```

For each format, the XQuery API includes a `parse-`*`format`* function for converting from the common representation to a MarkLogic geospatial primitive type, and the JavaScript API includes a `parse`*`Format`* function for the same purpose. This operation is equivalent to calling the `geo:parse` XQuery function or the `geo.parse` JavaScript function with input of the same format. The API also includes a `to-`*`format`* XQuery function and `to`*`Format`* JavaScript function for converting from a MarkLogic primitive type to the target format.

For example, the GeoJSON library module includes the following functions that can be used to convert data between GeoJSON and `cts:region`.

XQuery	JavaScript
`geojson:parse-geojson`	`geojson.parseGeojson`
`geojson:to-geojson`	`geojson.toGeojson`
`geojson:box`	`geojson.box`
`geojson:circle`	`geojson.circle`
`geojson:complex-polygon`	`geojson.complexPolygon`
`geojson:linestring`	`geojson.linestring`
`geojson:multi-linestring`	`geojson.multiLinestring`
`geojson:point`	`geojson.point`
`geojson:polygon`	`geojson.polygon`

You can use the built-in `geo:parse` XQuery function or `geo.parse` JavaScript function to convert nodes in any of the supported formats into an equivalent MarkLogic geospatial primitive type, without regard to the input format or region type. For best performance, if you know the format, use the equivalent format-specific functions.

14.12.2 WKT and WKB Conversions in XQuery

MarkLogic represents geospatial data using the `cts:region` type and types derived from it, such as `cts:point`, `cts:polygon`, and `cts:circle`. You can convert from WKT or WKB into `cts:region` items and from `cts:region` into WKT or WKB.

Use the `geo:parse-wkt` function to convert WKT data into a sequence of `cts:region` items. Similarly, use `geo:parse-wkb` to convert WKB data into a sequence of `cts:region` items. You can use the resulting items in geospatial `cts:query` constructors or geospatial operations.

For example, the following call converts a WKT polygon with an inner and outer boundary into a `cts:complex-polygon`:

```
geo:parse-wkt("
 POLYGON(
  (0 0, 0 10, 10 10, 10 0, 0 0),
  (0 5, 0 7, 5 7, 5 5, 0 5) )" )
```

The input to `geo:parse-wkb` is a binary node that contains a WKB byte sequence. For example, the following code converts a WKB byte sequence representing the coordinates (-73.700380647, 40.739754168) into a `cts:point`:

```
geo:parse-wkb(
  binary { "0101000000726759090D36C52C0E151BB43B05E4440" }
)
```

To convert from `cts:region` to WKT, use `geo:to-wkt`. For example, the following code returns a WKT `POINT`:

```
geo:to-wkt(cts:point(1, 2))
```

Similarly, the following code returns a WKB `POINT`:

```
geo:to-wkb(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.

14.12.3 WKT and WKB Conversions in JavaScript

MarkLogic represents geospatial data using the `cts.region` type and types derived from it, such as `cts.point`, `cts.polygon`, and `cts.circle`. MarkLogic provides the following conversions between WKT or WKB and `cts.region`: You can use the `cts.region` representation `cts:query` constructors or geospatial operations.

- Explict conversion from WKT or WKB to `cts.region` using the `geo.parseWkt` function. For example:

```
// Convert WKT polygon into a cts.complexPolygon
geo.parseWkt(
  'POLYGON((0 0, 0 10, 10 10, 10 0, 0 0),(0 5, 0 7, 5 7, 5 5, 0 5))'
)

// Convert WKB bye sequence representing the coordinates
// (-73.700380647, 40.739754168) into a cts.point
geo.parseWkb(
  new NodeBuilder()
    .addBinary('0101000000726759 09D36C52C0E151BB43B05E4440')
    .toNode()
)
```

- Explicit converstion from `cts.region` to WKT or WKB using the `geo.toWkt` or `geo.toWkb` functions. For example:

```
// cts.point to WKT
geo.toWkt(cts.point(1, 2))

// cts.point to WKB
geo.toWkb(cts.point(1, 2))
```

- Implicit conversion from WKT to `cts.region` where the expected type is a `cts.region`. For example:

```
// create a cts.polygon from WKT via implicit conversion
cts.polygon('POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))')
```

Note that `geo.parseWkt` and `geo.toWkt` return a `Sequence` rather than an array or a single value. The input to `geo.parseWkb` is a binary node that contains a WKB byte sequence.

The supported conversions from WKT to `cts.region` mean all the following calls pass the same `cts.polygon` value to `geo.polygonContains`, which returns true.:

```
// Use a cts.polygon created from a set of cts.point values
geo.polygonContains(
  cts.polygon([
    cts.point(30,10), cts.point(40,40), cts.point(20,40),
    cts.point(10,30), cts.point(30,10)]),
  cts.point(25,25))
```

```
// Use a cts.polygon created by explicitly converting from WKT
geo.polygonContains(
  geo.parseWkt('POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))'),
  cts.point(25,25))

// Use a cts.polygon created by implicitly converting from WKT
geo.polygonContains(
  cts.polygon('POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))'),
  cts.point(25,25))
```

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.

14.12.4 Mapping of WKT and WKB Types to MarkLogic Types

The following table shows how the WKT and WKB types map to the MarkLogic geospatial types. That is, the equivalent value type resulting from calling the `geo:parse-wkt` XQuery function or `geo.parseWkt` JavaScript function, or the WKB equivalents.

WKT/WKB Geometry	MarkLogic XQuery Type	MarkLogic JavaScript Type
POINT	`cts:point`	`cts.point`
POINT EMPTY (WKT only)	`cts:point` (flagged as empty)	`cts.point` (flagged as empty)
POLYGON	`cts:complex-polygon` | `cts:polygon`	`cts.complexPolygon` | `cts.polygon`
POLYGON EMPTY	`cts:complex-polygon` (flagged as empty)	`cts.complexPolygon` (flagged as empty)
LINESTRING	`cts:linestring`	`cts.linestring`
LINESTRING EMPTY	`cts:linestring` (flagged as empty)	`cts.linestring` (flagged as empty)
TRIANGLE	`cts:polygon`	`cts.polygon`
TRIANGLE EMPTY	`cts:complex-polygon` (flagged as empty)	`cts.complexPolygon` (flagged as empty)
MULTIPOINT	`cts:point`*	zero or more `cts.point` nodes
MULTIPOINT EMPTY	`()`	an empty `Sequence`
MULTILINESTRING	`cts:linestring`*	zero or more `cts.linestring`

WKT/WKB Geometry	MarkLogic XQuery Type	MarkLogic JavaScript Type
MULTILINESTRING EMPTY	`()`	null, empty array, or empty `Sequence`
MULTIPOLYGON	(`cts:polygon` | `cts:complex-polygon`)*	(`cts.polygon` | `cts.complexPolygon`)*
MULTIPOLYGON EMPTY	`()`	null, empty array, or empty `Sequence`
GEOMETRYCOLLECTION	`cts:region`*	zero or more `cts.region` nodes
GEOMETRYCOLLECTION EMPTY	`()`	null, empty array, or empty `Sequence`
others	throws XDMP-BADWKT	throws XDMP-BADWKT

14.13 Constructing Geospatial Point and Region Values

Use the following APIs to construct geospatial regions. You can use the resulting region values in geospatial query constructors and other geospatial operations, such as those listed in "Summary of Other Geospatial Operations" on page 560.

XQuery	JavaScript
`cts:box`	`cts.box`
`cts:circle`	`cts.circle`
`cts:complex-polygon`	`cts.complexPolygon`
`cts:linestring`	`cts.linestring`
`cts:point`	`cts.point`
`cts:polygon`	`cts.polygon`

These constructors accept either the raw data, such as a pair of float values for constructing a point, or a string representing the serialization of the underlying primitive type. The serialized representation can be either the MarkLogic internal representation, such as a serilaized cts:point, or a WKT serialization. If the primitive is not constructible from the string input, an exception is thrown.

Each constructor produces a region value of the corresponding primitive type. For example, the `cts:box` constructor function creates a value of type `cts:box`. Each of the geospatial primitive types is an instance of the `cts:region` base type (`cts.region` in JavaScript).

For example, the following call constructs a `cts:polygon` from a string that is a serialized cts:point value (space-separated points):

XQuery	JavaScript
`cts:polygon("38,-10 40,-10 39, -15")`	`cts.polygon('38,-10 40,-10 39, -15')`

You can also construct the primitive types from XML or JSON nodes that contain geospatial data in the supported formats. For example, the following XQuery code uses the `geokml:box` function to construct a cts:box from an XML element containing a KML LatLongBox.

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

geokml:box(
  <LatLongBox xmlns="http://www.opengis.net/kml/2.2">
    <north>30</north>
    <south>12.5</south>
    <east>-122.24</east>
    <west>-127.24</west>
  </LatLongBox>)
```

Similarly, the following example uses `geojson.box` JavaScript function to construct a cts.box from a JSON node that contains a suitable GeoJSON polygon. For example:

```
const geojson = require('/MarkLogic/geospatial/geojson.xqy');

geojson.box(
  { type: 'Feature',
    bbox: [-180.0, -90.0, 180.0, 90.0],
    geometry: {
      type: 'Polygon',
      coordinates: [[
        [-180.0, 10.0], [20.0, 90.0], [180.0, -5.0], [-30.0, -90.0]
      ]]
  }}
)
```

For details and examples on these functions, see the *MarkLogic XQuery and XSLT Function Reference* or the *MarkLogic Server-Side JavaScript Function Reference*.

14.14 Geospatial Query Support in Other APIs

The Search API enables geospatial queries through the following features:

- Define geospatial constraints using query options such as `geo-elem-pair-constraint`, `geo-path-constraint`, and `geo-json-property-constraint`. For details, see `search:search` and "Search Customization Using Query Options" on page 381.
- Create geospatial structured queries using composers such as `geo-elem-query` and `geo-json-property-pair-query`. For details, see "Searching Using Structured Queries" on page 74.
- Add a `heatmap` to a geospatial point constraint to generate geospatial facets. For an example, see "Geospatial Constraint Example" on page 393.

For information on specific geospatial query options, see "Appendix: Query Options Reference" on page 816.

The Client APIs for REST, Java and Node.js applications provide similar support. For more details and example see the following topics:

- "Creating Point Queries with the Client APIs" on page 522
- "Creating Region Queries Using the Client APIs" on page 542

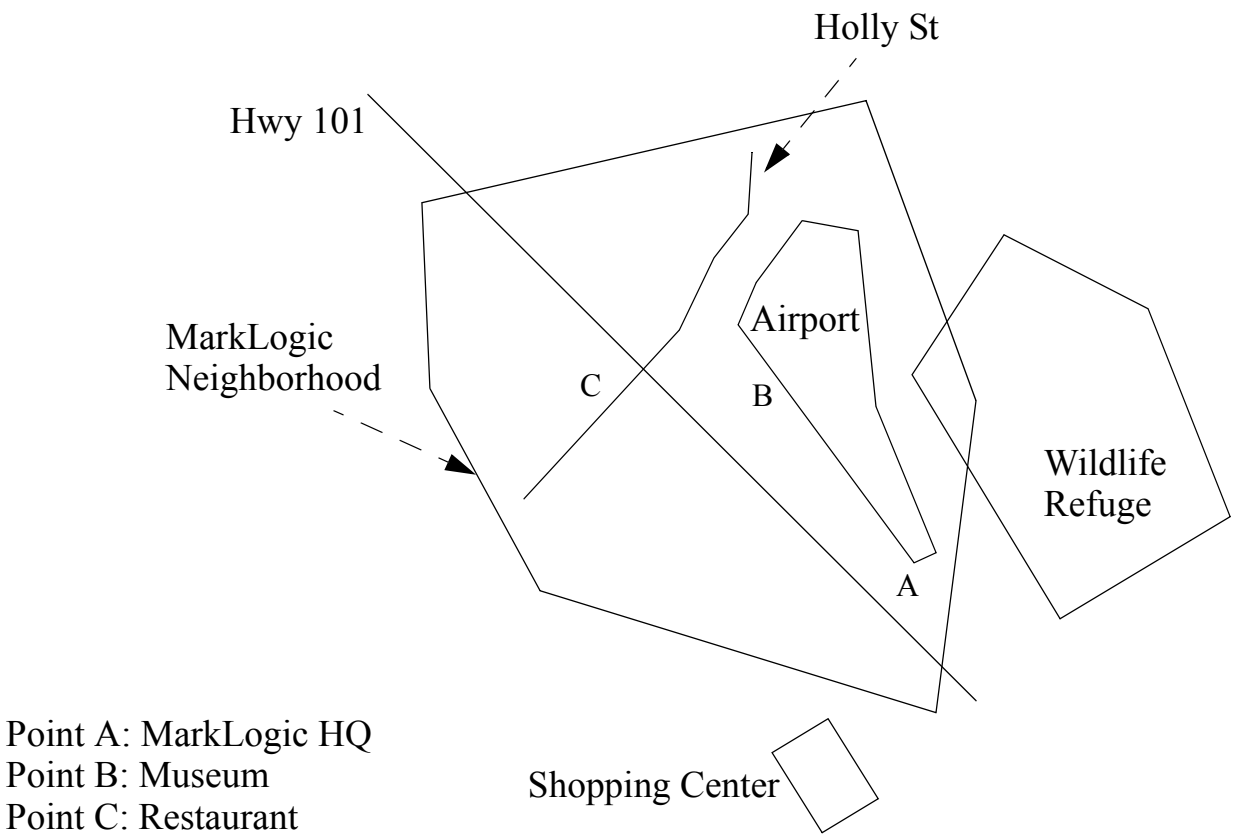
14.15 Preparing to Run the Examples

Use the instructions in this section to load the data and configure the indexes used in several examples in this chapter. The following topics are covered:

- Overview of the Sample Data
- Configuring the Indexes
- Creating the Input Data Files
- Loading the Sample Data

14.15.1 Overview of the Sample Data

The sample data contains points, linestrings, and polygons associated with landmarks near the MarkLogic headquarters. The following diagram approximates the relative positions of the features in the sample data.



This geospatial data is made available in two formats: KML (XML) and GeoJSON. Each document describes a single point or a region. The points and regions are the same in the two types of documents (XML and JSON). For example, the documents "/geo-examples/Airport.xml" and "/geo-examples/Airport.json" describe the same region.

The documents are added to the following collections to make it easy to select the type of data to work with just XML, just JSON, or both formats.

Document Set	Collections
KML	geo-examples, geo-xml-examples
GeoJSON	geo-examples, geo-json-examples

Each document uses the envelope pattern to encapsulate the original KML or GeoJSON region coordinates with a serialized cts region that is suitable for use with `cts:geospatial-region-query`. The unprocessed KML input is a single XML file that contains a series of KML `Placemark` elements. The following data snippet shows the structure of the raw input:

```
<kml xmlns="http://www.opengis.net/kml/2.2">
<Folder>
  <Placemark>
    <name>Hwy 101</name>
    <LineString>
      <extrude>0</extrude>
      <tessellate>1</tessellate>
      <coordinates>
        -122.2637558,37.5206187 -122.2428131,37.5020318
      </coordinates>
    </LineString>
  </Placemark>
  <Placemark>...</Placemark>
  ...
</Folder>
</kml>
```

The ingestion process splits the input into one document per `Placemark`. The envelope pattern is used to encapsulate the original `Placemark` with an equivalent serialized cts:region if the Placemark represents a non-point region. Region queries only operate on regions expressed as WKT or serialized cts:regions, so you cannot query the `Placemark` coordinates directly. (Point regions are left untranslated for convenience in demonstrating point queries; you could choose to treat them the same way.)

The following examples shows the final document format, with the `envelope` root element and the `cts-region` element created by the ingest transformation. If a `Placemark` represents a point, then no `cts-region` element is added because it is not needed.

```
<envelope>
  <cts-region>LINESTRING(-122.26376 37.520619,-122.24281
37.502032)</cts-region>
  <Placemark xmlns="http://www.opengis.net/kml/2.2">
    <name>Hwy 101</name>
```

```
      <LineString>
        <extrude>0</extrude>
        <tessellate>1</tessellate>
        <coordinates>
          -122.2637558,37.5206187 -122.2428131,37.5020318
        </coordinates>
      </LineString>
    </Placemark>
  <envelope>
```

The GeoJSON data receives similar treatment. The raw input is a feature collection. Ingestion creates one document per feature. The envelope pattern is used to encapsulate each feature with an equivalent serialized cts:region to facilitate queries. Point regions are not transformed.

For example, the raw GeoJSON input has the following structure:

```
{ "type": "FeatureCollection",
  "features": [
    { "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [-122.2465038,37.5073428]},
      "properties": {"name": "MarkLogic HQ"}
    },
    { "type": "Feature", ...},
    ...
  ]
}
```

Ingestion produces documents of the following form. For documents containing a region, the ingestion transformation adds the `envelope` wrapper and a `ctsRegion`. For documents containing a point, the ingestion transformation just adds the `envelope` wrapper.

```
{ "envelope": {
  "feature": {
    "type": "Feature",
    "geometry": {
      "type": "LineString",
      "coordinates": [...]
    }
    "properties": { "name": "Holly St" } },
  "ctsRegion": "LINESTRING(...))"
} }
```

You can thus use region queries on `/envelope/cts-region` (XML) or `/envelope/ctsRegion` (JSON) and point queries on `/envelope/kml:Placemark/Point/coordinates` (XML) or `geometry[type = 'Point']/array-node('coordinates')` (JSON).

14.15.2 Configuring the Indexes

This section walks through configuring a point index and a region index over the XML and JSON samples documents, for a total of 4 indexes. Separate indexes are used for each data set to showcase a variety of indexes and to make it easy to focus on one content type or the other.

Point indexes are optional for some types of geospatial point queries, but required for geospatial point range queries and lexicon operations. An index is usually recommended for best performance.

Geospatial region queries always required an index.

You can skip over indexes related to content that does not interest you. For example, you can skip the XML-related indexes if you are only interested in JSON. However, some examples in this chapter may not work properly without the related indexes.

Choose one of the following methods to create the indexes.

- Creating Indexes Using the Admin Interface
- Creating the Indexes with XQuery
- Creating the Indexes with JavaScript

You can also create indexes using the REST Management API. This method is not included here. For details, see the *MarkLogic REST API Reference*.

14.15.2.1Creating Indexes Using the Admin Interface

The following table summarizes the configuration characteristics of the indexes you will create in the Admin Interface. Use this information in Steps 5 and 7 of the following procedure. Use the default value for any characteristic not specified here.

Index Type	Characteristics
Geospatial Element Child Index (Point, XML)	Parent namespace URI: `http://www.opengis.net/kml/2.2` Parent localname: `Point` Child namespace URI: `http://www.opengis.net/kml/2.2` Child localname: `coordinates` Coordinate system: `wgs84` Point format: `long-lat-point`
Geospatial Path Index (Point, JSON)	Path expression: `geometry[type = 'Point']/array-node('coordinates')` Coordinate system: `wgs84` Point format: `long-lat-point`

Index Type	Characteristics
Geospatial Region Index (XML)	Path expression: `/envelope/cts-region` Coordinate system: `wgs84` Geohash precision: `2`
Geospatial Region Index (JSON)	Path expression: `/envelope/cts-region` Coordinate system: `wgs84` Geohash precision: `2`

Use the following procedure to create the geospatial indexes using the above configuration information. For more information about the Admin Interface, see *Administrator's Guide*.

1. Navigate to the Admin Interface in your browser. For example, navigate to `http://localhost:8001` if your MarkLogic installation is on localhost. Authenticate as a user with administrative privileges.

2. Click Databases in the tree menu on the left to expand the list of databases. The tree menu expands to display the available databases.

3. Click the name of the database for which you want to create an index. For example, click Documents. The tree menu expands to display the configuration categories for this database.

4. Click Geospatial Indexes icon in the tree menu, under the selected database.

5. To create a point index:

    a. Click Geospatial Point Indexes in the tree menu, under the selected database.

    b. Click the type of point index you want to create. For example, click Geospatial Element Child Indexes. The configuration page for this index type is displayed on the right.

6. To create a region index, click Geospatial Region Indexes in the tree menu, under the selected database. The configuration page for this index type is displayed on the right.

7. Click the Add tab at the top of the configuration page.

8. Fill in the configuration from the data in the table above.

9. Click OK to create the index.

10. Repeat from Step 5 until you have created all the required indexes.

14.15.2.2Creating the Indexes with XQuery

Use the procedure in this section to create the indexes using XQuery and Query Console. If you are not familiar with Query Console, see the *Query Console User Guide*. For equivalent JavaScript instructions, see "Creating the Indexes with JavaScript" on page 576.

The following procedure creates two point indexes and two region indexes.

1. Navigate to Query Console in your browser. For example, navigate to `http://localhost:8000` if your MarkLogic installation is on localhost.

2. Copy the following code into a new query tab in Query Console.

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

let $database := "Documents"
let $config := admin:get-configuration()
let $config :=         (: point index for XML docs :)
  admin:database-add-geospatial-element-child-index(
    $config, admin:database-get-id($config, $database),
    admin:database-geospatial-element-child-index(
      "http://www.opengis.net/kml/2.2", "Point",
      "http://www.opengis.net/kml/2.2", "coordinates",
      "wgs84", fn:false(), "long-lat-point", "reject")
  )
let $config :=         (: point index for JSON docs :)
  admin:database-add-geospatial-path-index(
    $config, admin:database-get-id($config, $database),
    admin:database-geospatial-path-index(
      "geometry[type = 'Point']/array-node('coordinates')",
      "wgs84", fn:false(), "long-lat-point", "reject")
  )
let $config :=         (: region index over XML docs :)
  admin:database-add-geospatial-region-path-index(
    $config, admin:database-get-id($config, $database),
    admin:database-geospatial-region-path-index(
      "/envelope/cts-region", "wgs84", 2, "reject")
  )
let $config :=         (: region index over JSON docs :)
  admin:database-add-geospatial-region-path-index(
    $config, admin:database-get-id($config, $database),
    admin:database-geospatial-region-path-index(
      "/envelope/ctsRegion", "wgs84", 2, "reject")
  )
(: create the configured indexes :)
return admin:save-configuration($config)
```

3. If you are not using the Documents database for your content database, modify the value of the `$database` variable.

4. Select XQuery in the Query Type dropdown if it is not already selected.

5. Click the Run button in Query Console to create the indexes.

The script produces no output when it is successful. You can use the Admin Interface to explore the geospatial indexes and confirm the indexes were created.

For the next step, see "Creating the Input Data Files" on page 577.

14.15.2.3Creating the Indexes with JavaScript

Use the procedure in this section to create the indexes using Server-Side JavaScript and Query Console. If you are not familiar with Query Console, see the *Query Console User Guide*. For equivalent XQuery instructions, see "Creating the XML Input File" on page 577.

The following procedure creates two point indexes and two region indexes.

1. Navigate to Query Console in your browser. For example, navigate to `http://localhost:8000` if your MarkLogic installation is on localhost.

2. Copy the following code into a new query tab in Query Console.

```
const admin = require('/MarkLogic/admin');

const database = 'Documents';
const config = admin.getConfiguration();

// Point index over the XML samples
config = admin.databaseAddGeospatialElementChildIndex(
    config, admin.databaseGetId(config, database),
    admin.databaseGeospatialElementChildIndex(
      'http://www.opengis.net/kml/2.2', 'Point',
      'http://www.opengis.net/kml/2.2', 'coordinates',
      'wgs84', false, 'long-lat-point', 'reject')
  );
// Point index over the JSON samples
config = admin.databaseAddGeospatialPathIndex(
    config, admin.databaseGetId(config, database),
    admin.databaseGeospatialPathIndex(
      'geometry[type = "Point"]/array-node("coordinates")',
      'wgs84', false, 'long-lat-point', 'reject')
  );
// Region index over the XML samples
config = admin.databaseAddGeospatialRegionPathIndex(
    config, admin.databaseGetId(config, database),
    admin.databaseGeospatialRegionPathIndex(
      '/envelope/cts-region', 'wgs84', 2, 'reject')
  );
// Region index over the JSON samples
config = admin.databaseAddGeospatialRegionPathIndex(
```

```
    config, admin.databaseGetId(config, database),
    admin.databaseGeospatialRegionPathIndex(
      '/envelope/ctsRegion', 'wgs84', 2, 'reject')
  );

// Create the configured indexes.
admin.saveConfiguration(config);
```

3. If you are not using the Documents database for your content database, modify the value of the `database` variable.

4. Select JavaScript in the Query Type dropdown if it is not already selected.

5. Click the Run button in Query Console to create the indexes.

The script produces no output when it is successful. You can use the Admin Interface to explore the geospatial indexes and confirm the indexes were created.

14.15.3 Creating the Input Data Files

Follow the instructions in this section to create two files containing the raw XML and JSON sample data. These files are used by the procedure in “Loading the Sample Data” on page 582. Create both files, unless you plan to skip the examples involving one document format.

- Creating the XML Input File
- Creating the JSON Input File

14.15.3.1Creating the XML Input File

Copy the following data to a file on the filesystem. Choose a location that is readable by your MarkLogic installation. You can use any file name, but the subsequent instructions assume “geo-examples.xml”.

Next, create the JSON data file following the instructions in “Creating the JSON Input File” on page 580.

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Placemark>
    <name>MarkLogic HQ</name>
    <description></description>
    <Point>
      <altitudeMode>clampedToGround</altitudeMode>
      <coordinates>-122.2465038,37.5073428</coordinates>
    </Point>
  </Placemark>
  <Placemark>
    <name>Restaurant</name>
```

```
  <description></description>
  <Point>
    <altitudeMode>clampedToGround</altitudeMode>
    <coordinates>-122.2581983,37.5128407</coordinates>
  </Point>
</Placemark>
<Placemark>
  <name>Hiller Aviation Museum</name>
  <description></description>
  <Point>
    <altitudeMode>clampedToGround</altitudeMode>
    <coordinates>-122.2527051,37.5128917</coordinates>
  </Point>
</Placemark>
<Placemark>
  <name>Hwy 101</name>
  <description>Length: 2.775 km (1.724 mi)</description>
  <visibility>1</visibility>
  <open>0</open>
  <LineString>
    <extrude>0</extrude>
    <tessellate>1</tessellate>
    <altitudeMode>clampedToGround</altitudeMode>
    <coordinates>
      -122.2637558,37.5206187 -122.2428131,37.5020318
    </coordinates>
  </LineString>
</Placemark>
<Placemark>
  <name>Holly St</name>
  <description>Length: 1.384 km (0.86 mi)</description>
  <visibility>1</visibility>
  <open>0</open>
  <LineString>
    <extrude>0</extrude>
    <tessellate>1</tessellate>
    <altitudeMode>clampedToGround</altitudeMode>
    <coordinates>
      -122.2598934,37.5096578 -122.2551727,37.5148321
      -122.2536278,37.5172148 -122.2523403,37.5185083
      -122.2520828,37.5202102
    </coordinates>
  </LineString>
</Placemark>
<Placemark>
  <name>Wildlife Refuge</name>
  <description>Length: 3.104 km (1.929 mi)</description>
  <visibility>1</visibility>
  <open>0</open>
  <Polygon>
    <extrude>0</extrude>
    <tessellate>1</tessellate>
    <altitudeMode>clampedToGround</altitudeMode>
    <outerBoundaryIs>
```

```
        <LinearRing>
          <coordinates>
            -122.2428131,37.5173510 -122.2468472,37.5131641
            -122.2422123,37.5069683 -122.2356892,37.5102365
            -122.2384787,37.5154788 -122.2428131,37.5173510
          </coordinates>
        </LinearRing>
      </outerBoundaryIs>
    </Polygon>
  </Placemark>
  <Placemark>
    <name>MarkLogic Neighborhood</name>
    <description>Length: 5.591 km (3.474 mi)</description>
    <visibility>1</visibility>
    <open>0</open>
    <styleUrl>#track</styleUrl>
    <Polygon>
      <extrude>0</extrude>
      <tessellate>1</tessellate>
      <altitudeMode>clampedToGround</altitudeMode>
      <outerBoundaryIs>
        <LinearRing>
          <coordinates>
            -122.2634554,37.5190870 -122.2480488,37.5212994
            -122.2446156,37.5122790 -122.2455597,37.5033596
            -122.2598076,37.5061853 -122.2633696,37.5134364
            -122.2634554,37.5190870
          </coordinates>
        </LinearRing>
      </outerBoundaryIs>
    </Polygon>
  </Placemark>
  <Placemark>
    <name>Shopping Center</name>
    <description>Length: 0.746 km (0.463 mi)</description>
    <visibility>1</visibility>
    <open>0</open>
    <styleUrl>#track</styleUrl>
    <Polygon>
      <extrude>0</extrude>
      <tessellate>1</tessellate>
      <altitudeMode>clampedToGround</altitudeMode>
      <outerBoundaryIs>
        <LinearRing>
          <coordinates>
            -122.2485638,37.5033937 -122.2465038,37.5015552
            -122.2478342,37.5003635 -122.2502375,37.5022020
            -122.2485638,37.5033937
          </coordinates>
        </LinearRing>
      </outerBoundaryIs>
    </Polygon>
  </Placemark>
  <Placemark>
```

```
    <name>Airport</name>
    <description>Length: 2.787 km (1.732 mi)</description>
    <visibility>1</visibility>
    <open>0</open>
    <styleUrl>#track</styleUrl>
    <Polygon>
      <extrude>0</extrude>
      <tessellate>1</tessellate>
      <altitudeMode>clampedToGround</altitudeMode>
      <outerBoundaryIs>
        <LinearRing>
          <coordinates>
            -122.2487354,37.5181339 -122.2481775,37.5121769
            -122.2457314,37.5086705 -122.2466755,37.5083982
            -122.2543144,37.5148321 -122.2537994,37.5157171
            -122.2509670,37.5177254 -122.2487354,37.5181339
          </coordinates>
        </LinearRing>
      </outerBoundaryIs>
    </Polygon>
  </Placemark>
</kml>
```

14.15.3.2Creating the JSON Input File

Copy the following data to a file on the filesystem. Choose a location that is readable by your MarkLogic installation. You can use any file name, but the subsequent instructions assume "geo-examples.json".

After saving the data to a file, load the sample data into the database using the instructions in "Loading the Sample Data" on page 582.

```
{ "type": "FeatureCollection",
  "features": [
    { "type": "Feature",
      "geometry": {"type": "Point", "coordinates":
[-122.2465038,37.5073428]},
      "properties": {"name": "MarkLogic HQ"}
    },
    { "type": "Feature",
      "geometry": {"type": "Point", "coordinates":
[-122.2581983,37.5128407]},
      "properties": {"name": "Restaurant"}
    },
    { "type": "Feature",
      "geometry": {"type": "Point", "coordinates":
[-122.2527051,37.5128917]},
      "properties": {"name": "Museum"}
    },
    { "type": "Feature",
      "geometry": {
        "type": "LineString",
```

```
      "coordinates": [
        [-122.2637558,37.5206187], [-122.2428131,37.5020318]
      ]
    },
    "properties": { "name": "Hwy 101" }
  },
  { "type": "Feature",
    "geometry": {
      "type": "LineString",
      "coordinates": [
        [-122.2598934,37.5096578], [-122.2551727,37.5148321],
        [-122.2536278,37.5172148], [-122.2523403,37.5185083],
        [-122.2520828,37.5202102]
      ]
    },
    "properties": { "name": "Holly St" }
  },
  { "type": "Feature",
    "geometry": {
      "type": "Polygon",
      "coordinates": [
        [
          [-122.2428131,37.5173510], [-122.2468472,37.5131641],
          [-122.2422123,37.5069683], [-122.2356892,37.5102365],
          [-122.2384787,37.5154788], [-122.2428131,37.5173510]
        ]
      ]
    },
    "properties": { "name": "Wildlife Refuge" }
  },
  { "type": "Feature",
    "geometry": {
      "type": "Polygon",
      "coordinates": [
        [
          [-122.2634554,37.5190870], [-122.2480488,37.5212994],
          [-122.2446156,37.5122790], [-122.2455597,37.5033596],
          [-122.2598076,37.5061853], [-122.2633696,37.5134364],
          [-122.2634554,37.5190870]
        ]
      ]
    },
    "properties": { "name": "MarkLogic Neighborhood" }
  },
  { "type": "Feature",
    "geometry": {
      "type": "Polygon",
      "coordinates": [
        [
          [-122.2485638,37.5033937], [-122.2465038,37.5015552],
          [-122.2478342,37.5003635], [-122.2502375,37.5022020],
          [-122.2485638,37.5033937]
        ]
      ]
```

```
        },
        "properties": { "name": "Shopping Center" }
      },
      { "type": "Feature",
        "geometry": {
          "type": "Polygon",
          "coordinates": [
            [
              [-122.2487354,37.5181339], [-122.2481775,37.5121769],
              [-122.2457314,37.5086705], [-122.2466755,37.5083982],
              [-122.2543144,37.5148321], [-122.2537994,37.5157171],
              [-122.2509670,37.5177254], [-122.2487354,37.5181339]
            ]
          ]
        },
        "properties": { "name": "Airport" }
      }
    ]
}
```

14.15.4 Loading the Sample Data

The procedures in this section load the raw data from “Creating the Input Data Files” on page 577 into documents of the form discussed in “Overview of the Sample Data” on page 569.

This section uses XQuery to load the XML documents and Server-Side JavaScript to load the JSON documents. You could use either language to load both, but the XML transformations flow more naturally in XQuery, while the JSON transformations flow more naturally in JavaScript.

You can load either or both data sets, but some examples in this chapter will not work if you do not load both. You do not need to be familiar with either XQuery or JavaScript to follow the instructions in this section.

- Loading the XML Sample Data
- Loading the JSON Sample Data

14.15.4.1 Loading the XML Sample Data

The procedure in this section uses XQuery to load the sample data because it is easier to do XML transformations using XQuery. You do not need to be familiar with XQuery to follow this procedure.

Before you begin, you should have completed the steps in “Creating the Input Data Files” on page 577.

1. Open the Query Console tool in your browser. For example, if MarkLogic is installed on localhost, navigate to the following URL: `http://localhost:8000`.

2. Copy the following query into a new query tab in Query Console:

```
xquery version "1.0-ml";
import module namespace geokml = "http://marklogic.com/geospatial/kml"
         at "/MarkLogic/geospatial/kml.xqy";
declare namespace kml="http://www.opengis.net/kml/2.2";

(: *** CHANGE THIS VAR VALUE TO MATCH YOUR ENV *** :)
declare variable $INPUT-FILE := "/my/dir/geo-examples.xml";

(: Convert the KML regions into cts regions :)
declare function local:region-convert(
  $nodes as node()*
) as cts:region*
{
  for $n in $nodes return
  typeswitch($n)
    case element(kml:Polygon) return geokml:parse-kml($n)
    case element(kml:LineString) return geokml:parse-kml($n)
    case element(kml:Point) return () (: return geokml:parse-kml($n) :)
    default return local:region-convert($n/node())
};

(: Create a doc for each KML Placemark, with a wrapper around
 : the KML that contains the cts region equiv of the KML region :)
let $file := xdmp:document-get($INPUT-FILE)
return
  for $place in $file//*:Placemark
  let $basename := fn:string-join(fn:tokenize($place/*:name, " "),"-")
  return xdmp:document-insert(
    fn:concat("/geo-examples/",$basename,".xml"),
    <envelope>{
       let $converted-region := local:region-convert($place)
       return
         if (fn:empty($converted-region))
         then ()
         else <cts-region>{$converted-region}</cts-region>
    }
    {$place}
    </envelope>,
    xdmp:default-permissions(), ("geo-xml-examples", "geo-examples"))
```

3. Modify the query to set the value of the `$INPUT-FILE` variable to the absolute path to the file containing the raw XML input data. This is the file you created in “Creating the Input Data Files” on page 577.

4. Choose XQuery in the Query Type dropdown list.

5. Choose the database into which you want to insert the documents in the Database dropdown list. For example, choose the Documents database.

6. Click the Run button to evaluate the query and create documents in the database.

7. Optionally, click the explorer icon to the right of the Database dropdown to explore the database contents and examine the new documents.

If the query is successful, the following documents are created. All the documents have a "/geo-examples/" directory prefix and are in collections named "geo-xml-examples" and "geo-examples".

- /geo-examples/Airport.xml
- /geo-examples/Holly-St.xml
- /geo-examples/Hwy-101.xml
- /geo-examples/MarkLogic-HQ.xml
- /geo-examples/MarkLogic-Neighborhood.xml
- /geo-examples/Museum.xml
- /geo-examples/Restaurant.xml
- /geo-examples/Shopping-Center.xml
- /geo-examples/Wildlife-Refuge.xml

For more information about the data, see "Overview of the Sample Data" on page 569.

Next, load the JSON sample documents using the instructions in "Loading the JSON Sample Data" on page 584.

14.15.4.2Loading the JSON Sample Data

The procedure in this section uses Server-Side JavaScript to load the sample data because it is easier to do JSON transformations using JavaScript. You do not need to be familiar with JavaScript to follow this procedure.

1. Open the Query Console tool in your browser. For example, if MarkLogic is installed on localhost, navigate to the following URL: `http://localhost:8000`.

2. Copy the following query into a new query tab in Query Console:

```
declareUpdate();
const geojson = require('/MarkLogic/geospatial/geojson');

// *** CHANGE FILE NAME TO MATCH YOUR ENV ***
const inputFilename = '/my/dir/geo-examples.json';

const rawData = fn.head(xdmp.documentGet(inputFilename)).toObject();
for (let feature of rawData.features) {
  // replace whitespace in feature name with a dash
  const uri = '/geo-examples/' +
    feature.properties.name.replace(/\s+/g,'-') + '.json';
```

```
    const newDoc = { envelope: {feature: feature} };
    if (feature.geometry.type != "Point") {
      newDoc.envelope.ctsRegion =
        fn.head(geojson.parseGeojson(feature.geometry));
    }
    xdmp.documentInsert(
      uri, newDoc,
      xdmp.defaultPermissions(),
      ['geo-json-examples', 'geo-examples']
    );
  }
```

3. Modify the query to set the value of the `$INPUT-FILE` variable to the absolute path to the file containing the raw input data. This is the file you created in "Creating the Input Data Files" on page 577.

4. Choose JavaScript in the Query Type dropdown list.

5. Choose the database into which you want to insert the documents in the Database dropdown list. For example, choose the Documents database.

6. Click the Run button to evaluate the query and create documents in the database.

7. Optionally, click the explorer icon to the right of the Database dropdown to explore the database contents and examine the new documents.

If the query is successful, the following documents are created. All the documents have a "/geo-examples/" directory prefix and are in collections named "geo-json-examples" and "geo-examples".

- /geo-examples/Airport.json
- /geo-examples/Holly-St.json
- /geo-examples/Hwy-101.json
- /geo-examples/MarkLogic-HQ.json
- /geo-examples/MarkLogic-Neighborhood.json
- /geo-examples/Museum.json
- /geo-examples/Restaurant.json
- /geo-examples/Shopping-Center.json
- /geo-examples/Wildlife-Refuge.json

For more information about the data, see "Overview of the Sample Data" on page 569.

Your database is now properly configured to run the examples in this chapter.

15.0 Entity Extraction and Enrichment

This chapter describes how to perform entity extraction or enrichment in MarkLogic Server. You can use these features to identify entities such as people and places in text, and then either add markup around the entities in your documents or extract a list of entities. You can use entity enrichment and extraction to classify documents and improve search accuracy.

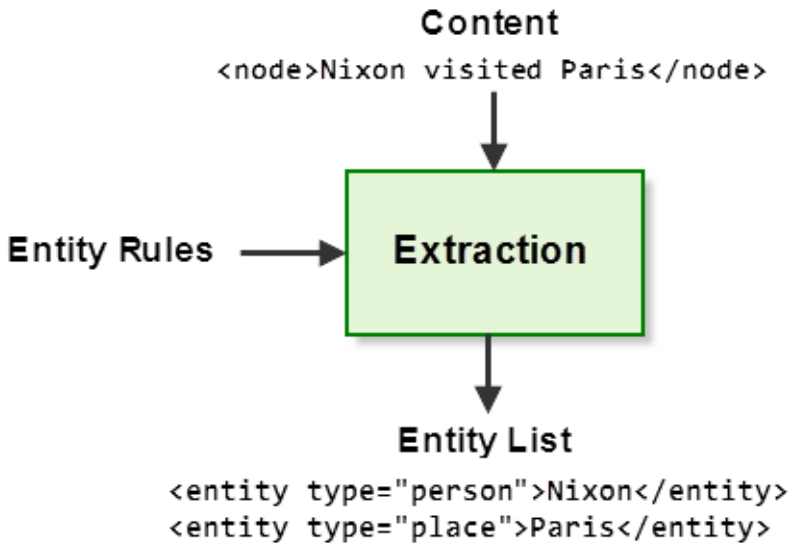
This chapter covers the following topics:

- Overview of Entity Extraction and Enrichment
- Understanding Dictionary-Based Extraction and Enrichment
- Creating an Entity Dictionary
- Dictionary-Based Entity Enrichment
- Dictionary-Based Entity Extraction
- Using an Entity Type Map for Extraction or Enrichment
- Overlapping Entity Match Handling
- Entity Identification Using Reverse Query
- Entity Enrichment Pipelines

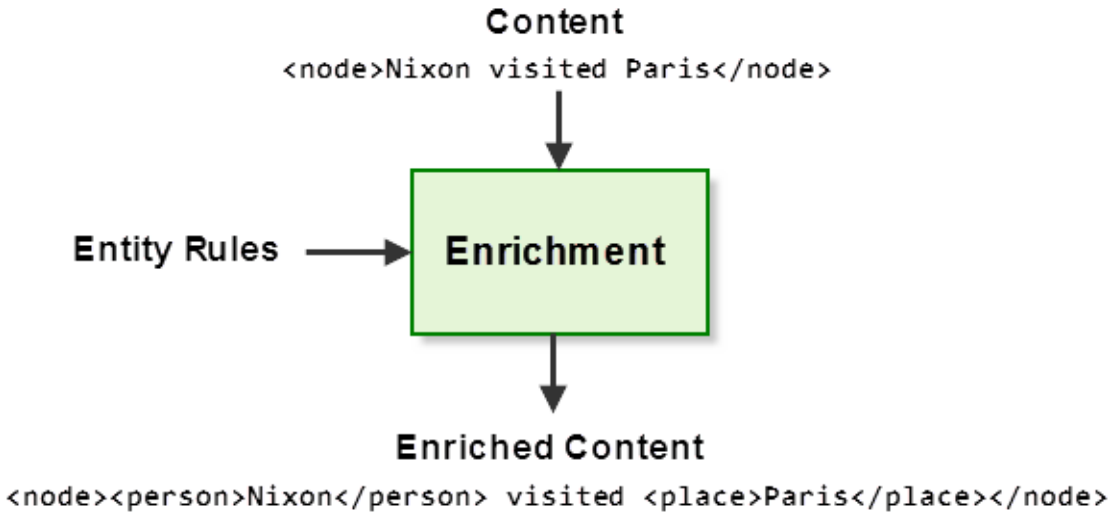
15.1 Overview of Entity Extraction and Enrichment

Entity extraction and entity enrichment are the process of identifying words or phrases that represent logical or business entities, and then either extracting a list of the entities from your content or enriching the content with information about the entities. Many industries have domain-specific entities that are useful to identify, such as extracting or marking up references to prescription drugs in patient history documents.

The following diagram illustrates the extraction process at a high level. Suppose you have entity rules that say the term “Nixon” represents a “person” entity and the term “Paris” represents a “place” entity. Then you could use the rules to extract a “person” and a “place” entity from the phrase “Nixon visited Paris” in an XML document:



Similarly, you could use the rules to enrich the phrase “Nixon visited Paris” with markup around the “person” and “place” entities:



MarkLogic provides out-of-the-box support for expressing entity rules as an opaque entity dictionary or a search query. MarkLogic APIs support both approaches. You can create dictionaries in various ways, including deriving one from a Simple Knowledge Organization System (SKOS) ontology.

You can also use third-party entity enrichment services by integrating them into a the Content Processing Framework (CPF) pipeline. MarkLogic includes some sample entity enrichment pipelines; for details, see "Entity Enrichment Pipelines" on page 634.

The following table can help you select the right extraction or enrichment approach for your application:

Use Case	Recommended Interface
Your entities can be identitied using simple string matching	Entity dictionaries and the entity enrichment and extraction APIs described in this chapter. For details, see "Understanding Dictionary-Based Extraction and Enrichment" on page 588.
Your entities can best be described by a cts query, or you require advanced string matching such as stemming or diacritic sensitivity	Reverse query and `cts:highlight` or `cts.highlight`. For more details, see "Entity Identification Using Reverse Query" on page 631.
You want to use a 3rd party entity extraction library	A Content Processing Framework (CPF) pipeline. For more details, see "Entity Enrichment Pipelines" on page 634.

15.2 Understanding Dictionary-Based Extraction and Enrichment

MarkLogic comes with a set of built-in and library module functions that support basic entity extraction and enrichment using entity dictionaries.

These interfaces can only be used when simple codepoint equality can be used to identify entity matches. You can control whether the comparison should be case sensitive, but you cannot use pattern matching, stemming, or diacritic sensitivity. If you need such features, use the technique described in "Entity Identification Using Reverse Query" on page 631.

You can create an entity dictionary from tab-delimited text, from a SKOS ontology, or from a set of entity objects created using `cts:entity` (XQuery) or `cts.entity` (JavaScript). For more details, see "Creating an Entity Dictionary" on page 589.

Once you create a dictionary that describes your entities, you can use it for operations such as the following:

- Extract entities from text as XML nodes constructed by MarkLogic or with custom markup. For details, see "Dictionary-Based Entity Extraction" on page 609.
- Enrich text with markup constructed by MarkLogic or with custom markup. For more details, see "Dictionary-Based Entity Enrichment" on page 598.

The following table summarizes the entity dictionary-based built-in and library functions. The functions in the `entity` library module provide an easy-to-use interface with limited customization options. The built-in `cts` functions provide finer control, at the cost of increased complexity.

Operation	XQuery	Server-Side JavaScript

Dictionary Management	`cts:entity-dictionary`
	`entity:skos-dictionary`
	`cts:entity`
	`cts:entity-dictionary-parse`
	`cts:get-entity-dictionary`
	`entity:dictionary-insert`
	`entity:dictionary-load`
		`cts.entityDictionary`
		`entity.skosDictionary`
		`cts.entity`
		`cts.entityDictionaryParse`
		`cts.getEntityDictionary`
		`entity.dictionaryInsert`
		`entity.dictionaryLoad`

Content Enrichment	`entity:enrich` `cts:entity-highlight`	`entity.enrich` `cts.entityHighlight`
Entity Extraction	`cts:entity-walk` `entity:extract`	`cts.entityWalk` `entity.extract`

15.3 Creating an Entity Dictionary

This section covers the following topics related to entity dictionary creation:

- Understanding Entity Dictionaries
- Creating a Dictionary Using Entity Constructors
- Creating a Dictionary From Text
- Creating a Dictionary From a SKOS Ontology
- Persisting or Retrieving an Entity Dictionary
- Serializing a Dictionary as Text

15.3.1 Understanding Entity Dictionaries

An entity dictionary is a set of entity definitions that specify the following characteristics of each entity:

- entity id - A unique id for the entity.
- normalized text - The normalized form of the entity.
- text - The word or phrase to match against this entry during entity extraction.
- entity type - The type of the entity.

You can create an entity dictionary in memory from the following sources.

- A Simple Knowledge Organization System (SKOS) ontology, stored in MarkLogic as a graph. For details, see "Creating a Dictionary From a SKOS Ontology" on page 592.
- Tab-delimited text. For details, see "Creating a Dictionary From Text" on page 591.
- A sequence of cts:entity (XQuery) or cts.entity (Server-Side JavaScript) objects. For details, see "Creating a Dictionary Using Entity Constructors" on page 591.

For efficient re-use, you should persist your entity dictionaries in MarkLogic. For details, see "Persisting or Retrieving an Entity Dictionary" on page 595.

When you use the dictionary-based APIs, such as `entity:enrich` or `entity.enrich`, matching is based on strict codepoint equality. You can only tailor the matching by specifying whether or not matches against a given dictionary should be case-insensitive. You cannot use an entity dictionary to find matches that depend on pattern matching, stemming, or other advanced algorithms.

A dictionary can contain multiple entries for the same entity id. For example, suppose former United States President Richard Nixon is a logical entity in your application domain. You might create dictionary entries that specify the phrases "Richard Nixon", "Richard M. Nixon", and "President Nixon" resolve to equivalent entities, with the same id, entity type, and normalized text. That is, you might create a dictionary that includes the following entries:

Id	Normalized Text	Text	Type
11208172	Nixon	Richard M. Nixon	person
11208172	Nixon	Richard Nixon	person
11208172	Nixon	President Nixon	person

Thus, entity extraction or enrichment can map any of the phrases "Richard M. Nixon", "Richard Nixon", and "President Nixon" to the "person" entity with the id 11208172.

If your dictionary includes entries whose text overlaps, then multiple entries can match overlapping portions of a text node. For example, if your dictionary contains both "President Nixon" and "Nixon Library", applying the dictionary to the phrase "President Nixon Library" results in overlapping entity matches. You can use the dictionary creation options "allow-overlaps" and "remove-overlaps" to affect overlap handling. The default behavior is "allow-overlaps". For more details, see "Overlapping Entity Match Handling" on page 627.

15.3.2 Creating a Dictionary Using Entity Constructors

In XQuery, you can use `cts:entity` to construct opaque dictionary entry objects, and then use `cts:entity-dictionary` to create an in-memory entity dictionary from them.

In Server-Side JavaScript, you can use `cts.entity` to construct opaque dictionary entry objects, and then use `cts.entityDictionary` to create an in-memory entity dictionary from them.

For example, the following example construct an in-memory entity dictionary contianing four entries:

Language	Example

XQuery	`cts:entity-dictionary((`
	`  cts:entity("11208172", "Nixon", "Nixon", "person"),`
	`  cts:entity("11208172", "Nixon", "Richard Nixon", "person"),`
	`  cts:entity("09145751", "Paris", "Paris", "district:town"),`
	`  cts:entity("09500217", "Paris", "Paris", "mythical being")`
	`))`

Server-Side JavaScript	`const dictionary = cts.entityDictionary([`
	`  cts.entity('11208172', 'Nixon', 'Nixon', 'person'),`
	`  cts.entity('11208172', 'Nixon', 'Richard Nixon', 'person'),`
	`  cts.entity('09145751', 'Paris', 'Paris', 'district:town'),`
	`  cts.entity('09500217', 'Paris', 'Paris', 'mythical being')`
	`]);`

You can persist the dictionary in MarkLogic using `cts:dictionary-insert` (XQuery) or `cts.dictionaryInsert` (JavaScript). For details, see "Persisting or Retrieving an Entity Dictionary" on page 595.

15.3.3 Creating a Dictionary From Text

You can construct an entity dictionary from specially formatted text using `cts:entity-dictionary-parse` (XQuery) or `cts.entityDictionaryParse` (JavaScript). The input must be strings containing dictionary entry lines of the following form. Dictionary entries must be newline separated, and the fields of entry must be tab separated.

```
id   normalizedText   text    entityType
```

This is the same format produced when you serialize a dictionary; for details, see "Serializing a Dictionary as Text" on page 597.

For example, suppose you have a file "/my/ent-dict.txt" on the filesystem containing the following lines of tab-delimited text:

```
11208172        Nixon   Nixon   person:head of state
11208172        Nixon   Richard Nixon   person:head of state
09145751        Paris   Paris   administrative district:town
09500217        Paris   Paris   imaginary being:mythical being
```

Then the following example code creates an in-memory entity dictionary from the file contents.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`cts:entity-dictionary-parse(`
	`  xdmp:document-get('/my/ent-dict.txt'))`

Server-Side JavaScript	`cts.entityDictionaryParse(` `  xdmp.documentGet('/my/ent-dict.txt'));`

You can persist such an in-memory dictionary in MarkLogic using `entity:dictionary-insert` (XQuery) or `entity.dictionaryInsert` (JavaScript). You can also load the text representation of an entity dictionary directly into MarkLogic using `entity:dictionary-load` (XQuery) or `entity.dictionaryLoad` (JavaScript). For details, see "Persisting or Retrieving an Entity Dictionary" on page 595.

15.3.4 Creating a Dictionary From a SKOS Ontology

You can create an entity dictionary from a Simple Knowledge Organization System (SKOS) ontology. A SKOS is a semantic graph composed of RDF triples; for details, see https://www.w3.org/TR/skos-primer/. SKOS ontologies are available for many application domains. A SKOS ontology includes exactly the kind of information used in a MarkLogic entity dictionary entry: An entity ID, with one or more matching terms, a normalized form, and an entity type.

Use the following steps to create an entity dictionary from a SKOS ontology:

1. Insert the graph representing the ontology into MarkLogic, as described in Loading Semantic Triples in the *Semantic Graph Developer's Guide*.

2. Use the library function `entity:skos-dictionary` (XQuery) or `entity.skosDictionary` (JavaScript) to create a dictionary from the graph.

A dictionary entry is created for each `skos:Concept` in the graph, where `skos` is shorthand for the namespace http://www.w3.org/2004/02/skos/core#. Dictionary entries will not be extracted for triples that use any other SKOS namespace.

The following table provides an overview of the mapping from SKOS properties to dictionary entry attributes. For more details on the mapping, see the function reference for `entity:skos-dictionary` (XQuery) or `entity.skosDictionary` (JavaScript).

Entity Component	SKOS Source
id	The concept IRI.
normalized text	The `skos:prefLabel`.
text	The `skos:prefLabel`, plus any additional lablels (`skos:altLabel`, `skos:hiddenLabel`).
entity type	If the concept is in a `skos:ConceptScheme`, the `rdfs:label` or `dc:title` from the concept scheme; otherwise, the graph URI.

For example, suppose you have a file on the filesystem with path "/examples/canal.rdf" that contains the following simplified SKOS ontology:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:skos="http://www.w3.org/2004/02/skos/core#">
  <skos:Concept rdf:about="http://www.my.com/#canals">
    <skos:definition>A feature type category for places such as
      the Erie Canal</skos:definition>
    <skos:prefLabel>canals</skos:prefLabel>
    <skos:altLabel>canal bends</skos:altLabel>
    <skos:altLabel>canalized streams</skos:altLabel>
    <skos:altLabel>ditch mouths</skos:altLabel>
    <skos:altLabel>ditches</skos:altLabel>
    <skos:altLabel>drainage canals</skos:altLabel>
    <skos:altLabel>drainage ditches</skos:altLabel>
    <skos:broader
      rdf:resource="http://www.my.com/#hydrographic%20structures"/>
    <skos:related rdf:resource="http://www.my.com/#channels"/>
    <skos:related rdf:resource="http://www.my.com/#locks"/>
    <skos:related
      rdf:resource="http://www.my.com/#transportation%20features"/>
    <skos:related rdf:resource="http://www.my.com/#tunnels"/>
    <skos:scopeNote>Manmade waterway used by watercraft or for
      drainage, irrigation, mining, or water power</skos:scopeNote>
  </skos:Concept>
</rdf:RDF>
```

Then you can load the ontology into a graph in MarkLogic with the URI "http://marklogic.com/examples/canal" as follows:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace sem = "http://marklogic.com/semantics"`
	`  at "/MarkLogic/semantics.xqy";`
	
	`sem:graph-insert(`
	`  sem:iri("http://marklogic.com/examples/canal"),`
	`  sem:rdf-get("/examples/canal.rdf", ("rdfxml"))`
	`)`

Server-Side JavaScript	`declareUpdate();`
	`const sem = require('/MarkLogic/semantics');`
	
	`sem.graphInsert(`
	`  sem.iri('http://marklogic.com/examples/canal'),`
	`  sem.rdfGet('/examples/canal.rdf', ['rdfxml']));`

Now, you can create an entity dictionary from the graph and save it in MarkLogic as shown by the following example. Note that your dictionary URI should not be the same as the graph URI. To learn more about creating graphs in MarkLogic, see *Semantic Graph Developer's Guide*.

Language	Example

XQuery	`import module namespace entity="http://marklogic.com/entity"`
	`  at "/MarkLogic/entity.xqy";`
	
	`entity:dictionary-insert("/ontology/canal",`
	`  entity:skos-dictionary(`
	`  "http://marklogic.com/examples/canal",`
	`  "en", "case-insensitive")`
	`)`

Server-Side JavaScript	`'use strict';`
	`declareUpdate();`
	`const entity = require('/MarkLogic/entity');`
	
	`entity.dictionaryInsert('/ontology/canal',`
	`  entity.skosDictionary(`
	`  'http://marklogic.com/examples/canal',`
	`  'en', ['case-insensitive'])`
	`);`

The resulting entity dictionary contains the following entries. All the terms share the same entity type because the trivial example onotology defines only one concept.

ID	Norm. Text	Text	Entity Type
http://www.my.com/#canal	canals	canal bends	http://marklogic.com/examples/canal
http://www.my.com/#canals	canals	canalized streams	http://marklogic.com/examples/canal
http://www.my.com/#canals	canals	canals	http://marklogic.com/examples/canal
http://www.my.com/#canals	canals	ditch mouths	http://marklogic.com/examples/canal
http://www.my.com/#canals	canals	ditch	http://marklogic.com/examples/canal
http://www.my.com/#canals	canals	ditches	http://marklogic.com/examples/canal
http://www.my.com/#canals	canals	drainage canals	http://marklogic.com/examples/canal
http://www.my.com/#canals	canals	drainage ditches	http://marklogic.com/examples/canal

15.3.5 Persisting or Retrieving an Entity Dictionary

For best performance, large dictionaries and dictionaries you use frequently should be stored in MarkLogic. To persist a dictionary in the database, use the following functions:

- `entity:dictionary-insert` (XQuery) or `entity.dictionaryInsert` (JavaScript): Save an in-memory dictionary in the database.
- `entity:dictionary-load` (XQuery) or `entity.dictionaryLoad` (JavaScript): Read in dictionary entries from a tab-delimited text file and save the results in the database. For format details, see “Creating a Dictionary From Text” on page 591.

For example, the following code creates an in-memory dictionary using entity constructors, and then saves it in the database:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace entity="http://marklogic.com/entity"`
	`  at "/MarkLogic/entity.xqy";`
	`let $dictionary := cts:entity-dictionary((`
	`  cts:entity("11208172", "Nixon", "Nixon", "person"),`
	`  cts:entity("11208172", "Nixon", "Richard Nixon", "person")`
	`))`
	`return entity:dictionary-insert("/dict/people", $dictionary)`

Server-Side JavaScript	`'use strict';`
	`declareUpdate();`
	`const entity = require('/MarkLogic/entity');`
	
	`const dictionary = cts.entityDictionary([`
	`  cts.entity('11208172', 'Nixon', 'Nixon', 'person'),`
	`  cts.entity('11208172', 'Nixon', 'Richard Nixon', 'person'),`
	`]);`
	
	`entity.dictionaryInsert('/dict/people', dictionary);`

The following example loads a properly serialized dictionary on the filesystem directly into MarkLogic. The expected format is the same as that described in “Creating a Dictionary From Text” on page 591 and “Serializing a Dictionary as Text” on page 597.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace entity="http://marklogic.com/entity"`
	`  at "/MarkLogic/entity.xqy";`
	
	`entity:dictionary-load(`
	`  '/path/to/my/dict.txt', '/dict/people');`

Server-Side JavaScript	`declareUpdate();`
	`const entity = require('/MarkLogic/entity');`
	
	`entity.dictionaryLoad(`
	`  '/path/to/my/dict.txt', '/dict/people');`

To retrieve a dictionary stored in MarkLogic, use `cts:entity-dictionary-get` (XQuery) or `cts.entityDictionaryGet` (JavaScript).

For example:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace entity="http://marklogic.com/entity"`
	`  at "/MarkLogic/entity.xqy";`
	`let $dictionary :=`
	`cts:entity-dictionary-get("/ontology/fibo/vocabulary")`
	`return dictionary`

Server-Side JavaScript	`'use strict';`
	`const entity = require('/MarkLogic/entity');`
	`const dictionary =`
	`cts.entityDictionaryGet('/ontology/fibo/vocabulary');`
	`dictionary;`

15.3.6 Serializing a Dictionary as Text

You can serialize an entity dictionary as text, suitable for exporting to a file. You can use `cts:entity-dictionary-parse` or `cts.entityDictionaryParse` to re-create a cts:entity-dictionary object from the serialization.

The following example serializes an in-memory dictionary:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`let $dictionary := cts:entity-dictionary((`
	`  cts:entity("11208172", "Nixon", "Nixon", "person"),`
	`  cts:entity("11208172", "Nixon", "Richard Nixon", "person"),`
	`  cts:entity("09145751", "Paris", "Paris", "district:town"),`
	`  cts:entity("09500217", "Paris", 'Paris', "mythical being")`
	`), ("remove-overlaps", "case-insensitive"))`
	`return xdmp:quote($dictionary)`

Server-Side JavaScript	`const dictionary = cts.entityDictionary([`
	`  cts.entity('11208172', 'Nixon', 'Nixon', 'person'),`
	`  cts.entity('11208172', 'Nixon', 'Richard Nixon', 'person'),`
	`  cts.entity('09145751', 'Paris', 'Paris', 'district:town'),`
	`  cts.entity('09500217', 'Paris', 'Paris', 'mythical being')],`
	`  ['remove-overlaps', 'case-insensitive']);`
	`xdmp.quote(dictionary);`

This example produces the following output. Each serialized dictionary entry is separated by a newline. Each field within an entry is separated by a TAB character. The first line, with the "##" prefix, encodes the options used to create the dictionary.

```
## remove-overlaps case-insensitive
11208172    Nixon    Nixon    person
11208172    Nixon    Richard Nixon    person
09145751    Paris    Paris    district:town
09500217    Paris    Paris    mythical being
```

15.4 Dictionary-Based Entity Enrichment

Entity enrichment is the process of adding markup to a document that identifies the occurrence of entities in the text. MarkLogic provides a set up of APIs that enable you to define the set of possible entities in one or more entity dictionaries, and then tag matching entities in your XML documents. To generate a list of entities found in a document rather than add enrichment, use entity extraction; for details, see "Dictionary-Based Entity Extraction" on page 609.

This section covers the following topics related to using the dictionary-based APIs for entity enrichment:

- API Summary
- Using entity:enrich or entity.enrich
- Using cts:entity-highlight or cts.entityHighlight
- XQuery Example: entity:enrich
- XQuery Example: cts:entity-highlight
- JavaScript Example: entity.enrich
- JavaScript Example: cts.entityHighlight

15.4.1 API Summary

The following table summarizes the dictionary-based APIs available for adding entity enrichment to your XML documents. These APIs require you to create one or more entity dictionaries, as described in "Creating an Entity Dictionary" on page 589.

Function	Description
`entity:enrich` (XQuery) `entity.enrich` (JavaScript)	Enclose words and phrases matching dictionary entries in a wrapper element decorated with the entity type. Some customization is available.
`cts:entity-highlight` (XQuery) `cts.entityHighlight` (JavaScript)	Replace words and phrases matching dictionary entries with content of your choosing.

The `enrich` function is the easiest to use, and suitable for many applications. Use `cts:entity-highlight` or `cts.entityHighlight` if you require fine-grained control over the enrichment.

15.4.2 Using entity:enrich or entity.enrich

When you call `entity:enrich` or `entity.enrich` with just an input node and one or more dictionaries, MarkLogic wraps matched text in an `<entity/>` element that has a `type` attribute whose value is the entity type from the matching dictionary entry.

For example, if you call `enrich` in the form shown below:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace entity="http://marklogic.com/entity"`
	`  at "/MarkLogic/entity.xqy";`
	
	`let $some-dictionary := ...`
	`entity:enrich($some-node, $some-dictionary)`

Server-Side JavaScript	`const entity = require('/MarkLogic/entity');`
	
	`const someDictionary = ...;`
	`entity.enrich(someNode, someDictionary);`

Then the enrichment uses a wrapper such as the following:

```
<e:entity xmlns:e="http://marklogic.com/entity">
  type="person:head of state">Nixon</e:entity>
```

For a complete example, see “XQuery Example: entity:enrich” on page 601 or “JavaScript Example: entity.enrich” on page 605.

You can further tailor the enrichment as follows:

- Use the “full” option to decorate the wrapper element with additional information from the matched dictionary entry, such as the entity id and normalized text.
- Pass in a mapping between entity type names and element QNames to change the QName of the wrapper element. For details, see “Using an Entity Type Map for Extraction or Enrichment” on page 622.

If you pass multiple dictionaries to `enrich`, then the dictionaries are applied in turn, in the order provided.

For example, suppose you have an entity dictionary that defines the word “Nixon” as an entity of type “person:head of state”. Further, suppose you define a mapping from “person:head of state” to the QName “entity:vip”. Then, the following table summarizes different forms of enrichment available using `entity:enrich` or `entity.enrich`.

Use Case	Example
original text	`Nixon`
default markup	`<e:entity xmlns:e="http://marklogic.com/entity">` `  type="person:head of state">Nixon</e:entity>`

“full” option to add additional entity attributes	`<e:entity xmlns:e="http://marklogic.com/entity"`
	`  id="11208172" norm="Nixon"`
	`  type="person:head of state">Nixon</e:entity>`

entity type map to change wrapper QName	`<entity:vip>Nixon</entity:vip>`
map plus “full” option	`<entity:vip id="11208172" norm="Nixon">Nixon</entity:vip>`

If this level of customization does not meet the needs of your application, see “Using cts:entity-highlight or cts.entityHighlight” on page 600.

15.4.3 Using cts:entity-highlight or cts.entityHighlight

The XQuery function `cts:entity-highlight` and the JavaScript function `cts.entityHighlight` give you complete control over construction of enriched content, at the cost of somewhat greater complexity.

The `cts:entity-highlight` XQuery function accepts a block of inline XQuery code that gets evaluated for each entity match. Use this code block to construct your enrichment. Nodes returned by your inline code are inserted into the final result.

The `cts.entityHighlight` JavaScript function accepts a callback function as a parameter. Your function gets called for each entity match. Your callback adds enriched content to the final result by interacting with the `NodeBuilder` passed in by MarkLogic.

In both XQuery and JavaScript, details about the matching dictionary entry are made available to your generator code. For details, see the function reference documentation for `cts:entity-highlight` and `cts.entityHighlight`.

For example, the following snippets use the entity type and matched text information provided by Marklogic to construct enriched replacement content for the matched text.

Language	Example

XQuery	`cts:entity-highlight(`
	`  $input-node,`
	`  (element { fn:replace($cts:entity-type, ":| ", "-") }`
	`  { $cts:text }),`
	`  $dictionary)`

Server-Side JavaScript	`cts.entityHighlight(inputNode,`
	`  function(builder, entityType, text, normText,`
	`  entityId, node, start) {`
	`  builder.addElement(`
	`  fn.replace(entityType, ':| ', '-'), text);`
	`  },`
	`  resultNodeBuilder, dictionary`
	`);`

For a complete example, see “XQuery Example: cts:entity-highlight” on page 603 or “JavaScript Example: cts.entityHighlight” on page 607.

15.4.4 XQuery Example: entity:enrich

This example uses `entity:enrich` to add entity-based markup to XML content, as described in “Using entity:enrich or entity.enrich” on page 599. The example demonstrates the use of various customization features of `entity:enrich`.

The example uses an in-memory dictionary that defines the following:

- An entity of type “person:head of state” for various phrases that describe former United States President Richard Nixon.
- Several different entity types for the word “Paris”.

The example uses the dictionary add enrichment around the phrases “Nixon” and “Paris” in the following input node:

```
<node>Nixon visited Paris</node>
```

The example uses an in-memory dictionary and input data for the sake of self-containment. In a real application, you would usually store the dictionary in MarkLogic, as described in “Persisting or Retrieving an Entity Dictionary” on page 595. The input node can also be an XML document or other node in MarkLogic.

Copy and paste the following code into Query Console, set the Query Type to XQuery, and run it. If you are unfamiliar with Query Console, see the *Query Console User Guide*.

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

let $dictionary := cts:entity-dictionary((
  cts:entity("11208172", "Nixon", "Nixon", "person:head of state"),
  cts:entity("11208172", "Nixon", "Richard Nixon", "person:head of state"),
  cts:entity("11208172", "Nixon", "Richard M. Nixon", "person:head of state"),
  cts:entity("11208172", "Nixon", "Richard Milhous Nixon",
             "person:head of state"),
  cts:entity("11208172", "Nixon", "President Nixon", "person:head of state"),
  cts:entity("08932568", "Paris", "Paris",
             "administrative district:national capital"),
  cts:entity("09145751", "Paris", "Paris", "administrative district:town"),
  cts:entity("09500217", "Paris", "Paris", "imaginary being:mythical being")
))
let $mapping :=
  map:new((
    map:entry("",xs:QName("entity:entity")),
    map:entry("administrative district",xs:QName("entity:gpe")),
    map:entry("person",
              map:map() => map:with("", xs:QName("entity:location"))
                        => map:with("head of state",
                                    xs:QName("entity:vip")))
    ))
let $input-node := <node>Nixon visited Paris</node>

return (
  "------- default -------",
  entity:enrich($input-node, $dictionary),
  "------- full option -------",
  entity:enrich($input-node, $dictionary, "full"),
  "------- mapping -------",
  entity:enrich($input-node, $dictionary, (), $mapping),
  "------- full + mapping -------",
  entity:enrich($input-node, $dictionary, "full", $mapping)
)
```

You should see output similar to the following. (Whitespace has been added to improve readability. The enrichment does not introduce new whitespace or comments.)

```
------- default -------
<node xmlns:e="http://marklogic.com/entity">
  <e:entity type="person:head of state">Nixon</e:entity>
  visited
  <e:entity type="administrative district:national capital">Paris</e:entity>
</node>

------- full option -------
<node xmlns:e="http://marklogic.com/entity">
  <e:entity id="11208172" norm="Nixon"
    type="person:head of state">Nixon</e:entity>
  visited <e:entity id="08932568" norm="Paris"
     type="administrative district:national capital">Paris
  </e:entity>
</node>

------- mapping -------
<node xmlns:entity="http://marklogic.com/entity">
  <entity:vip>Nixon</entity:vip>
  visited
  <entity:gpe>Paris</entity:gpe>
</node>

------- mapping -------
<node xmlns:entity="http://marklogic.com/entity">
  <entity:vip id="11208172" norm="Nixon">Nixon</entity:vip>
  visited
  <entity:gpe id="08932568" norm="Paris">Paris</entity:gpe>
</node>
```

15.4.5 XQuery Example: cts:entity-highlight

This example illustrates how you can use `cts:entity-highlight` to enrich content when you need more control than that provided by `entity:enrich`. For details, see “Using cts:entity-highlight or cts.entityHighlight” on page 600.

The example uses an in-memory dictionary that defines the following:

- An entity of type “person:head of state” for various phrases that describe former United States President Richard Nixon.
- Several different entity types for the word “Paris”.

The example uses the dictionary add enrichment around the phrases “Nixon” and “Paris” in the following input node:

```
<node>Nixon visited Paris</node>
```

The example uses an in-memory dictionary and input data for the sake of self-containment. In a real application, you would usually store the dictionary in MarkLogic, as described in “Persisting or Retrieving an Entity Dictionary” on page 595. The input node can also be an XML document or other node in MarkLogic.

Copy and paste the following code into Query Console, set the Query Type to XQuery, and run it. If you are unfamiliar with Query Console, see the *Query Console User Guide*.

```
xquery version "1.0-ml";

let $dictionary := cts:entity-dictionary((
  cts:entity("11208172","Nixon","Nixon","person:head of state"),
  cts:entity("11208172","Nixon","Richard Nixon","person:head of state"),
  cts:entity("11208172","Nixon","Richard M. Nixon","person:head of state"),
  cts:entity("11208172","Nixon","Richard Milhous Nixon",
             "person:head of state"),
  cts:entity("11208172","Nixon","President Nixon","person:head of state"),
  cts:entity("08932568","Paris","Paris",
             "administrative district:national capital"),
  cts:entity("09145751","Paris","Paris","administrative district:town"),
  cts:entity("09500217","Paris","Paris","imaginary being:mythical being")
))
let $input-node := <node>Nixon visited Paris</node>
return cts:entity-highlight(
  $input-node,
  (if ($cts:text ne "")
    then element { fn:replace($cts:entity-type, ":| ", "-") } { $cts:text }
    else ()),
  $dictionary)
```

The example produces the following output. Whitespace has been added to improve readability. The enrichment does not introduce new whitespace.

```
<node>
  <person-head-of-state>Nixon</person-head-of-state>
  visited
  <administrative-district-national-capital>Paris</administrative-district-nat
ional-capital>
</node>
```

Each time `cts:entity-highlight` identifies a word or phrase that matches a dictionary entry, it evaluates the expression passed in as the second parameter. The example code simiply generates an entity wrapper that uses the entity type name as the wrapper element QName, after replacing any occurrences of ":" or " " with a dash ("-").

```
element { fn:replace($cts:entity-type, ":| ", "-") } { $cts:text }
```

The special variables `$cts:text` and `$cts:entity-type` are populated with information from the matching dictionary entry. Your code has access to other data from the matching dictionary entry, such as the normalized text (`$cts:entity-id`) and the entity id (`$cts:entity-id`). For details, see the function reference for `cts:entity-highlight`.

If text matches more than one dictionary entry, your code is evaluated for each match, but `$cts:text` will be empty for all but the first match. The example as given tests for an empty `$cts:text` and only generates replacement content for the first match.

```
if ($cts:text ne "")
then element { fn:replace($cts:entity-type, ":| ", "-") } { $cts:text }
else ()
```

For example, the term “Paris” matches 3 entries in the dictionary. If you remove the empty string test, as follows:

```
cts:entity-highlight(
  $input-node,
  (element { fn:replace($cts:entity-type, ":| ", "-") } { $cts:text }),
  $dictionary)
```

Then the example produces the following element related to the term “Paris”. The same wrapper is generated for the first match, but the subsequent matches insert an entity tag with no text content.

```
<administrative-district-national-capital>Paris</administrative-district-natio
nal-capital>
<administrative-district-town/>
<imaginary-being-mythical-being/>
```

15.4.6 JavaScript Example: entity.enrich

This example uses `entity.enrich` to add entity-based markup to XML content, as described in “Using entity:enrich or entity.enrich” on page 599. The example demonstrates the use of various customization features of `entity.enrich`.

The example uses an in-memory dictionary that defines the following:

- An entity of type “person:head of state” for various phrases that describe former United States President Richard Nixon.
- Several different entity types for the word “Paris”.

The example uses the dictionary add enrichment around the phrases “Nixon” and “Paris” in the following input node:

```
<node>Nixon visited Paris</node>
```

The example uses an in-memory dictionary and input data for the sake of self-containment. In a real application, you would usually store the dictionary in MarkLogic, as described in "Persisting or Retrieving an Entity Dictionary" on page 595. The input node can also be an XML document or other node in MarkLogic.

Copy and paste the following code into Query Console, set the Query Type to JavaScript, and run it. If you are unfamiliar with Query Console, see the *Query Console User Guide*.

```
'use strict';
const entity = require('/MarkLogic/entity');

// NOTE: The fields of each string below must be TAB separated.
const dictionary = cts.entityDictionary([
  cts.entity('11208172','Nixon','Nixon','person:head of state'),
  cts.entity('11208172','Nixon','Richard Nixon','person:head of state'),
  cts.entity('11208172','Nixon','Richard M. Nixon','person:head of state'),
  cts.entity('11208172','Nixon','Richard Milhous Nixon',
             'person:head of state'),
  cts.entity('11208172','Nixon','President Nixon','person:head of state'),
  cts.entity('08932568','Paris','Paris',
             'administrative district:national capital'),
  cts.entity('09145751','Paris','Paris','administrative district:town'),
  cts.entity('09500217','Paris','Paris','imaginary being:mythical being')
]);
const mapping = {
  '' : fn.QName('http://marklogic.com/entity', 'entity:entity'),
  'administrative district': fn.QName('http://marklogic.com/entity',
                                      'entity:gpe'),
  person: {
    '': fn.QName('http://marklogic.com/entity', 'entity:person'),
    'head of state': fn.QName('http://marklogic.com/entity',
                              'entity:vip')
  }
};
const inputNode = new NodeBuilder()
                    .addElement('node', 'Nixon visited Paris')
                    .toNode();
const result = [
  entity.enrich(inputNode, dictionary),
  entity.enrich(inputNode, dictionary, ['full']),
  entity.enrich(inputNode, dictionary, null, mapping),
  entity.enrich(inputNode, dictionary, ['full'], mapping)
];
result;
```

The example code generates XML of the forms shown below. Whitespace and comments have been added to improve readability. The enrichment does not introduce new whitespace or comments. (Due to the way Query Console formats XML for display, the generated XML appears as strings in the Query Console results window. In fact, they are XML element nodes.)

```
<!-- default enrichment -->
<node xmlns:e="http://marklogic.com/entity">
```

```
  <e:entity type="person:head of state">Nixon</e:entity>
  visited
  <e:entity type="administrative district:national capital">Paris</e:entity>
</node>

<!-- using the "full" option adds @id and @norm data -->
<node xmlns:e="http://marklogic.com/entity">
  <e:entity id="11208172" norm="Nixon"
    type="person:head of state">Nixon</e:entity>
  visited <e:entity id="08932568" norm="Paris"
     type="administrative district:national capital">Paris
  </e:entity>
</node>

<!-- using the entity type map changes the wrapper elements from
  -- e:entity to entity:vip and entity:gep -->
<node xmlns:entity="http://marklogic.com/entity">
  <entity:vip>Nixon</entity:vip>
  visited
  <entity:gpe>Paris</entity:gpe>
</node>

<!-- using the "full" option and the entity type map -->
<node xmlns:entity="http://marklogic.com/entity">
  <entity:vip id="11208172"
    norm="Nixon">Nixon</entity:vip>
  visited
    <entity:gpe id="08932568" norm="Paris">Paris</entity:gpe>
</node>
```

15.4.7 JavaScript Example: cts.entityHighlight

This example illustrates how you can use `cts.entityHighlight` to enrich content when you need more control than that provided by `entity.enrich`. The example uses an in-memory dictionary that defines the following:

- An entity of type “person:head of state” for various phrases that describe former United States President Richard Nixon.
- Several different entity types for the word “Paris”.

The example uses the dictionary add enrichment around the phrases “Nixon” and “Paris” in the following input node:

```
<node>Nixon visited Paris</node>
```

The example uses an in-memory dictionary and input data for the sake of self-containment. In a real application, you would usually store the dictionary in MarkLogic, as described in "Persisting or Retrieving an Entity Dictionary" on page 595. The input node can also be an XML document or other node in MarkLogic.

Copy and paste the following code into Query Console, set the Query Type to JavaScript, and run it. If you are unfamiliar with Query Console, see the *Query Console User Guide*.

```
'use strict';

const dictionary = cts.entityDictionary([
  cts.entity('11208172', 'Nixon', 'Nixon', 'person:head of state'),
  cts.entity('11208172', 'Nixon', 'Richard Nixon', 'person:head of state'),
  cts.entity('11208172', 'Nixon', 'Richard M. Nixon', 'person:head of state'),
  cts.entity('11208172', 'Nixon', 'Richard Milhous Nixon', 'person:head of
state'),
  cts.entity('11208172', 'Nixon', 'President Nixon', 'person:head of state'),
  cts.entity('08932568', 'Paris', 'Paris', 'administrative district:national
capital'),
  cts.entity('09145751', 'Paris', 'Paris', 'administrative district:town'),
  cts.entity('09145751', 'Paris', 'Paris', 'being:mythical being')
]);
const inputNode = new NodeBuilder()
                    .addElement('node', 'Richard Nixon visited Paris.')
                    .toNode();
const resultBuilder = new NodeBuilder();
cts.entityHighlight(inputNodeode,
  function(builder, entityType, text, normText, entityId, node, start)
  {
    if (text != '') {
      builder.addElement(fn.replace(entityType, ':| ', '-'), text);
    }
  },
  resultBuilder, dictionary);
resultBuilder.toNode();
```

The example produces the following output. Whitespace has been added to improve readability. The enrichment does not introduce new whitespace.

```
<node>
  <person-head-of-state>Nixon</person-head-of-state>
  visited
  <district-national-capital>Paris</district-national-capital>
</node>
```

The `builder` parameter of the callblack contains the `NodeBuilder` object you pass into `cts.entityHighlight`. The remaining parameters, such as `text` and `entityType` are populated with information from the matching dictionary entry. For details, see the function reference for `cts.entityHighlight`.

Each time `cts.entityHighlight` identifies a word or phrase that match a dictionary entry, it invokes the callback function passed in as the second parameter. The example function simiply generates an entity wrapper that uses the entity type name as the wrapper element QName, after replacing any occurrences of ":" or " " with a dash ("-").

```
builder.addElement(fn.replace(entityType, ':| ', '-'), text)
```

Note that you are responsible for extracting the final result from the NodeBuilder when the highlighting walk completes. For example, by calling `NodeBuilder.toNode()`.

If text matches more than one dictionary entry, your callback is invoked for each match, but the `text` parameter will be an empty string for all but the first match. The example as given tests for an empty `text` string and only generates replacement content for the first match.

```
function(builder, entityType, text, normText, entityId, node, start) {
  if (text != '') {
    builder.addElement(fn.replace(entityType, ':| ', '-'), text);
  }
}
```

For example, the term “Paris” actually matches 3 entries in the dictionary. If you remove the empty string test from the callback function, then the example produces the following output. The same wrapper is generated for the first match, but the subsequent matches insert an entity tag with no text content because `text` parameter is an empty string.

```
<administrative-district-national-capital>Paris</administrative-distri
ct-national-capital>
<administrative-district-town/>
<being-mythical-being/>
```

You can control the entity traversal through the value returned by the callback. The default action is “continue”. If you return “skip” or “break”, then you can interrupt the walk. For example, the following call exits the walk after the first match:

```
function(builder, entityType, text, normText, entityId, node, start) {
  if (text != '') {
    builder.addElement(fn.replace(entityType, ':| ', '-'), text);
    return 'break';
  }
}
```

15.5 Dictionary-Based Entity Extraction

You can use entity extraction to generate a list of entities from an XML document or other XML node. You define the set of possible entities in one or more entity dictionaries. You can use extracted entities for purposes such as creating searchable metadata or maintaining classification data outside of the original content. To mark up entity data inline, use entity enrichment; for details see “Dictionary-Based Entity Enrichment” on page 598.

This section covers the following topics related to using the dictionary-based APIs for entity extraction:

- API Summary
- Extraction Using entity:extract or entity.extract
- Extraction Using cts:entity-walk or cts.entityWalk
- XQuery Example: entity:extract
- XQuery Example: cts:entity-walk
- JavaScript Example: entity.extract
- JavaScript Example: cts.entityWalk

15.5.1 API Summary

The following table summarizes the dictionary-based APIs available for extracting entities from your XML documents. These APIs require you first to create one or more entity dictionaries, as described in "Creating an Entity Dictionary" on page 589.

Function	Description
`entity:extract` (XQuery) `entity.extract` (JavaScript)	Identify entities in a node and extract it as an XML element decorated with the entity type. Some customization of the generated XML is available.
`cts:entity-walk` (XQuery) `cts.entity-walk` (JavaScript)	Identify entities in a node and extract them in a custom format.

The `entity:extract` or `entity.extract` function generates entity elements of the same form as the replacement content generated by `entity:enrich` or `entity.enrich`. The output from `extract` should satisfy the needs of most applications. If you require more control, you can use `cts:entity-walk` or `cts.entityWalk` extraction instead.

15.5.2 Extraction Using entity:extract or entity.extract

When you call `entity:extract` (XQuery) or `entity.extract` (JavaScript) with just an input node and one or more entity dictionaries, then the extracted entities are wrapped in an `<entity/>` element with a `type` attribute that contains the entity type.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace entity="http://marklogic.com/entity"`
	`  at "/MarkLogic/entity.xqy";`
	
	`let $some-dictionary := ...`
	`entity:extract($some-node, $some-dictionary)`

Server-Side JavaScript	`const entity = require('/MarkLogic/entity');`
	
	`const someDictionary = ...;`
	`entity.extract(someNode, someDictionary);`

For example, the following element nodes were generated by `enrich` on content that contained text matching five entity dictionary entries for the terms “Richard Nixon”, “Nixon”, and “Paris”.

```
<e:entity type="person:head of state"
    xmlns:e="http://marklogic.com/entity">Richard Nixon</e:entity>
<e:entity type="person:head of state"
    xmlns:e="http://marklogic.com/entity">Nixon</e:entity>
<e:entity type="administrative district:national capital"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
<e:entity type="administrative district:town"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
<e:entity type="imaginary being:mythical being"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
```

For a complete example, see “XQuery Example: entity:extract” on page 614 or “JavaScript Example: entity.extract” on page 618.

If you pass multiple dictionaries to `extract`, then the dictionaries are applied in turn, in the order provided.

You can further tailor the output of `extract` as follows:

- Use the “full” option to decorate the entity element with additional information from the matched dictionary entry, such as the entity id and normalized text.
- Pass in a mapping between entity type names and element QNames to change the QName of the entity element based on the entity type.

For example, suppose you have an entity dictionary that defines the word “Nixon” as an entity of type “person:head of state”. Further, suppose you define a mapping from “person:head of state” to the QName “entity:vip”. Then, the following table illustrates different ways of formatting the extracted entities:

Use Case	Example
original text	`Nixon`
default extracted entity element	`<e:entity xmlns:e="http://marklogic.com/entity"> type="person:head of state"> Nixon </e:entity>`
“full” option to add additional entity attributes	`<e:entity id="11208172" norm="Nixon" start="1" path="/node/text()" type="person:head of state" xmlns:e="http://marklogic.com/entity"> Nixon </e:entity>`
entity type map to change wrapper QName	`<entity:vip xmlns:entity="http://marklogic.com/entity"> Nixon </entity:vip>`
map plus “full” option	`<entity:vip id="11208172" norm="Nixon" start="1" path="/node/text()" xmlns:entity="http://marklogic.com/entity"> Nixon </entity:vip>`

If this level of customization does not meet the needs of your application, see “Extraction Using cts:entity-walk or cts.entityWalk” on page 612.

15.5.3 Extraction Using cts:entity-walk or cts.entityWalk

When you use `cts:entity-walk` or `cts.entityWalk` , MarkLogic runs caller-specified code whenever text matches an entity dictionary entry. This means you have complete control over the result of the walk.

- Using cts:entity-walk in XQuery
- Using cts.entityWalk in JavaScript

15.5.3.1 Using cts:entity-walk in XQuery

When you use XQuery, you pass an inline entity generator expression to `cts:entity-walk` as an inline expression. The walk returns whatever items your generator produces.

MarkLogic makes information about the match available to your code through special variables such as `$cts:entity-type`, `$cts:text`, `$cts:entity-id`, `$cts:normalized-text`, and `$cts:start`. For details, see the function reference documentation for `cts:entity-walk`.

For example, the following code returns a sequence of JSON objects containing details about each match. The `$cts:*` variables are populated with details about the match by `cts:entity-walk`.

```
cts:entity-walk($input-node,
  (object-node {
     "type": $cts:entity-type,
     "text": $cts:text,
     "normText": $cts:normalized-text,
     "id": $cts:entity-id,
     "start": $cts:start
  }), $dictionary)
```

You can control the walk by using `xdmp:set` to set the variable `$cts:action` to “continue”, “skip”, or “break”. The default action is to continue.

For a complete example, see “XQuery Example: cts:entity-walk” on page 616.

15.5.3.2 Using cts.entityWalk in JavaScript

When you use Server-Side JavaScript, you pass an entity generator callback function to `cts.entityWalk`. MarkLogic invokes the callback whenever an entity match is found. The callback function has the following signature:

```
function(entityType, text, normText, entityId, node, start)
```

MarkLogic populates these parameters with details from the input node and matched entity dictionary entry.

You’re responsible for accumulating any data created by the extraction in a variable in scope at the point of call. For example, the following code creates a JavaScript object containing details about each match and accumulats the objects in a `results` variable.

```
const results = [];
cts.entityWalk(inputNode,
  function(entityType, text, normText, entityId, node, start) {
    results.push({
      type: entityType,
      text: text,
      norm: normText,
      id: entityId,
      start: start
    });
  },
  dictionary);
```

The value returned by your callback controls the walk. The default action is to continue the walk. You can return “skip” or “break” to halt the walk.

For a complete JavaScript example, see “JavaScript Example: cts.entityWalk” on page 620.

15.5.4 XQuery Example: entity:extract

This example uses `entity:extract` to extract entities from XML content, as described in “Extraction Using entity:extract or entity.extract” on page 611. The example demonstrates the use of various cutomization features of `entity:extract`.

The example uses an in-memory dictionary that defines the following:

- An entity of type “person:head of state” for various phrases that describe former United States President Richard Nixon.
- Several different entity types for the word “Paris”.

The example uses an in-memory dictionary and input data for the sake of self-containment. In a real application, you would usually store the dictionary in MarkLogic, as described in “Persisting or Retrieving an Entity Dictionary” on page 595.

The example uses the dictionary to extract entities for the phrases “Nixon” and “Paris” in the following input node:

```
<node>Nixon visited Paris</node>
```

Copy and paste the following code into Query Console, set the Query Type to XQuery, and run it. If you are unfamiliar with Query Console, see the *Query Console User Guide*.

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

let $dictionary := cts:entity-dictionary((
  cts:entity("11208172","Nixon","Nixon","person:head of state"),
  cts:entity("11208172","Nixon","Richard Nixon","person:head of state"),
  cts:entity("11208172","Nixon","Richard M. Nixon","person:head of state"),
  cts:entity("11208172","Nixon","Richard Milhous Nixon",
             "person:head of state"),
  cts:entity("11208172","Nixon","President Nixon","person:head of state"),
  cts:entity("08932568","Paris","Paris",
             "administrative district:national capital"),
  cts:entity("09145751","Paris","Paris","administrative district:town"),
  cts:entity("09500217","Paris","Paris","imaginary being:mythical being")
))
(: Entity type to element QName map :)
let $mapping := map:map()
  => map:with("", xs:QName("entity:entity"))
  => map:with("administrative district", xs:QName("entity:gpe"))
  => map:with("person",
```

```
        map:map() => map:with("", xs:QName("entity:location"))
                  => map:with("head of state", xs:QName("entity:vip")))
let $input-node := <node>Nixon visited Paris</node>
return (
  "------- default -------",
  entity:extract($input-node, $dictionary),
  "------- full option -------",
  entity:extract($input-node, $dictionary, ("full")),
  "------- mapping -------",
  entity:extract($input-node, $dictionary, (), $mapping),
  "------- full + mapping -------",
  entity:extract($input-node, $dictionary, ("full"), $mapping)
)
```

The example extracts four entities, in different formats: One match for “Nixon”, and three for “Paris”. The following entities are extracted by the various parameter and option combinations:

```
------- default -------
<e:entity type="person:head of state"
    xmlns:e="http://marklogic.com/entity">Nixon</e:entity>
<e:entity type="administrative district:national capital"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
<e:entity type="administrative district:town"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
<e:entity type="imaginary being:mythical being"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
------- full option -------
<e:entity id="11208172" norm="Nixon" start="1"
    path="/node/text()" type="person:head of state"
    xmlns:e="http://marklogic.com/entity">Nixon</e:entity>
<e:entity id="08932568" norm="Paris" start="15"
   path="/node/text()" type="administrative district:national capital"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
<e:entity id="09145751" norm="Paris" start="15"
    path="/node/text()" type="administrative district:town"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
<e:entity id="09500217" norm="Paris" start="15"
    path="/node/text()" type="imaginary being:mythical being"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
------- mapping -------
<entity:vip
    xmlns:entity="http://marklogic.com/entity">Nixon</entity:vip>
<entity:gpe
    xmlns:entity="http://marklogic.com/entity">Paris</entity:gpe>
<entity:gpe
    xmlns:entity="http://marklogic.com/entity">Paris</entity:gpe>
<entity:entity type="imaginary being:mythical being"
    xmlns:entity="http://marklogic.com/entity">Paris</entity:entity>
------- full + mapping -------
<entity:vip id="11208172" norm="Nixon" start="1" path="/node/text()"
    xmlns:entity="http://marklogic.com/entity">Nixon</entity:vip>
<entity:gpe id="08932568" norm="Paris" start="15" path="/node/text()"
    xmlns:entity="http://marklogic.com/entity">Paris</entity:gpe>
```

```
<entity:gpe id="09145751" norm="Paris" start="15" path="/node/text()"
    xmlns:entity="http://marklogic.com/entity">Paris</entity:gpe>
<entity:entity id="09500217" norm="Paris" start="15" path="/node/text()"
    type="imaginary being:mythical being"
    xmlns:entity="http://marklogic.com/entity">Paris</entity:entity>
```

If the "full" option and entity type map features of `entity:extract` do not provide enough control of the output to meet the needs of your application, use `cts:entity-walk` instead.

For more details on entity type maps, see "Using an Entity Type Map for Extraction or Enrichment" on page 622.

15.5.5 XQuery Example: cts:entity-walk

This example uses `cts:entity-walk` to extract entities as JSON object nodes, rather than as XML elements as you would get using `entity:extract`. Each object contains details about the match, such as the entity type, entity id, and codepoint offset in the input node.

For more details, see "Extraction Using cts:entity-walk or cts.entityWalk" on page 612.

The example uses an in-memory dictionary that defines the following:

- An entity of type "person:head of state" for various phrases that describe former United States President Richard Nixon.
- Several different entity types for the word "Paris".

The example uses an in-memory dictionary and input data for the sake of self-containment. In a real application, you would usually store the dictionary in MarkLogic, as described in "Persisting or Retrieving an Entity Dictionary" on page 595.

The example uses the dictionary to extract entities for the phrases "Nixon" and "Paris" in the following input node:

```
<node>Nixon visited Paris</node>
```

Copy and paste the following code into Query Console, set the Query Type to XQuery, and run it. If you are unfamiliar with Query Console, see the *Query Console User Guide*.

```
xquery version "1.0-ml";

let $dictionary := cts:entity-dictionary((
  cts:entity("11208172","Nixon","Nixon","person:head of state"),
  cts:entity("11208172","Nixon","Richard Nixon","person:head of state"),
  cts:entity("11208172","Nixon","Richard M. Nixon","person:head of state"),
  cts:entity("11208172","Nixon","Richard Milhous Nixon",
             "person:head of state"),
  cts:entity("11208172","Nixon","President Nixon","person:head of state"),
  cts:entity("08932568","Paris","Paris",
             "administrative district:national capital"),
  cts:entity("09145751","Paris","Paris","administrative district:town"),
```

```
  cts:entity("09500217","Paris","Paris","imaginary being:mythical being")
))
let $input-node := <node>Nixon visited Paris</node>
return cts:entity-walk($input-node,
  (object-node {
     "type": $cts:entity-type,
     "text": $cts:text,
     "normText": $cts:normalized-text,
     "id": $cts:entity-id,
     "start": $cts:start
  }), $dictionary)
```

You should get output similar to the following:

```
{ "type":"person:head of state",
  "text":"Nixon", "normText":"Nixon",
  "id":"11208172", "start":1}

{ "type":"administrative district:national capital",
  "text":"Paris", "normText":"Paris",
  "id":"08932568", "start":15}

{ "type":"administrative district:town",
  "text":"Paris", "normText":"Paris",
  "id":"09145751", "start":15}

{ "type":"imaginary being:mythical being",
  "text":"Paris", "normText":"Paris",
  "id":"09500217", "start":15}
```

The `$cts:*` variables used to populate the JSON property values are set by `cts:entity-walk`, based on the matched text and dictionary entry.

You can control the walk by setting `$cts:action`. The default action is “continue”. If you set the action to “skip” or “break” using `xdmp:set`, then you can interrupt the walk. For example, the following call exits the walk after the first match:

```
cts:entity-walk($input-node,
  (xdmp:set($cts:action, "break"),
   object-node {
     "type": $cts:entity-type,
     "text": $cts:text,
     "normText": $cts:normalized-text,
     "id": $cts:entity-id,
     "start": $cts:start
  }), $dictionary)
```

15.5.6 JavaScript Example: entity.extract

This example uses `entity.extract` to identify entities in your content and generate a sequence of entity elements that describe the matches, as described in "Extraction Using entity:extract or entity.extract" on page 611. The example demonstrates the use of various customization features of `entity.extract`.

The example uses an in-memory dictionary that defines the following:

- An entity of type "person:head of state" for various phrases that describe former United States President Richard Nixon.
- Several different entity types for the word "Paris".

The example uses an in-memory dictionary and input data for the sake of self-containment. In a real application, you would usually store the dictionary in MarkLogic, as described in "Persisting or Retrieving an Entity Dictionary" on page 595.

The example uses the dictionary to extract entities for the phrases "Nixon" and "Paris" in the following input node:

```
<node>Nixon visited Paris</node>
```

Copy and paste the following code into Query Console, set the Query Type to JavaScript, and run it. If you are unfamiliar with Query Console, see the *Query Console User Guide*.

```
'use strict';
const entity = require('/MarkLogic/entity');

// Construct the dictionary. Could also get it from the db.
const dictionary = cts.entityDictionary([
  cts.entity('11208172', 'Nixon', 'Nixon', 'person:head of state'),
  cts.entity('11208172', 'Nixon', 'Richard Nixon', 'person:head of state'),
  cts.entity('11208172', 'Nixon', 'Richard M. Nixon', 'person:head of state'),
  cts.entity('11208172', 'Nixon', 'Richard Milhous Nixon', 'person:head of
state'),
  cts.entity('11208172', 'Nixon', 'President Nixon', 'person:head of state'),
  cts.entity('08932568', 'Paris', 'Paris', 'administrative district:national
capital'),
  cts.entity('09145751', 'Paris', 'Paris', 'administrative district:town'),
  cts.entity('09500217', 'Paris', 'Paris', 'being:mythical being')
]);
// Entity type to wrapper element QName map
const mapping = {
  '' : fn.QName('http://marklogic.com/entity', 'entity:entity'),
  'administrative district':
       fn.QName('http://marklogic.com/entity', 'entity:gpe'),
  person: {
    '': fn.QName('http://marklogic.com/entity', 'entity:person'),
    'head of state': fn.QName('http://marklogic.com/entity', 'entity:vip')
  }
};
```

```
// Construct <node>Nixon visited Paris</node>
const inputNode = new NodeBuilder()
                  .addElement('node', 'Nixon visited Paris')
                  .toNode();
const resultBuilder = new NodeBuilder();
const result = [
  entity.extract(inputNode, dictionary),
  entity.extract(inputNode, dictionary, ['full']),
  entity.extract(inputNode, dictionary, null, mapping),
  entity.extract(inputNode, dictionary, ['full'], mapping)
];
result;
```

The example extracts four entities, in different formats: One match for “Nixon”, and three for “Paris”. The example extracts the following entities, based on the various parameter and option combinations:

```
------- default -------
<e:entity type="person:head of state"
    xmlns:e="http://marklogic.com/entity">Nixon</e:entity>
<e:entity type="administrative district:national capital"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
<e:entity type="administrative district:town"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
<e:entity type="imaginary being:mythical being"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
------- full option -------
<e:entity id="11208172" norm="Nixon" start="1"
    path="/node/text()" type="person:head of state"
    xmlns:e="http://marklogic.com/entity">Nixon</e:entity>
<e:entity id="08932568" norm="Paris" start="15"
   path="/node/text()" type="administrative district:national capital"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
<e:entity id="09145751" norm="Paris" start="15"
    path="/node/text()" type="administrative district:town"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
<e:entity id="09500217" norm="Paris" start="15"
    path="/node/text()" type="imaginary being:mythical being"
    xmlns:e="http://marklogic.com/entity">Paris</e:entity>
------- mapping -------
<entity:vip
    xmlns:entity="http://marklogic.com/entity">Nixon</entity:vip>
<entity:gpe
    xmlns:entity="http://marklogic.com/entity">Paris</entity:gpe>
<entity:gpe
    xmlns:entity="http://marklogic.com/entity">Paris</entity:gpe>
<entity:entity type="imaginary being:mythical being"
    xmlns:entity="http://marklogic.com/entity">Paris</entity:entity>
------- full + mapping -------
<entity:vip id="11208172" norm="Nixon" start="1" path="/node/text()"
    xmlns:entity="http://marklogic.com/entity">Nixon</entity:vip>
<entity:gpe id="08932568" norm="Paris" start="15" path="/node/text()"
    xmlns:entity="http://marklogic.com/entity">Paris</entity:gpe>
```

```
<entity:gpe id="09145751" norm="Paris" start="15" path="/node/text()"
    xmlns:entity="http://marklogic.com/entity">Paris</entity:gpe>
<entity:entity id="09500217" norm="Paris" start="15" path="/node/text()"
    type="imaginary being:mythical being"
    xmlns:entity="http://marklogic.com/entity">Paris</entity:entity>
```

If the "full" option and entity type map features of cts:extract do not provide enough control of the output to meet the needs of your application, use `cts.entityWalk` instead.

For more details on entity type maps, see "Using an Entity Type Map for Extraction or Enrichment" on page 622.

15.5.7 JavaScript Example: cts.entityWalk

This example uses `cts.entityWalk` to extract entities as JSON object nodes, rather than as the XML elements you get from `entity.extract`. Each object contains details about the match, such as the entity type, entity id, and codepoint offset in the input node.

For more details, see "Extraction Using cts:entity-walk or cts.entityWalk" on page 612.

The example uses an in-memory dictionary that defines the following:

- An entity of type "person:head of state" for various phrases that describe former United States President Richard Nixon.
- Several different entity types for the word "Paris".

The example uses an in-memory dictionary and input data for the sake of self-containment. In a real application, you would usually store the dictionary in MarkLogic, as described in "Persisting or Retrieving an Entity Dictionary" on page 595.

The example uses the dictionary to extract entities for the phrases "Nixon" and "Paris" in the following input node:

```
<node>Nixon visited Paris</node>
```

Copy and paste the following code into Query Console, set the Query Type to JavaScript, and run it. If you are unfamiliar with Query Console, see the *Query Console User Guide*.

```
'use strict';

// Construct the dictionary. Could also get it from the db.
const dictionary = cts.entityDictionary([
  cts.entity('11208172', 'Nixon', 'Nixon', 'person:head of state'),
  cts.entity('11208172', 'Nixon', 'Richard Nixon', 'person:head of state'),
  cts.entity('11208172', 'Nixon', 'Richard M. Nixon', 'person:head of state'),
  cts.entity('11208172', 'Nixon', 'Richard Milhous Nixon', 'person:head of
state'),
  cts.entity('11208172', 'Nixon', 'President Nixon', 'person:head of state'),
  cts.entity('08932568', 'Paris', 'Paris', 'administrative district:national
```

```
capital'),
  cts.entity('09145751', 'Paris', 'Paris', 'administrative district:town'),
  cts.entity('09500217', 'Paris', 'Paris', 'being:mythical being')
]);
// Construct <node>Nixon visited Paris</node>
const inputNode = new NodeBuilder()
                    .addElement('node', 'Richard Nixon visited Paris')
                    .toNode();
const resultBuilder = new NodeBuilder();
const results = [];
cts.entityWalk(inputNode,
  function(entityType, text, normText, entityId, node, start) {
    results.push({
      type: entityType,
      text: text,
      norm: normText,
      id: entityId,
      start: start
    });
  },
  dictionary);

results;
```

The example constructs a JavaScript object for each match. Each object contains details about the match, such as the entity type, entity id, and code-point offset in the input node. You should get output similar to the following:

```
[{"type":"person:head of state",
  "text":"Nixon", "normText":"Nixon",
  "id":"11208172", "start":1},

{ "type":"administrative district:national capital",
  "text":"Paris", "normText":"Paris",
  "id":"08932568", "start":15},

{ "type":"administrative district:town",
  "text":"Paris", "normText":"Paris",
  "id":"09145751", "start":15},

{ "type":"imaginary being:mythical being",
  "text":"Paris", "normText":"Paris",
  "id":"09500217", "start":15}]
```

The parameter values passed to your callback are populated by `cts.entityWalk` based on the matched text and dictionary entry.

You can control the walk by returning an action string value. The default action is “continue”. If you return “skip” or “break”, then you can interrupt the walk. For example, the following call exits the walk after the first match:

```
cts.entityWalk(inputNode,
  function(entityType, text, normText, entityId, node, start) {
    results.push({
      type: entityType,
      text: text,
      norm: normText,
      id: entityId,
      start: start
    });
    return 'break';
  },
  dictionary);
```

15.6 Using an Entity Type Map for Extraction or Enrichment

This section describes how to use the entity type map parameter accepted by the XQuery functions `entity:extract` and `entity:enrich`, or the JavaScript functions `entity.extract` and `entity.enrich`. Such a mapping gives you more control over the format of the extracted entities or enrichment markup.

This section covers the following topics:

- Entity Type Map Basics
- The Default Entity Type Map
- Handling Compound Entity Types
- Filtering Entity Types With a Mapping

15.6.1 Entity Type Map Basics

When you use the XQuery functions `entity:enrich` and `entity:extract`, or the JavaScript functions `entity.enrich` and `entity.extract`, you can pass in a mapping from entity type names to XML element QNames.

MarkLogic defines a default mapping that the `enrich` and `extract` functions use if you do not provide your own enitity type mapping. For details, see “The Default Entity Type Map” on page 624.

An entity type map enables you to change the QName of the generated entity wrapper element based on the entity type of a matching dictionary entry. For example, you can create a mapping that generates a `my:person` wrapper element when the entity type is “person”, instead of the default `e:entity` wrapper element.

In XQuery, use a `map:map` to define the entity type mappings. In JavaScript, use a JavaScript object.

The key value pairs in the mapping have the following characteristics:

- The key is an entity type name.
- A key that is an empty string specifies the QName to use when no explicit mapping exists for a type.
- The value is either a QName or another entity type map. When the value is a map, it defines a mapping for a segment of a compound entity type such as "place:building". For more details, see "Handling Compound Entity Types" on page 626.
- If you map a type to the default entity QName (`fn:QName("http://marklogic.com/entity", "entity")`), then the generated wrapper element includes a `type` attribute, just as it does when you do not use a map. If you map a type to any other QName, then the wrapper element has no `type` attribute because the type is implicit in the mapping.

If you use a type map, then any entity that is not covered by the map is discarded. That is, text of an umapped type is not treated as an entity, even if it matches an entity dictionary entry.

For example, the default entity wrapper generated by `enrich` and `extract` is of the following form:

```
<e:entity xmlns:e=... type="theEntityType">theText</e:entity>
```

Suppose you do not want all entities to generate an `<e:entity/>`. Instead, you want the following behavior:

If the entity type is	Then generate a wrapper with QName
`person`	`my:person`
`location`	`my:place`
`anything else`	`e:entity`

The followig map produces the desired behavior:

Language	Example

XQuery	`let $map := map:map()`
	`  => map:with("", xs:QName("entity:entity"))`
	`  => map:with("person", fn:QName("http://my/example","my:person"))`
	`  => map:with("location", fn:QName("http://my/example","my:place"))`

Server-Side JavaScript	`const map = {`
	`  '' : fn.QName('http://marklogic.com/entity', 'entity:entity'),`
	`  location: fn.QName('http://my/example', 'my:place'),`
	`  person: fn.QName('http://my/example', 'my:person')`
	`}`

The example map produce entities of the following forms.

```
<my:person xmlns:my=...>somePerson</my:person>
<my:place xmlns:my=...>somePlace</my:place>
<e:entity xmlns:e=... type="thing">someThing</e:entity>
```

Notice that only the `e:entity` element includes a type attribute. This is because the entity type is assumed to be implicit in the QName customization when you use a custom QName.

In JavaScript, you can also use associative array syntax to construct a map. For example:

```
map[''] = fn.QName('http://marklogic.com/entity', 'entity:entity');
map['location' = fn.QName('http://my/example', 'my:place');
map['person'] = person: fn.QName('http://my/example', 'my:person');
```

15.6.2 The Default Entity Type Map

If you do not pass your own entity type map into the `extract` and `enrich` library functions, MarkLogic uses its default map. The default map enables you to create a dictionary for some commonly used entity abstractions without adopting a complex external ontology or defining your own type system.

The wrapper elements generated using the default map are all in the namespace `http://marklogic.com/entity`. For example:

```
<e:entity xmlns:e="http://marklogic.com/entity">...</e:entity>
```

The default map defines mappings for common entity abstractions such as person, location, url, and currency. For example, the entity type name `PERSON` maps to the QName `e:person`, and the entity type `IDENTIFIER:URL` maps to `e:url`. Any unrecognized entity type maps to the QName `e:entity`.

For a complete list of the default key-value pairs, see the function reference for the XQuery functions `entity:enrich` or `entity:extract`, or the JavaScript functions `entity.enrich` and `entity.extract`.

The following example defines an entity dictionary that uses entity types from the default map (LOCATION, IDENTIFIER:MONEY, and NATIONALITY) and one type (thing) that is not used by the default map.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace entity="http://marklogic.com/entity"`
	`  at "/MarkLogic/entity.xqy";`
	
	`let $dictionary := cts:entity-dictionary((`
	`  cts:entity("1234", "Tokyo", "Tokyo", "LOCATION"),`
	`  cts:entity("2345", "Yen", "Yen", "IDENTIFIER:MONEY"),`
	`  cts:entity("4567", "Japanese", "Japanese", "NATIONALITY"),`
	`  cts:entity("5678", "trip", "trip", "thing")`
	`))`
	`let $input-node :=`
	`  <node>The cost of the trip to Tokyo was paid in Japanese Yen.</node>`
	`return entity:extract($input-node, $dictionary)`

Server-Side JavaScript	`'use strict';`
	
	`const entity = require('/MarkLogic/entity');`
	`const dictionary = cts.entityDictionary([`
	`  cts.entity('1234', 'Tokyo', 'Tokyo', 'LOCATION'),`
	`  cts.entity('2345', 'Yen', 'Yen', 'IDENTIFIER:MONEY'),`
	`  cts.entity('4567', 'Japanese', 'Japanese', 'NATIONALITY'),`
	`  cts.entity('5678', 'trip', 'trip', 'thing')`
	`]);`
	`const inputNode =`
	`  new NodeBuilder()`
	`  .addElement('node', 'The cost of the trip to Tokyo was paid in Japanese Yen.')`
	`  .toNode();`
	`entity.extract(inputNode, dictionary);`

The example extracts the following sequence of entities. Notice that the entity "trip", whose type does not have an entry in the default map, is extracted as an `e:entity` element.

```
<e:entity type="thing" xmlns:e="http://marklogic.com/entity">cost</e:entity>
<e:location xmlns:e="http://marklogic.com/entity">Washington, DC</e:location>
<e:nationality xmlns:e="http://marklogic.com/entity">Japanese</e:nationality>
<e:money xmlns:e="http://marklogic.com/entity">Yen</e:money>
```

15.6.3 Handling Compound Entity Types

A compound entity type is composed of colon (":") separated segments that specify sub-types of that type. For example, an entity type such as "person:head of state" has two segments: "person" and "head of state" and specifies a sub-type of person. You can create an entity type map that takes such specialization into consideration by creating a key-value pair where the value is a map.

For example, suppose you want to create a mapping that has the following effect:

If the entity type is	Then generate a wrapper with QName
`person`	`person`
`person:artist`	`artiste`
`person:head of state`	`vip`
`person:`*`anythingElse`*	`person`

Then you can use the following map get the desired behavior. Notice that the value for the "person" key is itself a type map.

Language	Example

XQuery	`map:map()`
	`  => map:with("person",`
	`  map:map() => map:with("", "person")`
	`  => map:with("artist", xs:QName("artiste")`
	`  => map:with("head of state", xs:QName("vip")`

Server-Side JavaScript	`{person: {`
	`  '': 'person',`
	`  artist: fn.QName('', 'artiste'),`
	`  'head of state': fn.QName('', 'vip')`
	`} }`

You can nest the type maps as deeply as necessary to cover additional type segments.

15.6.4 Filtering Entity Types With a Mapping

When you use type map, any entity type not covered by the map is discarded. That is, text that matches an unmapped type is not treated as an entity reference for purposes of enrichment or extraction. In this way, an entity type map can serve as a filter.

For example, if you have an entity dictionary that contains entries for the entity types “person”, “location” and “thing”, but you are only interested in extracting “person” entities, then you can define a map that only covers the “person” entity type, causing any “location” or “thing” entities to be treated as non-entity text, and thus not extracted. Note that you still incur the cost of entity matching.

The following example defines a map that covers only a single entity type, “person”. If used with an entity dictionary that also defines “location” and “thing” entity types, such entities would not be extracted when used with the map.

Language	Example
XQuery	`map:map()=> map:with("person", xs:QName("entity:entity"))`
Server-Side JavaScript	`{'person' : fn.QName('http://marklogic.com/entity', 'entity:entity') }`

15.7 Overlapping Entity Match Handling

This section discusses how the dictionary-based APIs behave when more than one entity definition applies to the same piece of text. See the following topics for more details:

- Understanding Entity Overlaps
- Overlap Handling Options
- Example: Overlap Handling in entity:extract and entity.extract
- Example: Overlap Handling in entity:enrich and entity.enrich
- Interaction with the Walk and Highlight Functions

15.7.1 Understanding Entity Overlaps

An entity overlap occurs when the same run of input text matches more than one entry in the same entity dictionary. For example, suppose you create a dictionary with entries for the terms “cat”, “black cat”, and “cat fur”. Then the phrase “A black cat fur ball” contains overlapping text runs matching all three of these entries:

The best treatment for such overlaps depends on your application. Allowing overlaps is often the best choice for entity extraction, but may not produce desirable results for entity enrichment.

When you allow overlaps during enrichment, the text "captured" for enrichment can be an empty string if one entity match is completely contained in another ("cat" in "black cat"), or a partial string if the matches partially overlap ("black cat" and "cat fur"). For more details, see "Example: Overlap Handling in entity:enrich and entity.enrich" on page 629.

MarkLogic supports both allowing and removing overlaps through options that are available during dictionary creation. For details, see "Overlap Handling Options" on page 628.

Overlaps are only a concern within a single dictionary. You can pass multiple dictionaries to the extract and enrich library functions, and those dictionaries can contain entries whose text overlaps, but the dictionary with the first match always "wins".

15.7.2 Overlap Handling Options

When you create an entity dictionary, you can use the following options to control the handling of overlaps. These options are mutually exclusive.

- `allow-overlaps`: During extraction, include all overlapping matches. During enrichment, enrich the non-overlapping portions of each match; do not enrich entities completely contained within another match.
- `remove-overlaps`: MarkLogic selects a single "best" match and discards the others. The "best" match is the longest match when the text is scanned from left to right. If more than one match qualifies, select the leftmost.

By default, dictionaries are created with "allow-overlaps".

For more details, see "Example: Overlap Handling in entity:extract and entity.extract" on page 628 and "Example: Overlap Handling in entity:enrich and entity.enrich" on page 629.

These options also affect how often your extraction or enrichment code is called and with which values when you use `cts:entity-highlight`, `cts:entity-walk`, `cts.entityHighlight`, or `cts.entityWalk`. For details, see "Interaction with the Walk and Highlight Functions" on page 631.

15.7.3 Example: Overlap Handling in entity:extract and entity.extract

This section explores how the overlap option set for a dictionary affects the output of `entity:extract` and `entity.extract`.

Suppose you have an entity dictionary containing the following entries:

ID	Norm. Text	Text	Entity Type
1234	cat	cat	feline
2345	black cat	black cat	superstition
3456	cat fur	cat fur	allergen

Suppose that your input data is the following XML element node:

```
<node>A black cat fur ball</node>
```

Then the following table illustrates default results from calling `entity:extract` or `entity.extract` with the example data and a dictionary using different overlap options. (Whitespace has been added to the sample output to improve readability.)

Option	Extraction Result

allow-overlaps	`<e:entity type="superstition" xmlns:e=...>`
	`  **black cat**`
	`</e:entity>`
	`<e:entity type="allergen" xmlns:e=...>`
	`  **cat fur**`
	`</e:entity>`
	`<e:entity type="feline" xmlns:e=...>`
	`  **cat**`
	`</e:entity>`

remove-overlaps	`<e:entity type="superstition" xmlns:e=...>`
	`  **black cat**`
	`</e:entity>`

Notice that when you use “remove-overlaps”, `extract` returns only the “black cat” entity match because this is longest match.

15.7.4 Example: Overlap Handling in entity:enrich and entity.enrich

This section explores how the overlap option set for a dictionary affects the output of `entity:enrich` and `entity.enrich`.

Suppose you have an entity dictionary containing the following entries:

ID	Norm. Text	Text	Entity Type
1234	cat	cat	feline
2345	black cat	black cat	superstition
3456	cat fur	cat fur	allergen

Suppose that your input data is the following XML element node:

```
<node>A black cat fur ball</node>
```

The following table illustrates default results from calling `entity:enrich` or `entity.enrich` with the example data and a dictionary using different overlap options. (Whitespace has been added to the sample output to improve readability.)

Option	Enrichment Result

allow-overlaps	`<node xmlns:e="http://marklogic.com/entity">A`
	`  **<e:entity type="superstition">black cat</e:entity>**`
	`  **<e:entity type="allergen"> fur</e:entity>** ball`
	`</node>`

remove-overlaps	`<node xmlns:e="http://marklogic.com/entity">`
	`  A **<e:entity type="superstition">black cat</e:entity>** fur ball`
	`</node>`

Notice the following about using a dictionary with “allow-overlaps”enabled during enrichment:

- The “cat” entity is not reflected in the output because the matched text (“cat”) is completely encapsulated in another match, “black cat”. The term “cat” cannot be marked up without adding new, duplicate text to the content, which the API never does.
- The “cat fur” entity markup only captures the text “ fur” because this is the non-overlapping portion of the matched text. Again, it is not possible to mark up the whole phrase “cat fur” without introducing duplicate text.

Thus, you usually want to use a dictionary with “remove-overlaps” enabled for enrichment.

15.7.5 Interaction with the Walk and Highlight Functions

The XQuery functions `cts:entity-walk` or `cts:entity-highlight` and the Server-Side JavaScript functions `cts.entityWalk` or `cts.entityHighlight` interact with overlaps as follows:

- When “remove-overlaps” is enabled on a dictionary, your code is only evaluated for the “best” match, as previously described.
- When “allow-overlaps” is enabled on a dictionary, your code is evaluated for every overlapping match in the dictionary.
- When “allow-overlaps” is enabled, then the `text` value made available to your code by `cts:entity-highlight` and `cts.entityHighlight` will be an empty string if it is completely contained in another match (as with “cat” and “black cat”)
- When “allow-overlaps” is enabled, then the `text` value made available to your code by `cts:entity-highlight` and `cts.entityHighlight` will be only the non-overlapping part of a partial overlap (as with “black cat” and “cat fur”).

For examples of cases where you might get an empty or partial string during entity highlighting, see “Example: Overlap Handling in entity:enrich and entity.enrich” on page 629. The `enrich` library function is basically an abstraction on top of `cts:entity-highlight` and `cts.entityHighlight`.

15.8 Entity Identification Using Reverse Query

If your entities cannot be identified by string matching, but can be described by a query, you can use a reverse query for entity identification. A “normal” query says “find all documents that match this query”. A reverse query says “find all queries that would match this document”.

Use the following procedure:

1. Create a rule document containing a serialized cts query that describe the entity.

2. Use `cts:reverse-query` (XQuery) or `cts.reverseQuery` (JavaScript) to find the rule documents that contain revere queries satisfied by your content.

3. If you want to enrich the content, apply `cts:highlight` (XQuery) or `cts.highlight` (JavaScript) to the input and matching rules.

4. If you want to extract entities, apply `cts:walk` (XQuery) or `cts.walk` (JavaScript) to the input and matching rules.

For example, suppose you want to annotate terms in your content that correspond to activities such as hiking, biking, and running. However, you want to use a stemmed word query instead of a simple string match so that terms such as “run”, “ran”, and “running” match the “run” activity. You cannot use `entity:enrich` or `entity.enrich` because dictionary matching does use stemmed search.

The following node can serve as an entity matching rule for terms that stem to “run”, “swim”, “hike”, and “bike”.

```
<activity type="outdoor">
  <query>{cts:word-query(("run", "swim", "hike", "bike"))}</query>
</activity>
```

If you insert such rules into MarkLogic in a collection with the URI “activity”, then the following query finds words that match the rules, and wraps each matched word in an wrapper element whose local name is the same as the `type` attribute on the matching rule:

Language	Example

XQuery	`xquery version "1.0-ml";`
	
	`let $input-node := <node>I ran 5 miles and then went hiking</node>`
	`return`
	`fn:fold-left( function($prev, $next) {`
	`  cts:highlight($prev, cts:query($next/activity/query/*),`
	`    element {$next/activity/@type} {`
	`      $cts:text`
	`    }`
	`  )},`
	`  $input-node,`
	`  cts:search(fn:collection("activity"), cts:reverse-query($input-node))`
	`)`

Server-Side JavaScript	`'use strict';`
	`const entity = require('/MarkLogic/entity');`
	
	`const inputNode =`
	`  new NodeBuilder()`
	`    .addElement('node', 'I ran 5 miles and then went hiking')`
	`    .toNode();`
	`const matchingRules = cts.search(`
	`  cts.andQuery([`
	`    cts.collectionQuery('activity'),`
	`    cts.reverseQuery(inputNode)]));`
	`const resultBuilder = new NodeBuilder();`
	
	`for (let rule of matchingRules) {`
	`  cts.highlight(inputNode,`
	`    cts.query(fn.head(rule.xpath('/activity/query/*'))),`
	`    function(builder, text, node, queries, start) {`
	`      builder.addElement(`
	`        fn.head(rule.xpath('/activity/@type/data()')), text);`
	`    },`
	`    resultBuilder);`
	`}`
	`resultBuilder.toNode();`

If the “activity” collection includes the rule for “run”, “swim”, “hike”, and “bike” shown above, then the example produces the following output:

```
<node>
  I <outdoor>ran</outdoor> 5 miles and then went <outdoor>hiking</outdoor>
</node>
```

If you use `cts:walk` or `cts.walk` instead of `cts:highlight` or `cts.highlight`, then you can extract entities, rather than enrich the content. For example:

Language	Example

XQuery	`xquery version "1.0-ml";`
	
	`let $input-node := <node>I ran 5 miles and then went hiking</node>`
	`return`
	`fn:fold-left( function($prev, $next) {`
	`  cts:walk($prev, cts:query($next/activity/query/*),`
	`    element {$next/activity/@type} {`
	`      $cts:text`
	`    }`
	`  )},`
	`  $input-node,`
	`  cts:search(fn:collection("activity"), cts:reverse-query($input-node))`
	`)`

Server-Side JavaScript	`'use strict';`
	`const entity = require('/MarkLogic/entity');`
	
	`const inputNode =`
	`  new NodeBuilder()`
	`    .addElement('node', 'I ran 5 miles and then went hiking')`
	`    .toNode();`
	`const matchingRules = cts.search(`
	`  cts.andQuery([`
	`    cts.collectionQuery('activity'),`
	`    cts.reverseQuery(inputNode)]));`
	`const results = [];`
	
	`for (let rule of matchingRules) {`
	`  cts.walk(inputNode,`
	`    cts.query(fn.head(rule.xpath('/activity/query/*'))),`
	`    function(text, node, queries, start) {`
	`      const localname = fn.head(rule.xpath('/activity/@type/data()'));`
	`      results.push(`
	`        new NodeBuilder().addElement(localname, text).toNode());`
	`    });`
	`}`
	`results;`

The example produces the following extracted entities:

```
<outdoor>ran</outdoor>
<outdoor>hiking</outdoor>
```

15.9 Entity Enrichment Pipelines

If your entities cannot be identified using a dictionary (string matching) or a query, you can use a 3rd party entity extraction or enrichment library.

MarkLogic Server includes Content Processing Framework (CPF) applications to perform entity enrichment on your XML. 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:

- 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.

15.9.1 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:

- Expert System Cogito® and TEMIS Luxid®
- Calais OpenCalais
- SRA NetOwl
- 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.

15.9.2 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.

16.0 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

Note: This chapter describes how to create alerting applications using XQuery and XML. You can also create alerting applications using JavaScript and JavaScript objects. It is a best practice to pass XML when the alert action is implemented by an XQuery module and a JavaScript object when the action is implemented by a JavaScript file.

16.1 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 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 639 and “Alerting Sample Application” on page 646.

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 636.

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.

16.2 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 248.

16.3 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 Serializations of cts:query Constructors section of the chapter “Composing cts:query Expressions” on page 248.

16.4 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

16.4.1 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 *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.

16.4.2 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

16.4.2.1 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.

16.4.2.2 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.

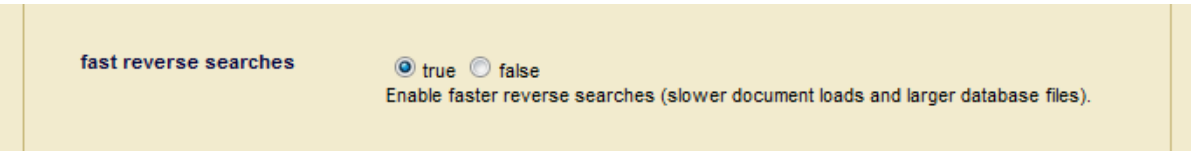
The `alert-user` role only has privileges that are used by the Alerting API; it does not provide execute privileges to any functions outside the scope of the Alerting API. The Alerting API uses the `alert-user` role as a mechanism to amp more privileged operations in a controlled way. It is therefore reasonably safe to assign this role to any user whom you trust to use your alerting application.

16.4.2.3 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.

16.5 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.

16.6 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*.

16.6.1 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

16.6.1.1 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.

16.6.1.2 Actions to Execute When an Alert Fires

An *action* is some XQuery or JavaScript 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.

16.6.1.3 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.

16.6.2 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

16.6.2.1 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 639. 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)
```

16.6.2.2 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. Alert actions can perform any action you can write in XQuery. For details about alert actions, see "Actions to Execute When an Alert Fires" on page 639.

In practice, setting up an alerting action requires 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 error log.

```
xdmp:log(fn:concat(xdmp:get-current-user(), " was alerted"))
```

You must install your action implementation in the modules database associated with your App Server. Once the implementation is installed, you can register it using the XQuery functions `alert:make-action` and `alert:action-insert` or the ServerSide JavaScript functions `alert.makeAction` and `alert.actionInsert`.

The following procedure outlines the steps for creating, installing, and registering an alerting action:

1. Implement your action in an XQuery library module. For example, the following is a simple action that logs a message to the error log:

```
xquery version "1.0-ml";
xdmp:log(fn:concat(xdmp:get-current-user(), " was alerted"))
```

2. Install your action module in the modules database associated with your App Server. For example, if your logging action is stored in a filesystem file with the path `/my/action/log.xqy`, then the following code installs it in the modules database with the URI `/alerts/log.xqy` when you run it against your App Server.

```
xquery version "1.0-ml";
xdmp:eval(
  'xdmp:document-load("/my/action/log.xqy",
      map:map() => map:with("uri", "/alerts/log.xqy")
                => map:with("format", "text"))',
  (), map:map() => map:with("database", xdmp:modules-database())
)
```

3. Associate your module with an alerting action using the XQuery function `alert:action-insert` or the Server-Side JavaScript function `alert.actionInsert`. This step must be performed as a user with the `alert-admin` role. For example:

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

let $action := alert:make-action(
    "logalert",
    "log to ErrorLog.txt",
    xdmp:modules-database(),
    xdmp:modules-root(),
    "/alerts/log.xqy",
    <alert:options>put anything here</alert:options> )
return
alert:action-insert("my-alert-config-uri", $action)
```

You can also create and insert an action using the REST Management API. For details, see `POST:/manage/v2/databases/{id|name}/alert/actions`.

For a more complex example of an alert logging action, see *MARKLOGIC_INSTALL_DIR*`/Modules/MarkLogic/alert/log.xqy` in your MarkLogic installation.

16.6.2.3 Create Rules (Users With alert-user Role)

To create a rule, use the XQuery functions `alert:make-rule` and `alert:rule-insert`, or the Server-Side JavaScript functions `alert.makeRule` and `alert.ruleInsert`.

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 example inserts a rule named “simple” that will fire the action named “logalert” whenever the specified word query matches. (See “Set Up Actions (User With alert-admin Role)” on page 641 for the implementation of the “logalert” action.) You must run this code as a user with the `alert-user` role or equivalent privileges. Note that equivalent production code will usually be much more complex, as this example has no user interface.

```
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"),
    "logalert",
    <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.

16.6.2.4 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.

16.6.3 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 are pre-built pipelines for alerting. The pipelines are 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. The Alerting pipleine is suitable for most alerting applications. The Alerting (spawn) pipeline spawns the actions in separate tasks, and therefore will result in increased parallelism which is beneficial if you have many actions that result from a single document change. For example, if you have an application that allows users to specify a query to alert on, and if many people specify the same query (for example, the name of a popular singer), then with the Alerting pipeline, each of those actions is run serially, and the actions will recover even if there is a failure in the middle of them; with the Alerting (spawn) pipeline, each action is spawned as a separate request, allowing more parallelism, but if there is a failure during the actions, the actions will not restart. Furthermore, if your alerting action updates the document being alerted on, then you must use the Alerting (spawn) pipeline, as the Alerting pipeline would result in a deadlock.

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.

> **Note:** When you use mlcp to import documents without extensions, CPF alerts do not work.

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 638.

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 641 and "Set Up Actions (User With alert-admin Role)" on page 641.

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 642.

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 the `alert:config-set-cpf-domain-names` function to notify the alerting configuration of the domain so the alerting action can determine which alerting configuration to use.

   For example, if your CPF domain 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.

16.7 Alerting Sample Application

A sample alerting application is available on http://developer.marklogic.com/code/alerting. 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.

17.0 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

17.1 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.

17.2 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.

17.3 Using cts:remainder to Estimate the Size of a Search

When you need to retrieve both search results and an estimate of the number of matching fragments as part of the same query statement, use the `cts:remainder` function. Running `cts:remainder` on a node or nodes returned by a search is more efficient that running `xdmp:estimate` on the sequence of nodes returned by `cts:search`. If you just need the estimate, but not the 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*.

17.4 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

17.4.1 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`.

17.4.2 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.

17.4.3 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).

18.0 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 to Emphasize a Query Match
- Using cts:contains to Test for a Stemmed Match
- Interaction With Wildcard Searches
- Using a User-Defined Stemmer Plugin

18.1 The Role of Stemming and Tokenization in Search

Tokenization splits a run of text into individual tokens, such as words, whitespace, and punctuation. The rules used to split text into tokens is language-specific. For example, in a language like English, word tokens are usually separated by whitespace and punctuation tokens. Thus, a string such as “ran, slept” tokenizes to the following in English:

- “ran” (token)
- “ “ (whitespace)
- “,” (punctuation)
- “ “ (whitespace)
- “slept” (word)

Tokenization is applied to documents when they are indexed, and to query text when you perform a search.

Stemming maps a word to its common lemma (stem). Thus, in the example above, “ran” stems to the verb “run” and “slept” stems to the verb “sleep”. Like tokenization, stemming rules are language-specific.

An unstemmed search matches only the word form you’re searching for. For example, searching for “ran” will not match a document containing “runs”. When stemmed search is enabled, the search matches the exact term, plus words with the same stem. Thus, a search for “ran” will also match documents containing “runs” or “running” because they all share the stem “run” in English.

Contains SAP BusinessObjects Text Analysis XI from SAP AG. Copyright © 1996-2011. All rights reserved.

18.2 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 761. You can also create a user-defined stemmer to add support for other languages; for details, see "Using a User-Defined Stemmer Plugin" on page 656.

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 683.

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. Each word evaluated in the context of a specific language. A term in one language will not match a stemmed search for the same term in another language. The language can be specified with an `xml:lang` attribute or by several other methods. For details on how languages affect queries, see "Querying Documents By Languages" on page 758.

18.3 Enabling Stemming

To use stemming in your searches, stemming must be enabled in your database configuration. All new databases created in MarkLogic Server have stemmed searches disabled by default. You can enable stemmed searches after initial creation of your database.

Stemmed searches are supported by special indexes. If you enable stemmed searches in an existing database, you must either reload or reindex the database to ensure that you get stemmed results from searches. You should plan on allocating additional disk space of about twice the size of the source content if you enable stemmed searches.

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. This is the default.
Basic	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.

When stemmed searches are enabled for a database, you can enable and disable the use of stemming on a per query basis through options. Query constructors such as `cts:word-query`, `cts:element-word-query`, and `cts:element-value-query` support "stemmed" and "unstemmed" options. For more details on these functions, see the *MarkLogic XQuery and XSLT Function Reference*.

Query terms that contain a wildcard will not be stemmed. If you leave the stemming option unspecified, the database configuration determines whether or not stemming is applied to words that do not contain a wildcard.

If stemming is turned off in the database, and stemming is explicitly specified in the query, the query will throw an error.

18.4 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 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 759.

18.5 Using cts:highlight to Emphasize a Query Match

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 468.

18.6 Using cts:contains to Test for a Stemmed Match

You can use `cts:contains` to test if a word matches a 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")
```

18.7 Interaction With Wildcard Searches

For information about how stemmed searches and Wildcard searches interact, see "Interaction with Other Search Features" on page 687.

18.8 Using a User-Defined Stemmer Plugin

You can use a user-defined stemmer plugin to affect how MarkLogic matches a word to its stems during search term resolution. You create a user-defined stemmer in C++ by implementing a subclass of the `marklogic::StemmerUDF` base class and deploying it to MarkLogic as a native plugin. The `StemmerUDF` class is a UDF (User Defined Function) interface.

MarkLogic also provides several built-in stemmer plugins that you can use to customize stemming instead of implementing your own. For details, see "Customization Using a Built-In Lexer or Stemmer" on page 766.

This section covers the following topics:

- When to Consider a User-Defined Stemmer
- StemmerUDF Interface Summary
- Understanding User-Defined Stemmer Control Flow
- Implementation Guidelines for User-Defined Stemmers
- Creating and Deploying a User-Defined Stemmer Plugin
- Registering a User-Defined Stemmer with MarkLogic
- Testing a User-Defined Stemmer
- Error Handling and Logging

18.8.1 When to Consider a User-Defined Stemmer

MarkLogic provides several built-in stemmers that you can configure for a language if you are not satisfied with the default stemmer. The following are some use cases in which you might consider implementing a your own stemmer:

- You need to stem a language that is not directly supported by MarkLogic.
- You want to use a specific 3rd party library for stemming for a given language.
- You need to use advanced stemming to obtain variants such as normalization and spelling variants not otherwise available.
- You require special format stemming in the context of specific data fields where the requirements are more complicated than simple reclassification.

In some cases, you might also need a custom lexer or custom dictionary. For example, if you're working with a language not supported by MarkLogic, you probably also need a custom lexer. For details, see "Custom Tokenization" on page 785 and "Custom Dictionaries for Tokenizing and Stemming" on page 665.

18.8.2 StemmerUDF Interface Summary

You implement a user-defined stemmer as a subclass of the `MarkLogic::StemmerUDF` base class. `StemmerUDF` is defined in *`MARKLOGIC_INSTALL_DIR`*`/include/MarkLogic.h`. You can find detailed documentation about the class in the User-Defined Function API reference and in `MarkLogic.h`. You can find an example implementation in *`MARKLOGIC_INSTALL_DIR`*`/Samples/NativePlugins`.

The following table contains a brief summary of the key methods of `StemmerUDF`. For a discussions of how the methods are used by MarkLogic, see "Understanding User-Defined Stemmer Control Flow" on page 658.

LexerUDF Method	Description
`initialize`	Initialize a `StemmerUDF` object after construction. This method is only called once per stemmer object.
`reset`	Prepare the stemmer to iterate over stems for a word. The first stem should be available through `StemmerUDF::stem` after calling this method. The preferred stem (if one exists) should be the first stem available.
`start`	Set the stemmer to the start of the list of stems. It should be possible to iterate over the stems repeatedly by successively calling `start`.
`next`	Advance the stemmer to the next stem. Returns false if there are no more stems.
`stem`	Return the current stem. Returns null if there is no current stem.
`delegate`	Returns true if stemming delegates to the default stemmer, instead of or in addition to this custom stemmer. For details, see "Understanding Stemming Delegation" on page 775.
`close`	Release the stemmer resources. This method is called when the stemmer is no longer needed.

18.8.3 Understanding User-Defined Stemmer Control Flow

When stemmed searches are enabled, MarkLogic can use either the default stemming plugin for a language, a user-defined stemming plugin, or both (via delegation). This section describes how MarkLogic interacts with a user-defined stemmer. See the following topics:

- When MarkLogic Uses a User-Defined Stemmer
- StemmerUDF Object Creation and Management
- Interaction During Stemming

18.8.3.1 When MarkLogic Uses a User-Defined Stemmer

When stemmed searches are enabled, stemming is performed when indexing documents and when evaluating queries. For more details, see "Tokenization and Stemming" on page 752.

When a word is eligible for stemming:

- MarkLogic first checks for matching entries in any custom dictionary for the language. If an entry is found, the stems from the dictionary are used.
- If no custom dictionary entry is found, then MarkLogic consults any configured custom stemmer, which could be a built-in stemmer plugin or a user-defined stemmer plugin, as described in "Stemming Customization" on page 763.
- MarkLogic might also consult the default stemming plugin for the language, depending on the delegation configuration. For details, see "Understanding Stemming Delegation" on page 775.

Thus, a user-defined stemming plugin will only be invoked when all the following conditions are met:

- Stemmed search is enabled on the database. For details, see Understanding the Text Index Settings in the *Administrator's Guide*.
- Stemming is needed in a language configured to use a user-defined stemming plugin. For details, see "Configuring Tokenization and Stemming Plugins" on page 764.
- No custom stemming dictionary is configured for the current language, or the configured dictionary contains no entry for the word under consideration.

For more information on custom dictionaries, see "Custom Dictionaries for Tokenizing and Stemming" on page 665.

18.8.3.2 StemmerUDF Object Creation and Management

`StemmerUDF` objects are created on demand and kept in a pool for re-use.

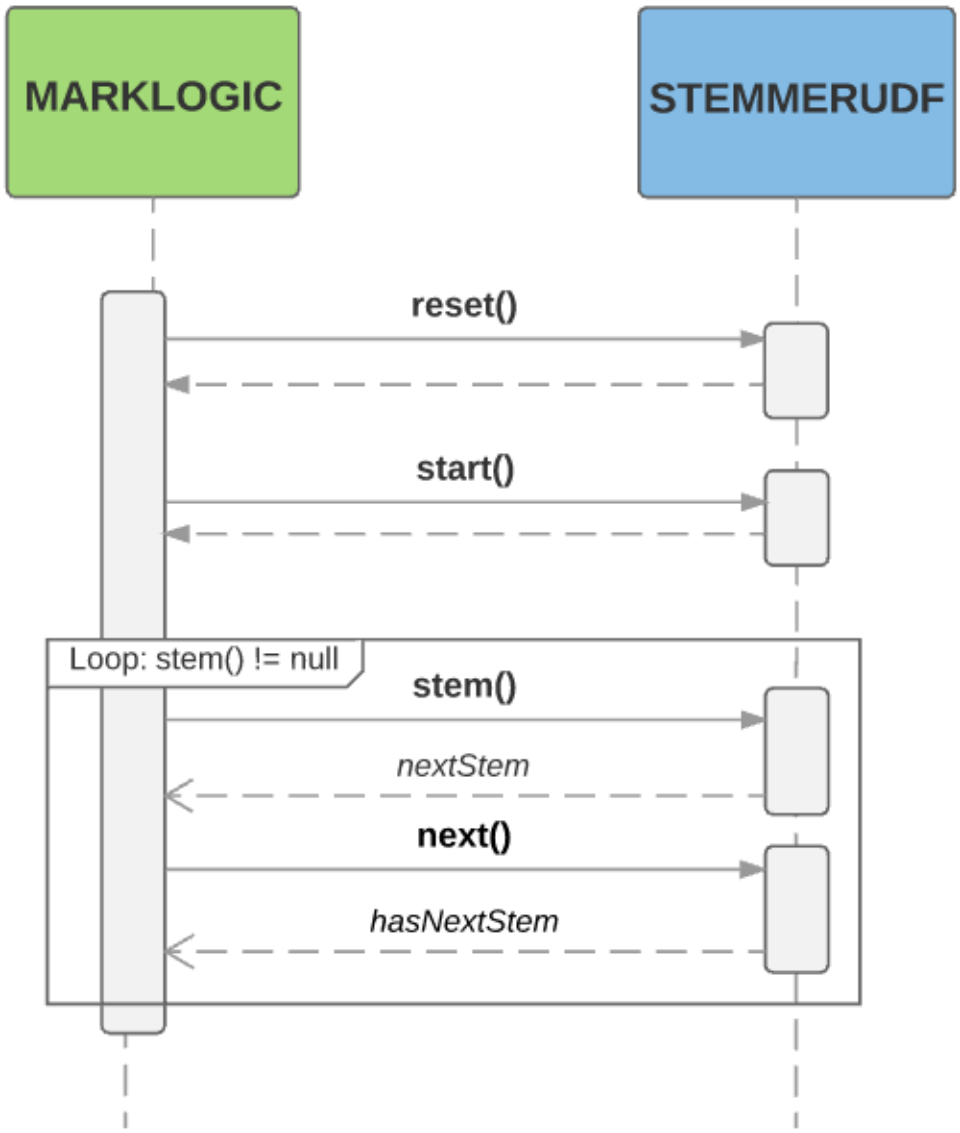
Stemmers tend to be heavy-weight objects, so MarkLogic maintains a (per-language) pool of stemmer objects for re-use. When MarkLogic needs one of your `StemmerUDF` objects, it first checks to see if one is available from the pool. If not, MarkLogic creates one using the object factory obtained during plugin registration. MarkLogic then calls the object's `initialize` method.

When a stemming task completes, the stemmer is returned to the pool unless it is marked as stale. MarkLogic can choose to mark a stemmer stale, or a stemmer can flag itself as stale by returning `true` from its `StemmerUDF::isStale` method.

When a stemmer is no longer needed, MarkLogic calls its `StemmerUDF::close` method. This enables the stemmer to deallocate memory and release other resources, as needed.

18.8.3.3 Interaction During Stemming

The following diagram is a high level illustration of how MarkLogic interacts with a `StemmerUDF` object while finding stems. The actual stemming process is more complex and has parts not represented here.



It is common to iterate over the stems for a word more than once. The `StemmerUDF::start` method is used to reset the iteration back to the first stem.

You can choose to have the default stemmer contribute stems instead of or in addition to your stemmer by returning true from the `delegate` method. For details, see "Understanding Stemming Delegation" on page 775.

18.8.4 Implementation Guidelines for User-Defined Stemmers

When implementing a `StemmerUDF` subclass, keep the following guidelines in mind.

> **Note:** Your implementation runs in the same memory and process space as MarkLogic Server, so errors in your implementation can crash MarkLogic Server. Before deploying a custom lexer, you should read and understand Using Native Plugins in the *Application Developer's Guide*. See also "Testing a User-Defined Stemmer" on page 663.

- You must implement a subclass of `marklogic::StemmerUDF` for each stemming algorithm you want to use.
- Stemming is a low-level, inner-loop operation that MarkLogic performs during indexing (including document ingestion) and query evaluation. Your stemmer should introduce as little overhead as possible.
- If the input word has a preferred stem, it should be the first stem returned after calling `StemmerUDF::reset` or `StemmerUDF::start`. The preferred stem is the only stem used when stemmed search is configured at the "basic" level.
- Indicate contractions with the "=" character and compound words with the "#" character. For example, "de=le" for the French word "du", or "Kind#Platz" for the German word "Kinderplatz".
- The tokenizer can optionally pass along a Part of Speech (POS) for the word being stemmed. The POS is supplied to `StemmerUDF::reset`. Stemming only makes use of this data for Japanese by default, but you can choose to use it in your stemmer.
- The stemmer owns the memory allocated for the stem returned by the `stem` method, and is responsible for releasing it when appropriate. Your stemmer can be called many times per word, so you should choose an efficient allocation strategy.
- Your implementation does not have to be thread safe. MarkLogic will instantiate a new stemmer object in each thread in which it wants to perform stemming.
- Report errors using the `Reporter` object that is passed to most `StemmerUDF` methods, rather than by throwing exceptions. For details, see "Error Handling and Logging" on page 663.
- You might also want to support your language with a custom lexer and/or a custom dictionary. To learn more about customization options, see "Stemming and Tokenization Customization" on page 762.

18.8.5 Creating and Deploying a User-Defined Stemmer Plugin

Follow the steps below to create and deploy a stemmer UDF in MarkLogic as a native plugin. A complete example is available in *`MARKLOGIC_DIR`*`/Samples/NativePlugins`.

1. Implement a subclass of the C++ class `marklogic::StemmerUDF`. See *`MARKLOGIC_DIR`*`/include/MarkLogic.h` for interface details.

2. Implement an extern "C" function called `marklogicPlugin` to perform plugin registration. For details, see Registering a Native Plugin at Runtime in the *Application Developer's Guide*.

3. Build a dynamically linked library containing your UDF and registration function. You should use the Makefile in *`MARKLOGIC_DIR`*`/Samples/NativePlugins` as the basis for building your plugin. For more details, see Building a Native Plugin Library in the *Application Developer's Guide*.

4. Following the directions in Using Native Plugins to package and install your plugin. See the note below about dependent libraries.

5. Configure your stemmer as the stemmer plugin for at least one language. For details, see "Configuring Tokenization and Stemming Plugins" on page 764.

The native plugin interface includes support for bundling dependent libraries in the native plugin zip file. However, many 3rd party natural language processing tools are large, complex, and have strict installation directory requirements. If you are using such a packge, you should install the 3rd party package package independently on each host in the cluster, rather than trying to include it inside your native plugin package.

18.8.6 Registering a User-Defined Stemmer with MarkLogic

A native plugin becomes available for use once you install it, but it will not be loaded until there is a reason to use it. For example, a plugin containing only a stemmer UDF is only loaded if it is associated with at least one language and MarkLogic needs to stem a word in that language.

When MarkLogic loads a native plugin, it performs a registration handshake to obtain details about the plugin such as what capabilities the plugin provides. This handshake is performed through an `extern "C"` function named `marklogicPlugin` that must be part of every native plugin.

The following code is an example of a registration function for a plugin that registers only a single stemmer capability. Assume the plugin implements a `StemmerUDF` subclass named `MyStemmerUDF`. The stemmer is registered with the plugin id "sample_stemmer".

```
extern "C" PLUGIN_DLL void
marklogicPlugin(Registry& r)
{
  r.version();
  r.registerStemmer<MyStemmerUDF>("sample_stemmer");
}
```

The plugin id returned by the `registerStemmer` method, along with the relative path under which the plugin is installed, is used elsewhere to identify your user-defined stemming plugin.

For details, see Registering a Native Plugin at Runtime in the *Application Developer's Guide*. For a complete example, see the code in *`MARLOGIC_DIR`*`/Samples/NativePlugins`.

18.8.7 Testing a User-Defined Stemmer

You can test your stemmer implementation in the following ways:

- Create standalone test scaffolding.
- Use the `cts:stem` XQuery function or the `cts.stem` Server-Side JavaScript function to exercise your plugin after it is installed and configured for at least one language.

Testing your stemmer standalone during development is highly recommended. It is much easier to debug your code in this setup. Also, since it is possible for native plugin code to crash MarkLogic, it is best to test and stabilize your code outside the server environment.

You can find example test scaffolding in *MARKLOGIC_DIR*`/Samples/NativePlugins/TestStemTok.cpp`. See the `main()` function for a starting point.

18.8.8 Error Handling and Logging

Use `marklogic::Reporter` to log messages and notify MarkLogic Server of fatal errors. Your code should not report errors to MarkLogic Server by throwing exceptions.

Report non-fatal errors and other messages using `marklogic::Reporter::log`. This method logs a message to the MarkLogic Server error log and returns control to your code. Most methods of `LexerUDF` accept a `marklogic::Reporter` input parameter.

Report **fatal** errors using `marklogic::Reporter::error`. You should reserve calls to `Reporter::error` for serious errors from which no recovery is possible. Reporting an error via `Reporter::error` has the following effects:

- If you report a fatal stemming error during document insertion, the insertion transaction aborts.
- If you report a fatal stemming error during reindexing, reindexing of the document fails.
- Control does not return to your code. Stemming stops.
- MarkLogic Server returns `XDMP-UDFERR` to the application. Your error message is included in the `XDMP-UDFERR` error.

The following snippet reports an error and aborts tokenization:

```
#include "MarkLogic.h"
using namespace marklogic;
...
void ExampleUDF::next(Reporter& r)
{
  ...
  r.log(Reporter::Error, "Bad codepoint.");
}
```

For more details, see the `marklogic::Reporter` class in `MARKLOGIC_DIR/include/MarkLogic.h`.

19.0 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
- Custom Dictionary Format
- Custom Dictionary Function Summary
- Example: Managing a Custom Dictionary in XQuery
- Example: Managing a Custom Dictionary in JavaScript
- Example: Exercising a Custom Dictionary

19.1 Custom Dictionaries in MarkLogic Server

One way you can customize stemming and/or tokenization in MarkLogic is by defining a custom stemming or tokenization dictionary for a language. A given language can have at most one custom stemming dictionary and one custom tokenization dictionary. Some languages, such as Japanese and Chinese, use a single dictionary for both stemming and tokenization.

Stemming is the process of reducing a word to one or more stems. A *stemming dictionary* maps a word to its lemma (stem). A stemmer can use a stemming dictionary to improve the precision of a search. For example, the default stemming dictionary for English enables MarkLogic to map the words “views”, “viewed”, and “viewing” back to their common stem, “view”. To learn more about stemming, see “Understanding and Using Stemmed Searches” on page 652.

Tokenization is the process of partitioning text into a sequence of word, whitespace, and punctuation tokens. A *tokenization dictionary* identifies runs of text that should be considered words. A tokenizer can use this data to model text and split it into tokens of the appropriate types.

The following list contains some use cases for creating a custom dictionary:

- For languages that do not tokenize based on whitespace, such as Japanese (`ja`), Simplified Chinese (`zh`), and Traditional Chinese (`zh_hant`), you can change the tokenizer’s behavior with a custom tokenization dictionary.
- Dictionaries for languages which tokenize based on whitespace and punctuation map inflections of words to their dictionary form, such as “viewing” mapping to “view” in English. The same is true of Japanese. You can use a custom dictionary to modify which words map to which stems.
- 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.

Custom dictionaries are validated when you install them so that errors do not occur every time you use the dictionary. Duplicate entries are not detected; such entries are unnecessary but do not cause errors. Validation does not detect non-Latin characters in a dictionary for a Latin based language such as English.

When you configure a dictionary for a language, you are also associating the dictionary with the lexer (for a tokenization dictionary) or stemmer (for a stemming dictionary) configured for that language. Each lexer or stemmer plugin has its own tokenization or stemming rules, so the "modifications" to those rules implied by a custom dictionary do not necessarily make sense for a different plugin.

> **Note:** If you change the lexer or stemmer configured for a language, you must reinstall the dictionary to update the lexer/stemmer-to-dictionary association.

You can create privileges to provide fine-grained control over who can manage the custom dictionary associated with a given stemmer or lexer plugin. For more details, see "Custom Dictionary Security Considerations" on page 776.

Custom dictionaries are stored in the data directory, so they survive MarkLogic server upgrades.

> **Note:** You should reindex if you change a custom dictionary.

19.2 Custom Dictionary Format

A custom dictionary can only be expressed as XML. A custom dictionary consists of a `<dictionary/>` root element with zero or more `<entry/>` child elements. Use the following structure for constructing a custom dictionary:

```
<dictionary xmlns="http://marklogic.com/xdmp/custom-dictionary">
  <entry>
    <word>wordToBeStemmed</word>
    <stem>theStem</stem>
    <pos>partOfSpeech</pos>
  </entry>
</dictionary>
```

The child elements of a dictionary entry have the following meaning:

Element	Description
word	Required. The word to be stemmed or identified as a token. The element value must not be empty.
stem	Required. The stem for the word specified in `<word/>`. The element value must not be empty. This value is not used in tokenization dictionaries.
pos	Optional. The part of speech classification of the word in `<word/>`. This is used primarily for languages without space-separated words, such as Chinese and Japanese. The element value must be one of the following values: `Adj` (adjective), `Adv` (adverb), `Interj` (interjection), `Nn` (noun), `NN-Prop` (proper noun), `Verb` (verb). If this element is not present, proper noun (`NN-Prop`) is assumed.

Stemming and tokenization dictionaries use the same format. For a tokenization dictionary, a dictionary entry effectively tells the tokenizer “this is a word token”.

Japanese ("ja"), Simplified Chinese ("zh"), and Traditional Chinese ("zh_Hant") use a linguistic tokenizer to divide text into tokens (words and punctuation). A custom dictionary affects the tokenizer for these languages. For Japanese, a custom dictionary also affects the stemmer. For all of these languages, a custom dictionary entry may have an optional `cdict:pos` element to give the part of speech for that word.

19.3 Custom Dictionary Function Summary

The custom dictionary interfaces are available to your application through the `custom-dictionary` XQuery library module. To use the functions in your own code, you must bring the module into scope, as shown below:

Language	Example
XQuery	`import module namespace cdict = "http://marklogic.com/xdmp/custom-dictionary" at "/MarkLogic/custom-dictionary.xqy";`
Server-Side JavaScript	`const cdict = require('/MarkLogic/custom-dictionary');`

The dictionary library module contains functions for performing the following tasks. For more details on each function, see the *MarkLogic XQuery and XSLT Function Reference* or *JavaScript Reference Guide*.

Task	Function
Insert or update a custom dictionary	XQuery: `cdict:dictionary-write` JavaScript: `cdict.dictionaryWrite`
Retrieve a custom dictionary	XQuery: `cdict:dictionary-read` JavaScript: `cdict.dictionaryRead`
Delete a custom dictionary	XQuery: `cdict:dictionary-delete` JavaScript: `cdict.dictionaryDelete`
Get a list of licensed languages	XQuery: `cdict:get-languages` JavaScript: `cdict.getLanguages`

19.4 Example: Managing a Custom Dictionary in XQuery

This section walks you through installing, updating and deleting a custom dictionary using XQuery. See the following topics for details:

- Install the Dictionary
- Modify and Update the Dictionary
- Delete the Dictionary

19.4.1 Install the Dictionary

The following example installs a custom stemming dictionary for English. The dictionary contains two entries: One that specifies the stem of “Furbies” is “Furby”, and one that specifies the stem of “servlets” is “servlet”.

```
xquery version "1.0-ml";
import module namespace cdict =
"http://marklogic.com/xdmp/custom-dictionary"
  at "/MarkLogic/custom-dictionary.xqy";

let $dict :=
  <cdict:dictionary xmlns:cdict="http://marklogic.com/xdmp/custom-dictionary">
    <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>
return cdict:dictionary-write("en", $dict)
```

Since no `tokenization` parameter is passed to the function, the dictionary is installed as a stemming-only dictionary by default.

19.4.2 Modify and Update the Dictionary

The following example reads back the dictionary created in "Install the Dictionary" on page 668, modifies it, and updates the installed dictionary. The dictionary is modified by removing the entry for "servlets" and adding an entry for "meetings".

To update a dictionary, you must make a copy and apply your changes to the constructed copy. You cannot use operations such as `xdmp:node-replace` because you are modifying an in-memory element, not a node in the database.

```
xquery version "1.0-ml";
import module namespace cdict =
"http://marklogic.com/xdmp/custom-dictionary"
  at "/MarkLogic/custom-dictionary.xqy";

let $current-dict := cdict:dictionary-read("en")
let $new-dict :=
  element {fn:node-name($current-dict)} {
    for $entry in $current-dict//*:entry return
      if ($entry/*:word eq "servlets") then ()
      else element {fn:node-name($entry)} {
        $entry/@*,
        $entry/*
      },
    <cdict:entry xmlns:cdict="http://marklogic.com/xdmp/custom-dictionary">
      <cdict:word>meeting</cdict:word>
      <cdict:stem>meeting</cdict:stem>
    </cdict:entry>
}
return cdict:dictionary-write("en", $new-dict)
```

If you read back the updated dictionary with `cdict:dictionary-read`, you should see output similar to the following:

```
<cdict:dictionary xmlns:cdict="http://marklogic.com/xdmp/custom-dictionary">
  <cdict:entry>
    <cdict:word>Furbies</cdict:word>
    <cdict:stem>Furby</cdict:stem>
```

```
  </cdict:entry>
  <cdict:entry>
    <cdict:word>meeting</cdict:word>
    <cdict:stem>meeting</cdict:stem>
  </cdict:entry>
</cdict:dictionary>
```

19.4.3 Delete the Dictionary

The following example deletes the dictionary created in “Install the Dictionary” on page 668.

```
xquery version "1.0-ml";
import module namespace cdict =
"http://marklogic.com/xdmp/custom-dictionary"
  at "/MarkLogic/custom-dictionary.xqy";

cdict:dictionary-delete("en")
```

Calling the function again (when there is no custom dictionary installed for English) has no effect.

19.5 Example: Managing a Custom Dictionary in JavaScript

This section walks you through installing, updating and deleting a custom dictionary using Server-Side JavaScript. See the following topics for details:

- Install the Dictionary
- Modify and Update the Dictionary
- Delete the Dictionary

19.5.1 Install the Dictionary

The following example installs a custom stemming dictionary for English. The dictionary contains two entries: One that specifies the stem of “Furbies” is “Furby”, and one that specifies the stem of “servlets” is “servlet”.

```
'use strict';
const cdict = require('/MarkLogic/custom-dictionary');

const dict = fn.head(xdmp.unquote(
  '<cdict:dictionary
xmlns:cdict="http://marklogic.com/xdmp/custom-dictionary">' +
    '<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>'
)).root;
cdict.dictionaryWrite('en', dict);
```

Since no `tokenization` parameter is passed to the function, the dictionary is installed as a stemming-only dictionary by default.

19.5.2 Modify and Update the Dictionary

The following example reads back the dictionary created in "Install the Dictionary" on page 670, modifies it, and updates the installed dictionary. The dictionary is modified by removing the entry for "servlets" and adding an entry for "meetings".

To update a dictionary, you must make a copy and apply your changes to the constructed copy. You cannot use operations such as `xdmp.nodeReplace` because you are modifying an in-memory element, not a node in the database.

Manipulating XML is much simpler in XQuery than in JavaScript, so you might find it easier to write dictionary data manipulation code using XQuery. The example below uses the `NodeBuilder` interface to create a modified copy of the dictionary in JavaScript. For an equivalent example in XQuery, see "Modify and Update the Dictionary" on page 669.

```
'use strict';
const cdict = require('/MarkLogic/custom-dictionary');

const dict = cdict.dictionaryRead('en');
const builder = new NodeBuilder();

// start a new dictionary
builder.startElement(
  'dictionary',
  'http://marklogic.com/xdmp/custom-dictionary');

// Copy all the entry elems except the one for "servlets"
for (let entry of dict.xpath('//*:entry')) {
  if (fn.data(fn.head(entry.xpath('*:word'))) != 'servlets') {
    builder.startElement(entry.localName, entry.namespaceURI);
    const entryChildren = entry.childNodes;
    for (i = 0; i < entryChildren.length; i++) {
      const child = entryChildren.item(i);
      builder.addElement(
        child.localName, child.textContent, child.namespaceURI);
    }
    builder.endElement();    // entry
  }
}

// Create a new entry for "meeting"
builder.startElement('entry',
'http://marklogic.com/xdmp/custom-dictionary');
```

```
builder.addElement('word', 'meeting',
'http://marklogic.com/xdmp/custom-dictionary');
builder.addElement('stem', 'meeting',
'http://marklogic.com/xdmp/custom-dictionary');
builder.endElement();    // entry

builder.endElement();    // dictionary

// Install the updated dictionary
cdict.dictionaryWrite('en', builder.toNode());
```

If you read back the updated dictionary with `cdict.dictionaryRead`, you should see output similar to the following:

```
<cdict:dictionary xmlns:cdict="http://marklogic.com/xdmp/custom-dictionary">
  <cdict:entry>
    <cdict:word>Furbies</cdict:word>
    <cdict:stem>Furby</cdict:stem>
  </cdict:entry>
  <cdict:entry>
    <cdict:word>meeting</cdict:word>
    <cdict:stem>meeting</cdict:stem>
  </cdict:entry>
</cdict:dictionary>
```

19.5.3 Delete the Dictionary

The following example deletes the dictionary created in “Install the Dictionary” on page 670.

```
'use strict';
const cdict = require('/MarkLogic/custom-dictionary');
cdict.dictionaryDelete('en');
```

Calling the function again (when there is no custom dictionary installed for English) has no effect.

19.6 Example: Exercising a Custom Dictionary

You can perform a simple test of a custom tokenization dictionary using the `cts:tokenize` XQuery function or the `cts.tokenize` JavaScript function.You can perform a simple test of a custom stemming dictionary using the `cts:stem` XQuery function or the `cts.stem` JavaScript function. You can also exercise you dictionary by performing a search against content in a configured language.

For example, suppose you have the following dictionary:

```
<cdict:dictionary xmlns:cdict="http://marklogic.com/xdmp/custom-dictionary">
  <cdict:entry>
    <cdict:word>servlets</cdict:word>
    <cdict:stem>servletti</cdict:stem>
```

```
    </cdict:entry>
</cdict:dictionary>
```

If you install this dictionary as a stemming dictionary for, say, French, then you can exercise it using the following code:

Language	Example
XQuery	`xquery version "1.0-ml";` `cts:stem("servlets", "fr")`
JavaScript	`'use strict';` `cts.stem('servlets', 'fr')`

The word “servlets” should stem to “servletti”.

If you install the same dictionary as a tokenization dictionary for French, then you can exercise it using the following code:

Language	Example
XQuery	`xquery version "1.0-ml";` `cts:tokenize("aservletse", "fr")`
JavaScript	`'use strict';` `cts.tokenize('aservletse', 'fr')`

The input should tokenize to three tokens: "a", "servlets", "e".

20.0 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

20.1 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.

MarkLogic Server server offers the XQuery built-in, `xdmp:document-filter`, and JavaScript method, `xdmp.documentFilter`, to extract and associate metadata from binary documents: These functions extract metadata and text from binary documents as XHTML. The results may be used as document properties. The extracted text contains little formatting or structure, so it is best used for search, classification, or other text processing.

20.2 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

20.2.1 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>
```

20.2.2 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.

20.2.3 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.

20.3 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

20.3.1 Archives

Formats: 7-Zip, ACE, 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, .EXE, .GZ, .ISO, JAR, LZH, .ORD, .RAR, .RPM, .SIT, .SEA, .SITX, .TAR, .TBZ2, .ZIP

20.3.2 Databases

Formats: dBase, dBase III, Microsoft Access, Paradox Database

Extensions: .DB, .DBF, .DB3, .MDB

20.3.3 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

20.3.4 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

20.3.5 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

20.3.6 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

20.3.7 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

20.3.8 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

20.3.9 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

20.3.10 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

20.3.11 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, VCalendar Electronic Calendar, 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, .ICS, .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, .VCS, .WPD, WRI, .WS, .WS2, .WSD, .XY

21.0 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

21.1 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

21.1.1 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.

21.1.2 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.
- You can only perform wildcard matches against JSON properties with text (string) values. Numbers, booleans, nulls are indexed separately in JSON. For details, see Creating Indexes and Lexicons Over JSON Documents in the *Application Developer's Guide*.

  Note: Combined punctuation-sensitive/whitespace-sensitive wildcards options require the character index and will not work with lexicon only. Lexicon expansion itself cannot handle whitespace and punctuation cases.

21.2 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 searches 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 Wildcard Indexes

21.2.1 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 248. 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.

21.2.2 Recommended Wildcard Index Settings

To enable any kind of wildcard query functionality with a good combination of performance and database size, MarkLogic recommends you enable the following index settings:

- `word searches`
- `three character searches`
- `word positions`
- `word lexicon` in the codepoint collation
- `three character word positions`

For details, see "Understanding the Wildcard Indexes" on page 686 and Understanding the Text Index Settings in the *Administrator's Guide*.

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.

If you only need wildcards against specific XML elements, XML attributes, JSON properties, or fields, you should consider using an element or field word lexicon instead of a general word lexicon. Doing so can improve the speed and accuracy of wildcard matching. Consider this option if you're primarily performing wildcard searches using the following query types or their equivalent:

- `cts:element-value-query`
- `cts:element-attribute-value-query`
- `cts:json-property-value-query`
- `cts:field-value-query`

21.2.3 Understanding the Wildcard Indexes

You configure the index settings at the database level, using the Admin Interface or Admin APIs (XQuery, Server-Side JavaScript, or REST). For details on configuring database settings and on other text indexes, see Database Settings and Text Indexing in the *Administrator's Guide*.

The following database settings can affect the performance and accuracy of wildcard searches. For details, see Understanding the Text Index Settings in the *Administrator's Guide*.

- word lexicons
- element, element attribute, and field word lexicons. (Use an element word lexicon for a JSON property).
- `three character searches`, `two character searches`, or `one character searches`. You do not need one or two character searches if three character searches is enabled.
- `three character word positions`
- `trailing wildcard searches`, `trailing wildcard word positions`, `fast element trailing wildcard searches`
- `fast element 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.

As with all indexing, choosing which indexes to use is a trade-off. Enabling 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.

If you only need to perform wildcard searches on specific elements, attributes, JSON properties, or fields, you can save some space and potentially improve accuracy by using an element, attribute, or field word lexicon instead of a general word lexicon.

Also, if you just want to apply wildcard searches to selected content, fields enable you to leave the wildcard indexes disabled at the database level, while still enabling them at the field level. For details, see Understanding Field Configurations in the *Administrator's Guide*.

21.3 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

21.3.1 Wildcarding and Stemming

Wildcard searches can be used in combination with stemming (for details on stemming, see "Understanding and Using Stemmed Searches" on page 652); 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.

21.3.2 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

21.3.2.1 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.

21.3.2.2 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") )
           )
```

21.3.2.3 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`.

22.0 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

22.1 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 697 for an example).

22.2 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*.

22.3 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

22.3.1 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.

22.3.2 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 697). 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. The permissions on a protected collection do not provide document level security; they only prevent unprivileged users from adding documents to the 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.

22.4 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
```

22.5 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 can be configured using the security module of the Admin Interface or by means of the `POST:/manage/v2/protected-collections` REST endpoint.
- Protected collections specify the roles that have read, insert and/or update permissions for the protected collection.
- Collection permissions control who can add documents to a protected collection, but they do not provide document access control. You must use document permissions to control document access. For example, a user with read permissions on a document in a protected collection can read the document whether or not they have any permissions on the collection.
- You can only add a document to a protected collection if you have insert or update permissions on the collection, as well as appropriate document permissions.

22.5.1 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.

22.5.2 Protected Collections

Protected collections enable you to control additions to a collection. They do not provide document access control.

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

Collection permissions only affect collection membership operations. Access to the documents in a collection, protected or otherwise, is controlled by document permissions. A user with no permissions on a protected collection can still read, search, update, or delete a document in the protected collection if he has sufficient document permissions.

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.

22.6 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

22.6.1 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%.

22.6.2 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.

23.0 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 in XQuery

23.1 The Thesaurus Module

There is an XQuery module to perform thesarus functions. You can use this module either in XQuery or in Server-Side JavaScript. 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 in XQuery, 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";
```

To use any of the functions in a JavaScript program, include a line similar to the following in your Server-Side JavaScript program:

```
const thsr = require("/MarkLogic/thesaurus");
```

23.2 Function Reference

The reference information for the thesaurus module functions is included in the *MarkLogic XQuery and XSLT Function Reference* and the *MarkLogic Server-Side JavaScript Function Reference* available through docs.marklogic.com.

23.3 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.

23.4 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 704.

23.5 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 in XQuery
- Loading Thesaurus Documents in JavaScript
- 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

23.5.1 Loading Thesaurus Documents in XQuery

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.

23.5.2 Loading Thesaurus Documents in JavaScript

To use a thesaurus in a Server-Side JavaScript program, 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:

```
const thsr = require("/MarkLogic/thesaurus");
declareUpdate();

thsr.load("c:\thesaurus\wordnet.xml", "/myThsrDocs/wordnet.xml")
```

This JavaScript program 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.

23.5.3 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>
            )
```

23.5.4 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/code/dictionaries). Once you download the thesaurus file, you can load it as a thesaurus document using the `thsr:load` XQuery function or the `thsr.load` JavaScript function.

Perform the following steps to download and load the *WordNet Thesaurus*:

1. Go to the code section of developer.marklogic.com and find the following page:

   ```
   http://developer.marklogic.com/code/dictionaries
   ```

2. Click the GitHub link.

3. Navigate to the thesaurus document section and find the `thesaurus.xml` document.

4. Save `thesaurus.xml` to a file (for example, `c:\thesaurus\thesaurus.xml`). Alternately, clone the GitHub repository.

5. Load the thesaurus with an XQuery statement 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")
```

Or you can load the thesaurus in JavaScript with a program similar to the following:

```
const thsr = require("/MarkLogic/thesaurus");
declareUpdate();

thsr.load("c:\thesaurus\wordnet.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.

23.5.5 Updating a Thesaurus Document

Use the following thesaurus functions to modify existing thesaurus documents:

XQuery Function	Server-Side JavaScript Function
`thsr:set-entry`	`thsr.setEntry`
`thsr:add-synonym`	`thsr.addSynonym`
`thsr:remove-entry`	`thsr.removeEntry`
`thsr:remove-term`	`thsr.removeTerm`
`thsr:remove-synonym`	`thsr.removeSynonym`

Additionally, the `thsr:insert` / `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. In XQuery, 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 in XQuery, each query must include the `import` statement in the prolog to resolve the thesaurus namespace.

23.5.6 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.

23.5.7 Example Queries Using Thesaurus Management Functions

This section includes the following examples, in both XQuery and JavaScript:

- Example: Adding a New Thesaurus Entry in XQuery
- Example: Adding a New Thesaurus Entry in JavaScript
- Example: Removing a Thesaurus Entry
- Example: Removing Term(s) from a Thesaurus in XQuery
- Example: Removing Term(s) from a Thesaurus in JavaScript
- Example: Adding a Synonym to a Thesaurus Entry in XQuery
- Example: Adding a Synonym to a Thesaurus Entry in JavaScript
- Example: Removing a Synonym From a Thesaurus in XQuery
- Example: Removing a Synonym From a Thesaurus in JavaScript

23.5.7.1 Example: Adding a New Thesaurus Entry in XQuery

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>
    <part-of-speech>noun</part-of-speech>
    <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.

23.5.7.2 Example: Adding a New Thesaurus Entry in JavaScript

The JavaScript `thsr.setEntry` function allows you to use a JavaScript object to update your thesaurs documents. The following JavaScript uses the `thsr.setEntry` function to add an entry for *Car* to the thesaurus with URI `/myThsrDocs/wordnet.xml`:

```
const thsr = require("/MarkLogic/thesaurus");
declareUpdate();

thsr.setEntry("/myThsrDocs/wordnet.xml",
{
      "term":"Car",
      "partOfSpeech":"noun",
      "synonyms":[
        {"term":"Ford",
         "partOfSpeech":"noun"
        },
        {"term":"automobile",
         "partOfSpeech":"noun"
        },
        {"term":"Fiat",
         "partOfSpeech":"noun"
        }
      ]
    });
```

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.

23.5.7.3 Example: Removing a Thesaurus Entry

The following XQuery uses the `thsr:remove-entry` function to remove the second 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])
```

Similarly, the following is a JavaScript example to do the same thing:

```
const thsr = require("/MarkLogic/thesaurus");
declareUpdate();

thsr.removeEntry("/myThsrDocs/roget.xml",
    thsr.lookup("/myThsrDocs/roget.xml","Car").toObject()[1])
```

This removes the second *Car* entry from the `/myThsrDocs/wordnet.xml` thesaurus document.

23.5.7.4 Example: Removing Term(s) from a Thesaurus in XQuery

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.

23.5.7.5 Example: Removing Term(s) from a Thesaurus in JavaScript

The following JavaScript program uses the `thsr.removeTerm` function to remove all entries for the term *Car* from the thesaurus with URI `/myThsrDocs/wordnet.xml`:

```
const thsr = require("/MarkLogic/thesaurus");
declareUpdate();

thsr.removeTerm("/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.removeTerm` function does the same as the `thsr.removeEntry` function.

23.5.7.6 Example: Adding a Synonym to a Thesaurus Entry in XQuery

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]
```

23.5.7.7 Example: Adding a Synonym to a Thesaurus Entry in JavaScript

The following JavaScript program adds the synonym *Alfa Romeo* to the thesaurus entry for *car* in the thesaurus with URI `/myThsrDocs/wordnet.xml`:

```
const thsr = require("/MarkLogic/thesaurus");
declareUpdate();

thsr.addSynonym(
  thsr.lookup("/myThsrDocs/wordnet.xml", "car"
     // requires the "elements" option because addSynonym takes an
     // element, not a JSON object
     "elements"),
    {"synonym":{
       "term": "Alfa Romeo"}
    })
```

This 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. Notice also that the lookup must specify `"elements"` because `thsr.addSynonym` requires an element entry. For example, if you wanted to add the synonym to the first *car* entry in the thesaurus, specify the first argument using the `first` variable from the following code:

```
fn.subsequence(
  thsr.lookup("/myThsrDocs/wordnet.xml", "car"), 2, 1))
```

23.5.7.8 Example: Removing a Synonym From a Thesaurus in XQuery

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]
```

23.5.7.9 Example: Removing a Synonym From a Thesaurus in JavaScript

The following JavaScript program removes the synonym *Fiat* from the thesaurus entry for *car* in the thesaurus with URI `/myThsrDocs/wordnet.xml`:

```
const thsr = require("/MarkLogic/thesaurus");
declareUpdate();

thsr.removeSynonym(thsr.lookup("/myThsrDocs/wordnet.xml", "car",
   "elements"),
   {"term": "Fiat"});
```

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:

```
fn.subsequence(
  thsr.lookup("/myThsrDocs/wordnet.xml", "car"), 2, 1))
```

23.6 Expanding Searches Using a Thesaurus in XQuery

You can expand a search to include terms from a thesaurus as well as the terms entered in the search. Consider the following XQuery statement:

```
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.

23.7 Expanding Searches Using a Thesaurus in JavaScript

You can expand a search to include terms from a thesaurus as well as the terms entered in the search. Consider the following JavaScript program:

```
const thsr = require("/MarkLogic/thesaurus");

let res = [];
for (const x of
cts.doc("/shakespeare/plays/hamlet.xml").xpath("//LINE")) {
if (cts.contains(x,
                 thsr.expand(
                   cts.wordQuery("weary"),
                   thsr.lookup("/myThsrDocs/thesaurus.xml", "weary"),
                   null, null, null ))) {
    res.push(x) } };
res;
```

This returns an array containing 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.

24.0 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

24.1 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.

24.2 Function Reference

The reference information for the spelling module functions is included in the *MarkLogic XQuery and XSLT Function Reference* and the *MarkLogic Server-Side JavaScript Function Reference* available through docs.marklogic.com. The spelling functions are divided into the following categories:

- The Spelling Built-In Functions
- The Spelling Dictionary Management Module Functions

24.2.1 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:

XQuery Function	Server-Side JavaScript Function
`spell:is-correct`	`spell.isCorrect`
`spell:suggest`	`spell.suggest`
`spell:suggest-detailed`	`spell.suggestDetailed`
`spell:double-metaphone`	`spell.doubleMetaphone`
`spell:levenshtein-distance`	`spell.levenshteinDistance`

The `spell:double-metaphone` / `spell.doubleMetaphone` and `spell:levenshtein-distance` / `spell.levenshteinDistance` functions return the raw values from which `spell:suggest` / `spell.suggest`, `spell:suggest-detailed` / `spell.levenshteinDistance`, and `spell:is-correct` / `spell.isCorrect` calculate their values.

The difference between `spell:suggest` (JavaScript `spell.suggest`) and `spell:suggest-detailed` (JavaScript `spell.suggestDetailed`) is that `spell:suggest-detailed` provides some of the information used in calculating the suggestions, and it returns a report (an XML representaiton in XQuery and an array of objects in JavaScript), 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 or JSON output.

24.2.2 The Spelling Dictionary Management Module Functions

There is an XQuery module to perform management of dictionary documents. You can use this module in either XQuery or in Server-Side JavaScript. 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 in an XQuery program, include the module declaration in the prolog of your XQuery program as follows:

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

To use the functions in this module in a JavaScript program, include a line similar to the following in your Server-Side JavaScript program:

```
const spell = require("/MarkLogic/spell");
```

24.3 Dictionary Documents

There are two types of dictionary documents you can load into MarkLogic:

- XML Dictionary Document
- JSON Dictionary Document

There are sample XML and JSON dictionary documents available at the following GitHub repository:

https://github.com/marklogic/dictionaries

You can use these documents or create your own dictionaries. You can also use the `spell:make-dictionary` / `spell.makeDictionary` spelling management function to create a dictionary document, and then use `spell:load` / `spell.load` to load the dictionary into the database.

24.3.1 XML Dictionary Document

Any XML dictionary documents loaded into MarkLogic must have the following basic structure:

```
<dictionary xmlns="http://marklogic.com/xdmp/spell">
    <metadata>
    </metadata>
    <word></word>
    <word></word>
    ......
</dictionary>
```

The `<metadata>` element is optional. Use `spell:make-dictionary` / `spell.makeDictionary` and `spell:load` / `spell.load` to create your own dictionary documents.

24.3.2 JSON Dictionary Document

Any JSON dictionary documents loaded into MarkLogic must have the following basic structure:

```
{
 "metadata": { ... },
 "words": ["word1", "word2", ... ]
}
```

The `metadata` property is optional. Use `spell:make-dictionary` / `spell.makeDictionary` and `spell:load` / `spell.load` to create your own dictionary documents.

24.4 Capitalization

The spelling lookup functions (`spell:is-correct`, `spell:suggest`, and `spell:suggest-detailed` in XQuery, `spell.isCorrect`, `spell.suggest`, and `spell.suggestDetailed` in JavaScript) 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` / `spell.isCorrect`. 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` / `spell.isCorrect`. 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` / `spell.isCorrect`. For example, *word* will not match an entry for *woRd*, nor will *WOrd*, but *WoRd* will match.
- The `spell:suggest` /`spell.suggest` functions and the `spell:suggest-detailed` / `spell.suggestDetailed` functions all observe 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` / `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` / `spell.isCorrect` and `spell:suggest` / `spell.suggest` functions before submitting your queries.

24.5 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 in XQuery
- Loading Dictionary Documents in JavaScript
- Loading one of the Sample XML Dictionaries
- Updating a Dictionary Document
- Security Considerations With Dictionary Documents

24.5.1 Loading Dictionary Documents in XQuery

To use a dictionary in a query, it must be in the database. To load a dictionary document using XQuery, use the `spell:load` function or the `spell:insert` function. 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.

24.5.2 Loading Dictionary Documents in JavaScript

To use a dictionary in a query, it must be in the database. To load a dictionary document using JavaScript, use the `spell.load` function or the `spell.insert` function. For example, to load a dictionary document with a URI `/mySpell/spell.json`, execute a program similar to the following:

```
const spell = require("/MarkLogic/spell");
declareUpdate();
spell.load("c:/dictionaries/spell.json", "/mySpell/spell.json");
```

This loads the file at the specified path into the dictionary JSON document at the specified URI.

24.5.3 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. ALternately, you can choose `small-dictionary.json`, `medium-dictionary.json`, or `large-dictionary.json` file to load a JSON dictionary.

4. Save `<size>-dictionary.xml` (or the corresponding JSON document) 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.

24.5.4 Updating a Dictionary Document

Use the following dictionary functions to modify existing dictionary documents:

- `spell:add-word` / `spell.addWord`
- `spell:remove-word` / `spell.removeWord`

The `spell:insert` XQuery function or the `spell.insert` JavaScript 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. In XQuery, 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 in XQuery, each query must include the `import` statement in the prolog to resolve the spelling module.

The following topics are about updating dictionary documents:

- Example: Adding a New Word to a Dictionary in XQuery
- Example: Adding a New Word to a Dictionary in JavaScript
- Example: Removing a Word From a Dictionary in XQuery
- Example: Removing a Word From a Dictionary in JavaScript

24.5.4.1 Example: Adding a New Word to a Dictionary in XQuery

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.

24.5.4.2 Example: Adding a New Word to a Dictionary in JavaScript

The following JavaScript program uses the `spell.addWord` function to add an entry for *albatross* to the dictionary with URI `/mySpell/Spell.json`:

```
const spell = require("/MarkLogic/spell.xqy");
declareUpdate();

spell.addWord("/mySpell/spell.json", "albatross");
```

If the `/mySpell/spell.json` dictionary has an identical entry, there will be no change to the dictionary. Otherwise, an entry for *albatross* is added to the dictionary.

24.5.4.3 Example: Removing a Word From a Dictionary in XQuery

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.

24.5.4.4 Example: Removing a Word From a Dictionary in JavaScript

The following JavaScript program uses the `spell.removeWord` function to remove the entry for *albatross* dictionary with URI `/mySpell/Spell.json`:

```
const spell = require("/MarkLogic/spell.xqy");
declareUpdate();

spell.removeWord("/mySpell/spell.json", "albatross")
```

This removes the word *albatross* from the `/mySpell/spell.json` dictionary document.

24.5.5 Security Considerations With Dictionary Documents

Dictionary documents are stored in XML or JSON 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` (JavaScript `xdmp.documentSetPermissions`) after a `spell:load` / `spell.load` operation.

24.6 Testing if a Word is Spelled Correctly

You can use the `spell:is-correct` XQuery function or the `spell.isCorrect` JavaScript function to see if a word is spelled correctly (according to the specified dictionary). Consider the following XQuery statement:

```
spell:is-correct("/mySpell/spell.xml", "alphabet")
```

This returns true because the word alphabet is spelled correctly. The following is the equivalent in JavaScript:

```
spell.isCorrect("/mySpell/spell.xml", "alphabet");
```

Now consider the following XQuery statement:

```
spell:is-correct("/mySpell/spell.xml", "alfabet")
```

This returns `false` because the word *alfabet* is not spelled correctly. The following is the equivalent in JavaScript:

```
spell.isCorrect("/mySpell/spell.xml", "alfabet");
```

24.7 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 XQuery statement:

```
spell:suggest("/mySpell/spell.xml", "alfabet")
```

Or the equivalent JavaScript program:

```
spell.suggest("/mySpell/spell.xml", "alfabet");
```

This 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 XQuery statement:

```
spell:suggest("/mySpell/spell.xml", "alphabet")
```

Or the equivalent JavaScript program:

```
spell.suggest("/mySpell/spell.xml", "alphabet");
```

This 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.

25.0 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

25.1 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 248.

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*.

25.2 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.

25.3 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 422.

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.

25.4 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:

```
xquery version "1.0-ml";

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.

26.0 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

26.1 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

26.1.1 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.

26.1.2 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.

26.1.3 Hyper-Planes and Thresholds for Classes

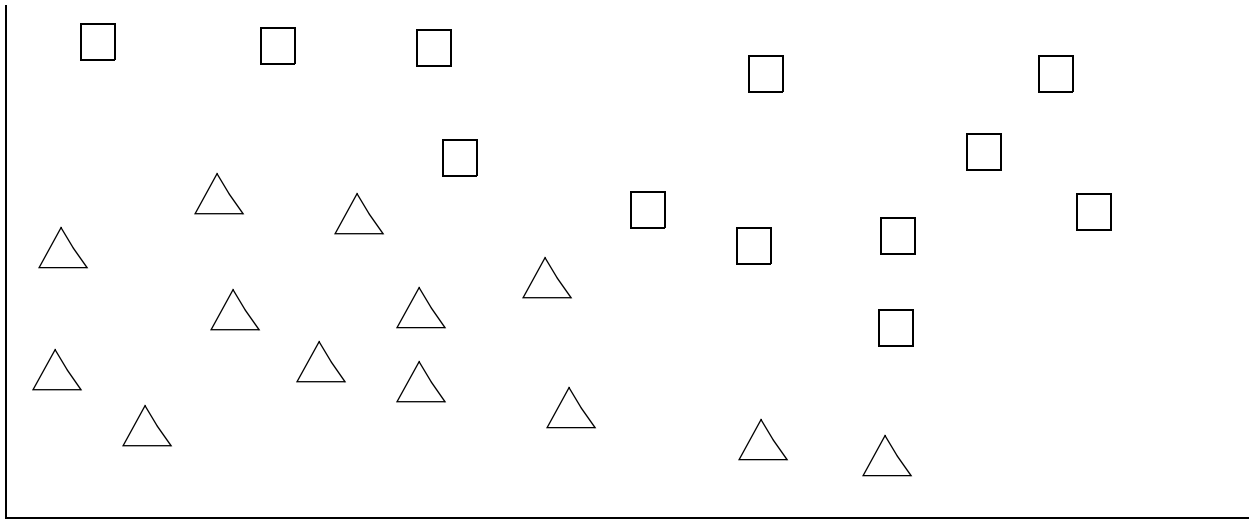
There are two main things that the computations behind the XML SVM classifier do:

- 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.

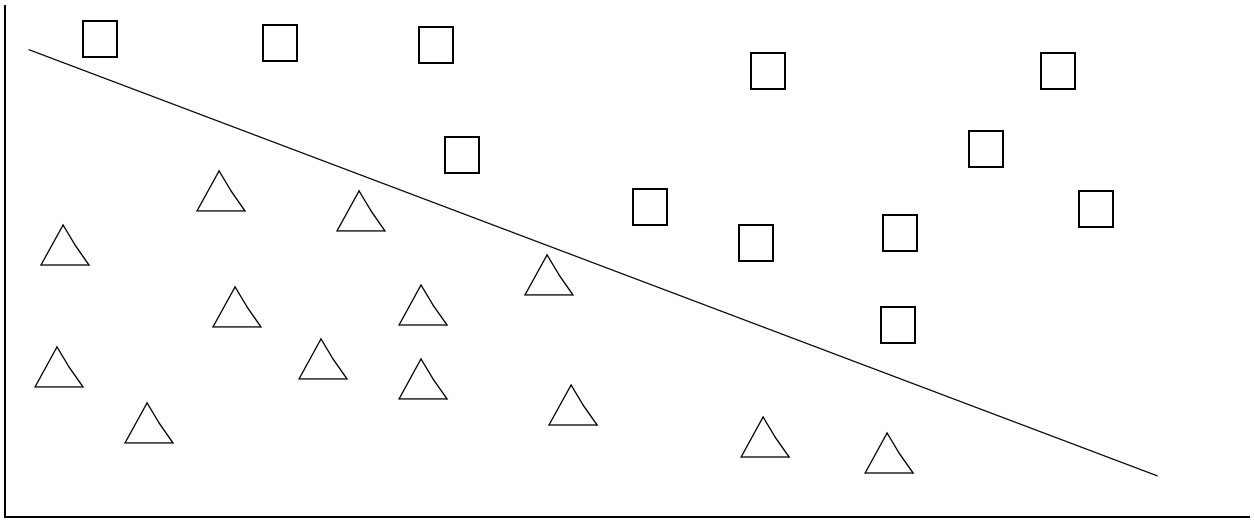
There can be any number of classes. A *term vector* is a representation of all of the terms (as defined by the index options) in a node. Therefore, classes consist of sets of term vectors which have been deemed similar enough to belong to the same class.

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.

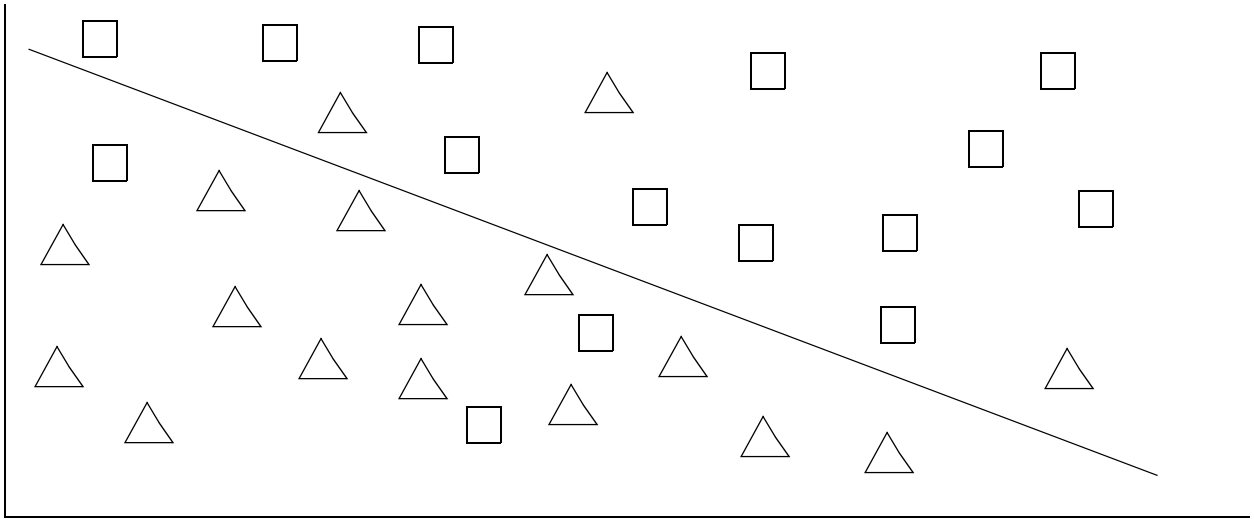


Now try to draw a line to separate the triangles from the squares. In this case, you can draw such a line that nicely divides the two classes as follows:

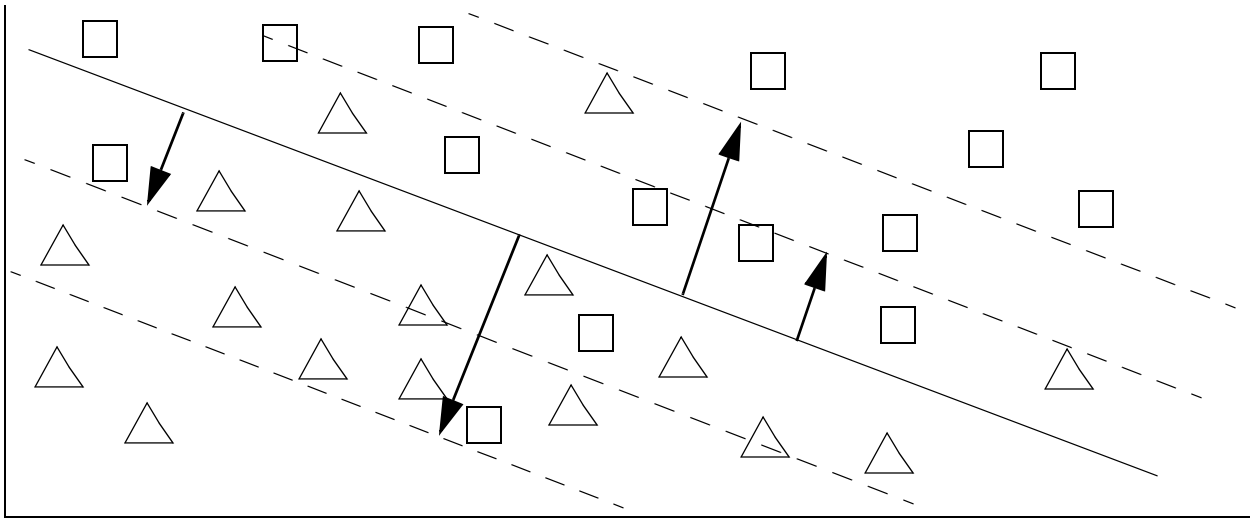


If this were three dimensions, instead of a line between the classes it would be a *plane* between the classes. When the number of dimensions grows beyond three, the extension of the plane is called a *hyper-plane*; it is the generalized representation of a boundary of a class (sometimes called the edge 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:



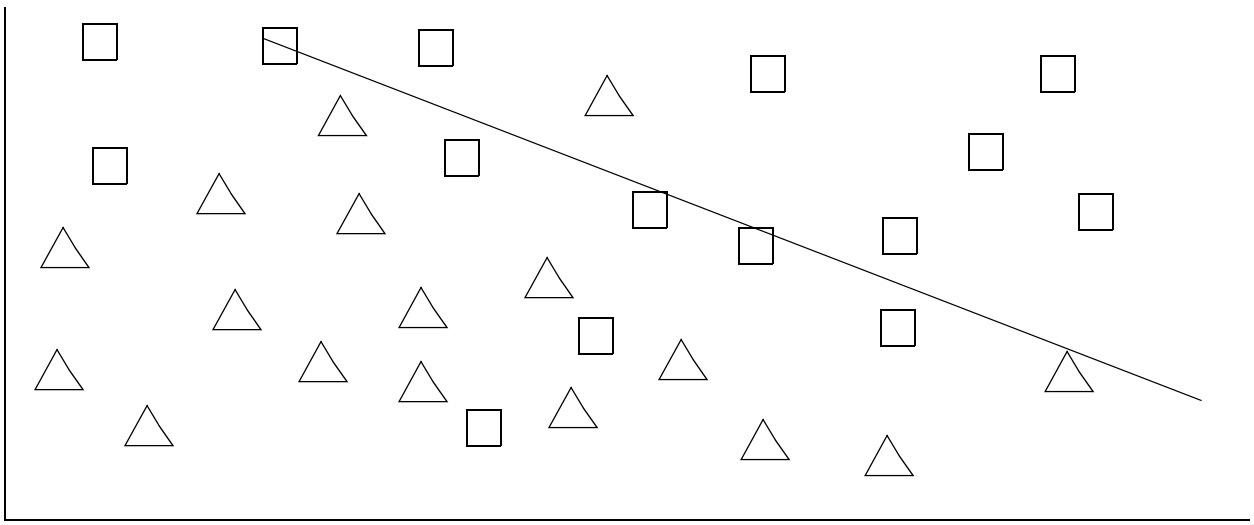
In these cases, you can draw other lines parallel to the boundaries (or in the *n*-dimensional cases, other hyper-planes). These other lines represent the *thresholds* for the classes. The distance between the boundary line and the threshold line represents the threshold value, which is a negative number 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

26.1.4 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 734 and "Methodology For Determining Thresholds For Each Class" on page 736.

26.2 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*.

26.2.1 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*.

26.2.2 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.

26.2.3 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*.

26.2.4 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.

26.2.5 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*.

26.2.6 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 727). 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 736.

26.3 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.

26.4 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

26.4.1 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 is 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.

26.4.2 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:label>
```

```
  <cts:class name="{
    xdmp:document-property(xdmp:node-uri($play))//playtype/text()}"/>
  </cts:label>
```

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>
```

26.5 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.

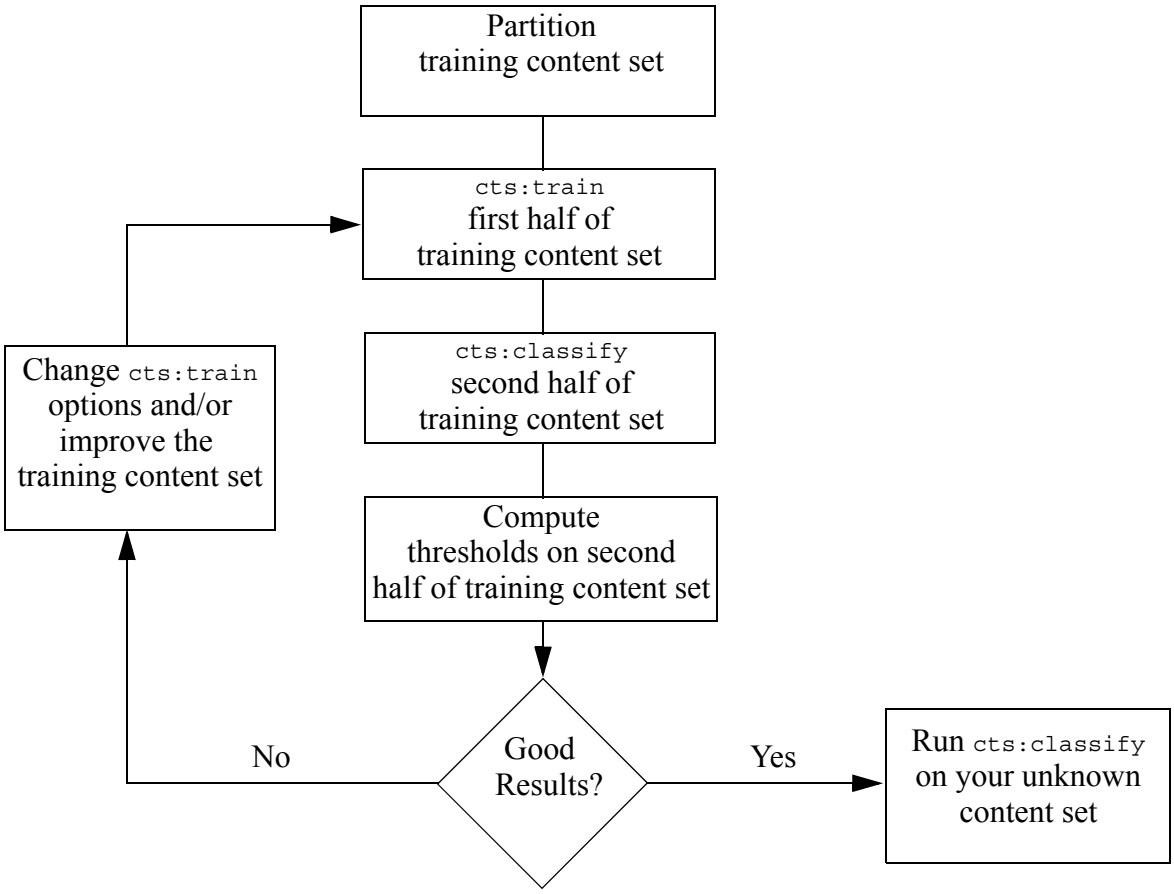
4. Run `cts:thresholds` to compute thresholds for the second half of the training set. This will further validate your training data and the parameters you set when running `cts:train` on your training data.

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:



26.6 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

26.6.1 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.

26.6.2 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*.

26.6.3 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.

26.6.4 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.

26.6.5 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.

26.6.6 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))
            //@name}"/>
      </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)
```

26.6.7 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>
```

26.6.8 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).

26.6.9 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.

27.0 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*.

27.1 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 745.

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 744, 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.

27.2 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.

27.2.1 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.

27.2.2 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.

27.3 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.

27.4 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() )
```

28.0 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
- Stemming and Tokenization Customization
- Configuring Tokenization and Stemming Plugins
- Language Support in JSON

28.1 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 755), making it possible to load documents with multiple languages. In a JSON document, the `language` or `lang` properties are used for the same purpose.

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. For MarkLogic to behave as if there is only a single language, all the following must be true:

- Your content is all in a single language.
- That language is the default language for that database.
- The XML content doesn't include any `xml:lang` attributes.
- The JSON content doesn't include any `language` or `lang` properties.

- Your queries all explicitly specify (or default to) the language in which the content is loaded.

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 describes these details, particularly the following topics:

- "Language Aspects of Loading and Updating Documents" on page 755
- "Querying Documents By Languages" on page 758

Note: You will need to manually trigger a database reindex when a license key is applied that has advanced language options. MarkLogic does not automatically reindex when the new license key is applied.

28.2 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

28.2.1 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 split into parts, each of which is called a *token*. Each token is classified as a word, punctuation, or whitespace. The process of breaking down strings into tokens is called tokenization.

Tokenization occurs during document loading as well as during query evaluation. The two processes are independent of each other. The tokenization of documents during loading affects indexing. The tokenization of query text affects how search terms are resolved. Though the processes are independent, they use the same tokenizer (for a given language).

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 XML content (for details, see "xml:lang Attribute" on page 755). For JSON content, the `language` or `lang` properties determine language-specific tokenization (for details, see Language Support in JSON).

Note the following about the way strings are tokenized in MarkLogic Server:

- You can use `cts:tokenize` XQuery function or the cts.tokenize Server-Side JavaScript function to see how text is tokenized for a given language.
- If you wrap a call to `xdmp:describe` around a call to `cts:tokenize` in XQuery, you can review both the tokens and their classification. Similarly if you wrap `xdmp.describe` around a call to `cts.tokenize` in JavaScript. 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).

- If your application has specialized tokenization requirements, you can use custom tokenizer overrides or a custom lexer to modify how characters are grouped into tokens. For details, see "Custom Tokenization" on page 785.

28.2.2 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). The purpose of stemming is to increase the recall for a search. For details about how stemming works in MarkLogic Server, including the different types of stemming available, see "Understanding and Using Stemmed Searches" on page 652. This section describes how the language settings affect stemmed searches.

Words derived from the same meaning and part of speech have the same stem (for example, "mouse" and "mice"). A word can have multiple stems if the word can be used as multiple parts of speech (for example, "play" can be both a noun and a verb in English), or if there are two words with the same spelling. If you enable advanced stemming, then stemmed searches find all of the words having the same stem as any of the stems. Advanced stemming finds multiple stems for a word.

Stemming is a language-specific operation. For example, the word "chat" is a different word in French than it is in English. In French, "chat" is a noun meaning "cat", while in English, it is a verb. In French, "chatting" is not a word, and therefore it does not stem to "chat". But in English, "chatting" does stem to "chat". Therefore, stemmed searches in one language might find different results than stemmed searches in another.

When you construct a query, you can specify a language to use for stemmed search. 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 758.

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 755.

For details about the syntax of the various `cts:query` constructors, see the *MarkLogic XQuery and XSLT Function Reference*.

28.3 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
- Protecting JSON Files That Should not be Stemmed

28.3.1 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 752 and "Stemmed Searches in Different Languages" on page 754). 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.

28.3.2 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 https://www.w3.org/TR/xml/#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 an XML 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>
```

28.3.3 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 XML content with `xml:lang` attributes, or into the JSON content with `language` or `lang` properties. 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`.
- XML documents with no `xml:lang` attribute are indexed upon load or update in the database default language.
- JSON documents with no `language` or `lang` properties are indexed upon load or update in the database default language.
- Any XML 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.
- Any JSON content within a scope that contains a `language` or `lang` property is indexed in that language. Additionally, the `language` or `lang` property is inherited by all of the descendants of that element, until another `language` or `lang` property 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.

28.3.4 Protecting JSON Files That Should not be Stemmed

The special `zxx` language code, which means no natural language present, allows users to protect their own configuration files (or other documents, elements, properties that contain no human-readable content) from customized and plugin tokenizers, as well as from stemmers. In the absence of these language codes, text will always be processed using the default database language. These files also process faster, because they are never stemmed and only use a simple tokenizer.

28.4 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

28.4.1 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. By default, a query uses the default language of the database. You can also specify a language when constructing a query. For more details, see “Tokenization and Stemming” on page 752.

For XML 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 755.

Constructed JSON nodes that contain the `language` or `lang` properties are indexed in that language. If neither of these properties is present, then they use the default language configured for the database.

28.4.2 Language-Aware Searches

All searches in MarkLogic Server are language-aware. You can specify a language when constructing a query. For example, most cts:query constructors accept a language option. If the language is not explicitly specified, MarkLogic uses the default language configured for the database. For details on the `cts:query` constructors, see “Composing cts:query Expressions” on page 248.

The language governing a query determines how to tokenize the search terms, whether stemmed search is enabled or not. If stemmed search is enabled, the language is also used to derive stems. Unstemmed searches use the unstemmed (`word searches`) indexes, which are language independent.

28.4.3 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. However, you can perform unstemmed searches without `word search` indexes using `cts:contains`. 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 binds the stemmed search results to a variable, then iterates over the results, 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 is not *guaranteed* that everything returned is in English. 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 are language-agnostic.
- Unstemmed searches use the `lang=<language>` query constructor 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")))
```

28.4.4 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 761—but for which you are not licensed). For details, see "Generic Language Support" on page 762.

You can implement a custom lexer (for tokenization) and stemmer if the default behavior for an unsupported language does not meet the needs of your application. For details, see "User-Defined Lexer Plugins" on page 795 and "Using a User-Defined Stemmer Plugin" on page 656.

28.5 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 762). The following are the supported languages:

- English
- French
- Italian
- German
- Russian
- Spanish
- Arabic
- Chinese (Simplified and Traditional)
- Korean
- Persian (Farsi)
- Dutch
- Japanese
- Portuguese
- Norwegian (Nynorsk and Bokmål)
- Swedish

For a list of base collations and character sets used with each language, see "Collations and Character Sets By Language" on page 811.

28.6 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="cz">
  <a>Some text in any language here.</a>
</doc>
```

then that document is loaded as the language `cz`, 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 "cz" :)
cts:search(doc(), cts:word-query("language", ("stemmed", "lang=en"))
```

The following search does match the document because it uses the same language:

```
(: does match because the query specifies "cz" as the language :)
cts:search(doc(), cts:word-query("language", ("stemmed", "lang=cz"))
```

Generic language support enables 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 more than the generic language support for some unsupported language, you can create a custom lexer and or stemmer plugin to enable language-specific handling. For details, see “Stemming and Tokenization Customization” on page 762.

28.7 Stemming and Tokenization Customization

This section summarizes the features available to you for customizing the stemming and tokenization processes. You can use these features separately or together.

- Tokenization Customization
- Stemming Customization

28.7.1 Tokenization Customization

With no customizations, each language has a default lexer and default tokenization dictionary associated with it. The default lexer is one of the built-in lexers described in "Built-in Lexer Plugin Reference" on page 778 and varies by language.

You can use the following tools to customize tokenization. You can use these features singly or in combination.

- Define tokenizer overrides. Overrides can affect whether a codepoint is classified as a word, punctuation or whitespace character. Overrides are applied independent of the configured lexer. For details, see "Custom Tokenizer Overrides" on page 785.
- Custom tokenization dictionary. You can install a custom dictionary to influence how text is tokenized. You configure custom dictionaries per language. For more details, see "Custom Dictionaries for Tokenizing and Stemming" on page 665.
- Custom lexer. You can use one of the built-in lexer plugins that come with MarkLogic, or create a user-defined lexer plugin using the `marklogic::LexerUDF` native C++ intefaces. You associate a custom lexer with a specific language. For details, see "User-Defined Lexer Plugins" on page 795 and "Configuring Tokenization and Stemming Plugins" on page 764.

You can use tokenization customizations in conjunction with stemming customizations. For details, see "Stemming Customization" on page 763.

Tokenization is a trusted operation. You should be selective about which users can register user-defined lexer plugins and customize language configurations.

28.7.2 Stemming Customization

With no customizations, each language has a base stemmer and stemming dictionary associated with it. The default stemmer is one of the built-in stemmer plugins that come with MarkLogic, and varies by language. For details, see "Built-in Stemmer Plugin Reference" on page 779.

You can use the following tools to customize stemming. You can use these customizations singly or in combination.

- Database text index options. For example, you can enable/disable stemmed searches, and set the level of complexity of the stemmer. For details, see Understanding the Text Index Settings in the *Administrator's Guide*.
- Custom dictionary. You can install a custom stemming dictionary to influence how words are stemmed. You configure custom dictionaries per language. For more details, see "Custom Dictionaries for Tokenizing and Stemming" on page 665.
- Custom stemmer. You can use one of the built-in stemmer plugins that come with MarkLogic, or create a user-defined stemmer plugin using the `marklogic::StemmerUDF` native C++ intefaces. You associate a custom stemmer with a specific language. For

details, see "Using a User-Defined Stemmer Plugin" on page 656 and "Configuring Tokenization and Stemming Plugins" on page 764.

You can use stemming customizations in conjunction with tokenization customizations. For details, see "Tokenization Customization" on page 763.

Stemming is a trusted operation. You should be selective about which users can register user-defined stemming plugins and customize language configurations.

28.8 Configuring Tokenization and Stemming Plugins

One way you can affect the results of tokenization and stemming is to configure a custom lexer or stemmer plugin for a language. Your customization can use either a built-in or user-defined plugin.

This section provides an overview of how to configure a custom lexer or stemmer for a language using the Custom Language Management library module. The following topics are covered:

- Function Summary for Custom Language Management
- Customization Using a Built-In Lexer or Stemmer
- Customization Using a User-Defined Lexer or Stemmer
- Example: Adding Configuration for a Language
- Example: Removing Configuration for a Language
- Example: Resetting Configuration for All Languages
- Understanding Stemming Delegation
- Custom Dictionary Security Considerations
- Built-in Lexer Plugin Reference
- Built-in Stemmer Plugin Reference

For more information on creating user-defined lexer and stemmer plugins, see the following topics:

- "User-Defined Lexer Plugins" on page 795
- "Using a User-Defined Stemmer Plugin" on page 656
- Using Native Plugins in the *Application Developer's Guide*

28.8.1 Function Summary for Custom Language Management

Lexer and stemmer plugin configuration is done through the custom language management library module. The module includes the following functions. For more details, see the *XQuery/XSLT Function Reference* or the *MarkLogic Server-Side JavaScript Function Reference*.

Function	Description
`clang:language-config-read` (XQuery) `clang.languageConfigRead` (JavaScript)	Read the current custom language configuration. You should always begin your configuration changes by calling this function.
`clang:language-config-write` (XQuery) `clang.languageConfigWrite` (JavaScript)	Commit custom language configuration changes. Your changes will not take effect unless you call this function. **Note**: Calling this function restarts MarkLogic.
`clang:language-config-delete` (XQuery) `clang.languageConfigDelete` (JavaScript)	Remove all custom language configuration from your MarkLogic installation. **Note**: Calling this function restarts MarkLogic.
`clang:update-user-language` (XQuery) `clang.updateUserLanguage` (JavaScript)	Modify a language config element to add/replace configuration for a specific language. Your change will not take effect until you call `clang:language-config-write` (XQuery) or `clang.languageConfigWrite` (JavaScript).
`clang:delete-user-language` (XQuery) `clang.deleteUserLanguage` (JavaScript)	Modify a language config element to remove configuration for a specific language. Your change will not take effect until you call `clang:language-config-write` (XQuery) or `clang.languageConfigWrite` (JavaScript).
`clang:user-language` (XQuery) `clang.userLanguage` (JavaScript)	Construct a custom language-to-plugin binding that can be used to update the custom language configuration. This is the "unit of change" for `clang:update-user-language` and `clang.updateUserLanguage`.
`clang:user-language-plugin` (XQuery) `clang.userLanguagePlugin` (JavaScript)	Construct a custom lexer/stemmer plugin reference that can be used to update the configuration for a language.

Function	Description
`clang:lexer` (XQuery) `clang.lexer` (JavaScript)	Construct a reference to a lexer capability in a native plugin. Use the output of this function as input to `clang:user-language-plugin` or `clang.userLanguagePlugin`.
`clang:stemmer` (XQuery) `clang.stemmer` (JavaScript)	Construct a reference to a stemmer capability in a native plugin. Use the output of this function as input to `clang:user-language-plugin` or `clang.userLanguagePlugin`.

28.8.2 Customization Using a Built-In Lexer or Stemmer

This sections describes how to construct a custom lexer or stemmer configuration item based on one of the built-in lexers or stemmers, rather than on a user-defined plugin.

Setting the `library` argument of `clang:user-language-plugin` or `clang.userLanguagePlugin` to an empty string tells MarkLogic you are referencing a built-in plugin. For example, the following call constructs a stemmer configuration item based on the built-in Snowball stemmer. Notice that the first parameter (`library`) is an empty string.

```
XQuery: clang:user-language-plugin("",(),clang:stemmer("snowball"))

JavaScript: clang.userLanguagePlugin('', null, clang.stemmer('snowball')
```

The first argument of the stemmer constructor should be one of the built-in stemmer names from "Built-in Stemmer Plugin Reference" on page 779. You can configure a custom lexer at the same time by including a `clang.lexer` or `clang.lexer` configuration item as the 3rd parameter.

Note: If you associate a custom lexer dictionary with a language, you must reinstall it if you change the lexer plugin for the language. Similarly, if you associate a custom stemming dictionary with a language, you must reinstall it if you change the stemmer plugin for the language.

The following example creates a configuration item for German. The default lexer for German is ICU. The default stemmer for German is Bitext. The new configuration specifies Snowball as the custom lexer and leaves the default lexer unchanged. In addition, the Snowball stemmer is configured to use the `german2` stemming algorithm.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace clang =`
	`  "http://marklogic.com/xdmp/custom-language"`
	`  at "/MarkLogic/custom-language.xqy";`
	
	`let $stemmer :=`
	`  clang:stemmer("snowball",(),("code=german2"))`
	`let $plugin := clang:user-language-plugin("",(),$stemmer)`
	`let $german := clang:user-language("de",$plugin)`
	
	`return`
	`  clang:update-user-language(`
	`  clang:language-config-read(), $german)`

Server-Side JavaScript	`'use strict';`
	`const clang = require('/MarkLogic/custom-language');`
	
	`const stemmer = clang.stemmer(`
	`  'snowball', null, Sequence.from(['code=german2']));`
	`const plugin = clang.userLanguagePlugin('', null, stemmer);`
	`const germanConfig = clang.userLanguage('de', plugin);`
	
	`clang.updateUserLanguage(`
	`  clang.languageConfigRead(), germanConfig);`

Note that this example doesn't actually change the language configuration because it does not call `clang:language-config-write` (XQuery) or `clang.languageConfigWrite` (JavaScript).

If you run the example in Query Console, you should see output similar to the following:

```
<lang:user-languages xml:lang="zxx"
xmlns:lang="http://marklogic.com/xdmp/language">
  <lang:user-language>
    <lang:name>de</lang:name>
    <lang:plugin>
      <lang:library/>
      <lang:stemmer>
        <lang:variant>snowball</lang:variant>
        <lang:arg>code=german2</lang:arg>
      </lang:stemmer>
    </lang:plugin>
  </lang:user-language>
</lang:user-languages>
```

28.8.3 Customization Using a User-Defined Lexer or Stemmer

This describes how to construct a custom lexer or stemmer configuration item based on a user-defined plugin, rather than on one of the built-in plugins. A user-defined lexer or stemmer must be installed as a native plugin before you can use it.

When you construct a lexer (or stemmer) configuration item for a user-defined plugin, you must identify the native plugin and the capability from the plugin library that exposes the `LexerUDF` or `StemmerUDF` implementation.

For a lexer, set the `variant` argument of `clang:lexer` or `clang.lexer` to a `LexerUDF` capability registered by plugin. For a stemmer, set the `variant` argument of `clang:stemmer` or `clang.stemmer` to a `StemmerUDF` capability registered by plugin. For both, set the `library` argument to "*plugin_path*/*plugin_id*".

For example, if you install a plugin with the path "native" and plugin id "sampleplugin", and the lexer UDF capability registered by the plugin is named "sample_lexer", then you'd construct a lexer configuration item for it as follows:

```
XQuery: clang:lexer("sample_lexer", (), (), "native/sampleplugin")

JavaScript: clang.lexer('sample_lexer', null, null, 'native/sampleplugin')
```

If you configure both a stemmer and lexer from the same native plugin, you can set the plugin library reference ("native/sampleplugin") in `clang:user-language-plugin` or `clang.userLanguagePlugin` instead. For example:

```
XQuery: clang:user-language-plugin(
          "native/sampleplugin",
          clang:lexer("sample_lexer"),
          clang:stemmer("sample_stemmer"))

JavaScript: clang.userLanguagePlugin('
              'native/sampleplugin',
              clang.lexer('sample_lexer'),
              clang.stemmer('sample_stemmer'));
```

When a library is specified in both the lexer/stemmer constructor and the language plugin constructor, the library in the lexer/stemmer takes precedence.

Note: If you associate a custom lexer dictionary with a language, you must reinstall it if you change the lexer plugin for the language. Similarly, if you associate a custom stemming dictionary with a language, you must reinstall it if you change the stemmer plugin for the language.

The following example creates a configuration item for German. The default lexer for German is ICU. The default stemmer for German is Bitext. The new configuration specifies a user-defined lexer named “sample_lexer” as the custom lexer and leaves the default stemmer unchanged. Assume the plugin configuration described above.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace clang =`
	`  "http://marklogic.com/xdmp/custom-language"`
	`  at "/MarkLogic/custom-language.xqy";`
	
	`let $lexer :=`
	`  clang:lexer("sample_lexer",(),(), "native/sampleplugin")`
	`let $plugin := clang:user-language-plugin("", $lexer, ())`
	`let $german := clang:user-language("de", $plugin)`
	
	`return`
	`  clang:update-user-language(`
	`  clang:language-config-read(), $german)`

Server-Side JavaScript	`'use strict';`
	`const clang = require('/MarkLogic/custom-language');`
	
	`const lexer = clang.lexer(`
	`  'sample_lexer', null, null, 'native/sampleplugin');`
	`const plugin = clang.userLanguagePlugin('', lexer, null);`
	`const germanConfig = clang.userLanguage('de', plugin);`
	
	`clang.updateUserLanguage(`
	`  clang.languageConfigRead(), germanConfig);`

Note that this example doesn’t actually change the language configuration because it does not call `clang:language-config-write` (XQuery) or `clang.languageConfigWrite` (JavaScript).

If you run the example in Query Console, you should see output similar to the following:

```
<lang:user-languages xml:lang="zxx"
    xmlns:lang="http://marklogic.com/xdmp/language">
  <lang:user-language>
    <lang:name>de</lang:name>
    <lang:plugin>
      <lang:library/>
      <lang:lexer>
        <lang:library>native/sampleplugin</lang:library>
        <lang:variant>sample_lexer</lang:variant>
      </lang:lexer>
    </lang:plugin>
  </lang:user-language>
</lang:user-languages>
```

28.8.4 Example: Adding Configuration for a Language

Use the `clang:user-language-plugin` XQuery function or the `clang.userLanguagePlugin` Server-Side JavaScript function to define a binding between a language and custom tokenization and stemming plugins. For more details, see "Customization Using a Built-In Lexer or Stemmer" on page 766 and "Customization Using a User-Defined Lexer or Stemmer" on page 768.

To put the configuration change into effect, use the following pattern. A complete example follow.

Language	Example

XQuery	`clang:language-config-write(`
	`  clang:update-user-language(`
	`  clang:language-config-read(), $changed-lang)`
	`  )`

JavaScript	`clang.languageConfigWrite(`
	`  clang.updateUserLanguage(`
	`  clang.languageConfigRead(), changedLang));`

This operation is an overwrite: Any previous configuration for the language will be replaced. Thus, if you are going to configure both a lexer and a stemmer for a language, do it in a single call to `clang:update-user-language` or `clang.updateUserLanguage`.

> **Note:** Calling `clang:language-config-write` or `clang.languageConfigWrite` causes MarkLogic to restart.

The following example configures a custom stemmer and lexer for the Catalan language using a user-defined plugin. Assume the plugin registers a lexer named “sample_lexer” and a stemmer named “sample_stemmer.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace`
	`clang="http://marklogic.com/xdmp/custom-language"`
	`  at "/MarkLogic/custom-language.xqy";`
	
	`(: Construct custom lexer and stemmer bindinf for Catalan :)`
	`let $catalan :=`
	`  clang:user-language("ca",`
	`    clang:user-language-plugin("native/sampleplugin",`
	`      clang:lexer("sample_lexer"),`
	`      clang:stemmer("sample_stemmer")`
	`    )`
	`  )`
	`(: Get the existing config so we update it :)`
	`let $existing :=  clang:language-config-read()`
	`return`
	`  (: Update the current config and commit the changes :)`
	`  (: NOTE: Causes a restart :)`
	`  clang:language-config-write(`
	`    clang:update-user-language($existing, $catalan)`
	`  )`

JavaScript	`'use strict';`
	`const clang = require('/MarkLogic/custom-language.xqy');`
	
	`// Construct a custom lexer and stemmer binding for Catalan`
	`const catalan = clang.userLanguage(`
	`  'ca',`
	`  clang.userLanguagePlugin(`
	`    'native/sampleplugin',`
	`    clang.lexer('sample_lexer'),`
	`    clang.stemmer('sample_stemmer'))`
	`  );`
	`// Get the existing config so we can update it`
	`const existing = clang.languageConfigRead();`
	
	`// Update the current config and commit the changes.`
	`// NOTE: Causes a restart`
	`clang.languageConfigWrite(`
	`  clang.updateUserLanguage(existing, catalan));`

You can configure just a lexer or just a stemmer for a language by including just that reference when calling `clang:user-language-plugin` or `clang.userLanguagePlugin`. For example, the following code only configures a custom stemmer.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace`
	`clang="http://marklogic.com/xdmp/custom-language"`
	`  at "/MarkLogic/custom-language.xqy";`
	
	`let $catalan :=`
	`  clang:user-language("ca",`
	`  clang:user-language-plugin("native/sampleplugin",`
	`  (), clang:stemmer("sample_stemmer")`
	`  )`
	`  )`
	`let $existing :=  clang:language-config-read()`
	`return`
	`  clang:language-config-write(`
	`  clang:update-user-language($existing, $catalan)`
	`  )`

JavaScript	`'use strict';`
	`const clang = require('/MarkLogic/custom-language.xqy');`
	
	`const catalan = clang.userLanguage(`
	`  'ca',`
	`  clang.userLanguagePlugin(`
	`  'native/sampleplugin', null,`
	`  clang.stemmer('sample_stemmer'))`
	`  );`
	`const existing = clang.languageConfigRead();`
	
	`clang.languageConfigWrite(`
	`  clang.updateUserLanguage(existing, catalan));`

To configure a lexer and stemmer from different plugin libraries for the same language, specify the plugin path to the lexer and stemmer reference constructors. For example, the following code configures a lexer and a stemmer from two different plugins:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace`
	`clang="http://marklogic.com/xdmp/custom-language"`
	`  at "/MarkLogic/custom-language.xqy";`
	
	`let $catalan :=`
	`  clang:user-language("ca",`
	`    clang:user-language-plugin("",`
	`      clang:lexer("my_lexer",(),(), "plugin1/lexers"),`
	`      clang:stemmer("my_stemmer", (), (), "plugin2/stemmers")`
	`    )`
	`  )`
	`let $existing :=  clang:language-config-read()`
	`return`
	`  clang:language-config-write(`
	`    clang:update-user-language($existing, $catalan)`
	`  )`

JavaScript	`'use strict';`
	`const clang = require('/MarkLogic/custom-language.xqy');`
	
	`const catalan = clang.userLanguage(`
	`  'ca',`
	`  clang.userLanguagePlugin(`
	`    '',`
	`    clang.lexer('my_lexer', null, null, 'plugin1/lexers'),`
	`    clang.stemmer('my_stemmer', null, null, 'plugin2/stemmers'))`
	`  );`
	`const existing = clang.languageConfigRead();`
	
	`clang.languageConfigWrite(`
	`  clang.updateUserLanguage(existing, catalan));`

28.8.5 Example: Removing Configuration for a Language

Use the `clang:delete-user-language` XQuery function or the `clang.deleteUserLanguage` JavaScript function to remove the custom configuration for a specific language. You must call `clang:language-config-write` (XQuery) or `clang.languageConfigWrite` (JavaScript) for your change to take effect, and doing so will cause MarkLogic to restart.

The following example removes the configuration for the Catalan language (language code “ca”).

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace`
	`clang="http://marklogic.com/xdmp/custom-language"`
	`  at "/MarkLogic/custom-language.xqy";`
	
	`let $language := "ca"`
	`let $existing :=  clang:language-config-read()`
	`return`
	`  clang:language-config-write(`
	`    clang:delete-user-language($existing, $language)`
	`  )`

JavaScript	`'use strict';`
	`const clang = require('/MarkLogic/custom-language.xqy');`
	
	`const language = 'ca';`
	`const existing = clang.languageConfigRead();`
	
	`clang.languageConfigWrite(`
	`  clang.deleteUserLanguage(existing, language));`

28.8.6 Example: Resetting Configuration for All Languages

To remove custom stemmer and lexer bindings for all languages, use the `clang:language-config-delete` XQuery function or the `clang.languageConfigDelete` Server-Side JavaScript function.

Note: Calling these functions restarts MarkLogic.

The following example code removes all language customizations and restarts the server.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace`
	`clang="http://marklogic.com/xdmp/custom-language"`
	`  at "/MarkLogic/custom-language.xqy";`
	
	`clang:language-config-delete()`

JavaScript	`'use strict';`
	`const clang = require('/MarkLogic/custom-language.xqy');`
	
	`clang.languageConfigDelete();`

28.8.7 Understanding Stemming Delegation

You can use delegation to control whether stemming consults the default stemmer in addition to the custom plugin for a language. Delegation can be controlled at two levels:

- User-defined stemming plugins and some built-in plugins include a delegation control on their interface. For example, the built-in Bitext plugin accepts a `delegation` option, and the `StemmerUDF` interface for user-defined plugins has a `delegate` method.
- The `clang:user-language-plugin` XQuery function and the `clang.userLanguagePlugin` JavaScript function accept a boolean `delegate` parameter. When set to true (the default), the stemming process asks the plugin whether or not to delegate.

If delegation is enabled at the language plugin configuration level, then the stemming process will consult the custom plugin about whether or not to delegate. For example, it will call the `delegate` method on `StemmerUDF`. If delegation is disabled at the language plugin configuration level, then the stemming process will not consult the custom plugin and will never delegate to the default stemmer.

Delegation has the following effect on the stemming results. The first column indicates whether or not the `delegate` parameter of `clang:user-language-plugin` or `clang.userLanguagePlugin` is set to true. The second column indicates whether or not the custom plugin agrees to delegate; for example, whether `StemmerUDF::delegate` returns true.

Lang Config Delegate	Plugin Says to Delegate	Result
true	true	stems = (stems from plugin + stems from default) if no stems are found, word is self-stemming
true	false	stems = stems from plugin if no stems are found, word is self-stemming
false	N/A	stem = stem from plugin if no stems are found, word is self-stemming

The following table contains examples of the stemming result with various delegation and stem count combinations. The "with Delegation" column signifies the plugin was consulted and agreed to delegation. The "without Delegation" column signifies either the plugin was not consulted or the plugin did not agree to delegation.

Input	Custom Plugin Stems	Default Stems	Final Result with Delegation	Final Result without Delegation
moogled	moogle		moogle	moogle
pabbling	peeble	pabble	peeble, pabble	peeble
furben		furby	furby	furben (self-stem)
zorks			zorks (self-stem)	zorks (self-stem)

A custom plugin determines its own delegation policy. For example, a plugin might choose among policies such as "always" (delegate regardless of the number of stems found), "never" (never delegate), or "on empty" (delegate only if the plugin found no stems).

28.8.8 Custom Dictionary Security Considerations

When you configure a language to use a custom user-defined stemmer or lexer, and also associate a custom dictionary with the language, then you can create special security privileges to enable finer control over who can administer the dictionary.

A custom dictionary is associated with both a language and a specific stemmer or lexer plugin. The lexer or stemmer is implicit in the configuration of the language. Usually, any user with the `custom-dictionary-admin` role or equivalent privileges can add, update, or delete a custom dictionary for any language-stemmer or language-lexer configuration.

You can create a privileges of the following form to make it possible to control dictionary management on a per stemmer/lexer basis.

```
http://marklogic.com/xdmp/privileges/custom-dictionary-admin/library
http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/library
```

Where *`library`* is of the form *`plugin_path/plugin_id`* and identifies a user-defined lexer or stemmer plugin.

For example, if you install a user-defined lexer plugin with the plugin path “native” and the plugin id “sampleplugin”, then you would create a privileges of the following form:

```
http://marklogic.com/xdmp/privileges/custom-dictionary-admin/native/sampleplug
in
http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/native/sam
pleplugin
```

MarkLogic will not create these privilege for you, but it will check for and enforce them if the privileges exist.

28.8.9 Built-in Lexer Plugin Reference

The table below lists the built-in lexers (tokenizers), which languages each one is configured for by default, and what configuration options (if any) are available for customization. Use the configuration lexer name and configuration options when calling the `clang:lexer` XQuery function or the `clang.lexer` JavaScript function; for details, see "Customization Using a Built-In Lexer or Stemmer" on page 766.

Lexer Name	Description	
simple lexer	The default lexer for English, Norwegian, and languages without advanced support. You cannot specify this lexer as a custom plugin, and it has no configuration options.	
icu	Default tokenizer for most licensed languages. Users might want to switch to this tokenizer for English to pick up better apostrophe and contraction handling, or for languages without advanced support. This lexer accepts no extra arguments.	
kytea	Default tokenizer for Chinese. With the appropriate language model, this tokenizer could be used for other languages. You can customize the behavior of this lexer using the following arguments:	
	model_filename	Required. The name of a model file in the *MARKLOGIC_DIR*/Lang directory of all hosts in the cluster. The model is used for tokenization. Only UTF-8 models are supported. Create a model file using the KyTea tools on a corpus, possible augmented with dictionaries. KyTea offers Japanese models alternative models for Chinese at http://www.phontron.com/kytea/model.html.
atilika	Default tokenizer for Japanese. You can customize the behavior of this lexer using the following arguments:	
	`search-mode` `normal-mode`	Specify the handling of compound words: `search-mode` breaks up compound words, while `normal-mode` does not. Default: `search-mode`.

28.8.10 Built-in Stemmer Plugin Reference

The tables below list the built-in stemmers. Use the stemmer name and options when constructing a stemmer configuration item using the `clang:stemmer` XQuery function or the `clang.stemmer` JavaScript function.

MarkLogic uses the following built-in stemmers by default:

Stemmer Name	Description
simple stemmer	The default stemmer for languages without advanced stemming support.
bitext	Default stemmer for all languages with advanced stemming support except Chinese and Japanese. Chinese is not stemmed, and Japanese uses the Atilika stemmer.
snowball	Default stemmer for Danish, Finnish, Hungarian, Romanian, Tamil, and Turkish.
atilika	Default stemmer for Japanese.

See the following topics for configuration options.

- Bitext Stemmer Options
- Snowball Stemmer Options
- Atilika Stemmer Options

28.8.10.1Bitext Stemmer Options

The Bitext stemmer supports the following options that you can specify in the `args` parameter of `clang:stemmer` or `clang.stemmer`.

Bitext Option	Description
`code=`*`value`*	A language code to be passed to Bitext. This is a 3-letter code, such as `DEU` for German.
`dict=`*`value`*	Which dictionary to use. You can specify multiple dictionaries by specifying this argument multiple times. The dictionary must be in *`MARKLOGIC_DIR`*`/Lang` on all hosts in the cluster. The dictionary must be in Bitext's format.
`decompounding` `no-decompounding`	Enable/disable decompounding. If the language does not support decompounding, this is a no-op. Default: `no-decompounding`.
`delegation=`*`value`*	Whether to delegate to the base stemmer, if there is one. Allowed values: `always` (always delegate, meaning Bitext stems are always added to the base stemmer), `on-empty` (delegate to the base stemmer only if the Bitext dictionary had no entry for the word), or `never` (no delegation). Default: on-empty.
`algorithm=`*`value`*	Which stemming algorithm to use. If not specified, MarkLogic uses the default algorithm for the language. Choose from the following values: arabic, danish, dutch, english, finnish, french, german, german2, hungarian, italian, porter (Porter algorithm for English), portuguese, romanian, russian, spanish, swedish, turkish, tamil, persian, korean, english2, french2, german3, italian2, spanish2, swedish2. The values english2, french2, german3, italian2, spanish2, and swedish2 specify a lemmatizing algorithm for that language, for use with Bitext dictionaries.
`pre-stemmer=`*`value`*	Which pre-stemming algorithm to use. Pre-stemmers perform normalization on the input to make better use of the Bitext dictionaries. Choose one of the following values: `normalize_latin` (map fullwidth characters to regular Latin character; map ligatures to their components), `arabic_transliteration` (transliterate Arabic characters to ASCII. Required for Arabic since it uses transliterated dictionaries.).

Bitext Option	Description
`use-algorithm` `no-use-algorithm`	Enable/disable the stemming algorithm backing the Bitext dictionary. Default: `use-algorithm`. Does not apply to the pre-stemmer.
`use-dictionary` `no-use-dictionary`	Enable/disable the Bitext dictionary. Default: `use-dictionary` (look up entries in the dictionary). If the dictionary is disabled, the stemmer will perform pre-stemming and (if the algorithm is enabled) stemming.
`lowercase` `no-lowercase`	Enable/disable lowercasing of the input string. Many of the standard algorithms use uppercase letters as markers and will not work properly if there are uppercase letters in the input. Default: `no-lowercase`.
`nfkd` `no-nfkd`	Enable/disable NFKD normalization of the input string. Some stemming algorithms do not work correctly when the input has been NFKD normalized. Default: `no-nfkd`.

28.8.10.2Snowball Stemmer Options

The Snowball stemmer supports the following options that you can specify in the `args` parameter of `clang:stemmer` or `clang.stemmer`.

Snowball Option	Description
`code=`*`value`*	Which stemming algorithm to use. Optional. If unspecified, use the default algorithm for the language. Choose from the following values: arabic, danish, dutch, english, finnish, french, german, german2, hungarian, italian, porter (Porter algorithm for English), portuguese, romanian, russian, spanish, swedish, turkish, tamil, persian, korean, english2, french2, german3, italian2, spanish2, swedish2. The values english2, french2, german3, italian2, spanish2, and swedish2 specify a lemmatizing algorithm for that language, for use with Bitext dictionaries.

28.8.10.3Atilika Stemmer Options

The Atilika stemmer supports the following options that you can specify in the `arguments` parameter of `clang:stemmer` or `clang.stemmer`.

Atilika Option	Description
`add-reading` `no-add-reading`	Specify whether or not to add the Katakana reading as an alternative stem. Default: `no-add-reading`. (MarkLogic's default configuration for Atilika uses `add-reading`, but if you're configuring Atilika as a custom plugin, the default is `no-add-reading`.)
`delegation`	Whether or not to delegate to the base stemmer. Allowed values: `never`, `always`, `on-empty`. Default: `on-empty`.

28.9 Language Support in JSON

- Overview
- API Changes
- JSON Serialization
- Upgrade Considerations

28.9.1 Overview

Beginning with version 10.0-1 of our server, MarkLogic allows natural language in JSON to be tagged with a language other than the default database language. When a `language` or `lang` tag is present in a JSON object, all textual content in that object will be interpreted as being processed under the language referred to by the ISO code in that tag. JSON language processing is very similar to XML language processing (see the ksjdf section, for details) with the following differences:

- Because JSON has no attributes, the language applies to sibling properties, as well as children of these properties.
- The xml:lang attribute has a schema in scope, you if it isn't a valid ISO code you may get errors.
- If the `language` or `lang` tag does not contain a valid language code (per RFC4646), then it will be treated analogously to such a value for `xml:lang` today: It will be treated as an unknown language for tokenization and stemming purposes.

A JSON document is allowed to have multiple `language` or `lang` tags in its content. A JSON node containing the key `language` will be processed according to that language. All descendant nodes will be processed according to that language. Language tags may be placed at any level in the JSON and are applied in a simple hierarchical way.

```
{

    language: "en-US", description: "This is US English text",
    components: [
        “Still US english”,
        {
            language: “nl”,
            data: “Dutch stuff”
        }
        {
            language: “es”,
            data: “Spanish stuff”
        }
        {
            data: “More US English”
        }
    ]
}
```

In the above example, content indexed with a particular language will have the key for that language added to the re-indexer keys stored with the document, as is now the case with XML content.

Note: When JSON is being parsed (for example from a file), making the language tag apply to preceding siblings would be expensive and require us to parse the whole object before doing any node construction on it. Serialization will put the language child first.

28.9.2 API Changes

The fn:lang function and the underlying datamodel functions that support it now handle JSON nodes as well as XML nodes.

```
fn:lang($testlang as xs:string?, [$node as node()]) as xs:boolean
```

The function fn:lang already exists with the above signature. In previous versions, it always returns false for a JSON node. Starting with version 10.0-1 of MarkLogic, it will return `true` if the JSON node or one of its ancestors has a `lang` or `language` key that matches the $testlang per the rules defined for the xml:lang attribute on XML nodes.

For example, the following will return `true`.

```
fn:lang("en", object-node{ "language" : "en-US", "item" : "example" } )
```

28.9.3 JSON Serialization

Serialization of JSON objects will put the language tag first. (with the limitations noted above).

For example:

```
xdmp:to-json( object-node{ "item" : "example", "language" : "en-US" } )
```

will return

```
{"language":"en-US", "item":"example"}
```

28.9.4 Upgrade Considerations

It is possible that you may already have JSON documents may already have `language` or `lang` properties used for some other purpose. In that case, normal language processing attempts to look up a given language, and will treat all unknown tags as equivalent. The content of the language property itself will still be indexed normally: the issue is that content will be indexed as "unknown language" instead of "default database language". That is a potential incompatibility, and a potential risk. This risk is attenuated by the fact that some JSON formats already use `language` or `lang` for precisely the purpose we want gives us some comfort that this will not be an issue in practice. In addition, MarkLogic will only attempt to apply a language or lang property if the node is a text node.

29.0 Custom Tokenization

The process of tokenization splits text in document content and query text into parts, classified as word, whitespace, and punctuation tokens. You can customize tokenization in two ways:

- Custom Tokenizer Overrides: Customize how the tokenizer classifies tokens.
- User-Defined Lexer Plugins: Customize how the tokenizer divides text into tokens.

You can use these customizations individually or together.

29.1 Custom Tokenizer Overrides

Use custom tokenizer overrides on fields to change the classification of characters in content and query text. Character re-classification affects searches because it changes the tokenization of text blocks. The following topics are covered:

- Introduction to Custom Tokenizer Overrides
- How Character Classification Affects Tokenization
- Using xdmp:describe to Explore Tokenization
- Performance Impact of Using Tokenizer Overrides
- Defining a Custom Tokenizer Override
- Examples of Custom Tokenizer Overrides

29.1.1 Introduction to Custom Tokenizer Overrides

A custom tokenizer override enables you to change the tokenizer classification of a character when it occurs within a field. You can use this flexibility to improve search efficiency, enable searches for special character patterns, and normalize data.

> **Note:** You can only define a custom tokenizer override on a field. For details, see Overview of Fields in the *Administrator's Guide*.

As discussed in "Tokenization and Stemming" on page 752, tokenization breaks text content and query text into word, punctuation, and whitespace tokens. Built-in language specific rules define how to break text into tokens. During tokenization, each character is classified as a word, symbol, punctuation, or space character. Each symbol, punctuation, or space character is one token. Adjacent word characters are grouped together into a single word token. Word and symbol tokens are indexed; space and punctuation tokens are not.

For example, with default tokenization, a text run of the form "456-1111" breaks down into 2 word tokens and 1 punctuation token. You can use a query similar to the following one to examine the break down:

```
xquery version "1.0-ml";
xdmp:describe(cts:tokenize("456-1111"));

==> ( cts:word("456"), cts:punctuation("-"), cts:word("1111") )
```

If you define a custom tokenizer override that classifies hyphen as a character to remove, the tokenizer produces the single word token "4561111". In combination with other database configuration changes, this can enable more accurate wildcard searches or allow searches to match against variable input such as 456-1111 and 4561111. For a full example, see “Example: Improving Accuracy of Wildcard-Enabled Searches” on page 790.

Tokenization rules apply to both content and query text. Since tokenizer overrides can only be applied to fields, you must use field queries such as `cts:field-word-query` or `cts:field-value-query` to take advantage of your overrides.

You cannot override a character to its default class. For example, the space character (“ “) has class `space` by default, so you cannot define an override that classifies it as `space`.

You cannot override a composite character that decomposes into multiple characters when NFD Unicode normalization is applied.

29.1.2 How Character Classification Affects Tokenization

You can define a custom tokenizer override to classify a character as one of the following categories:

- `space`: Treat as whitespace. Whitespace is not indexed.
- `word`: Adjacent word characters are grouped into a single token that is indexed.
- `symbol`: Each symbol character forms a single token that is indexed.
- `remove`: Eliminate the character from consideration when creating tokens.

Note that re-classifying a character such that it is treated as punctuation in query text can trigger punctuation-sensitive field word and field value queries. For details, see “Wildcarding and Punctuation Sensitivity” on page 688.

The table below illustrates how each kind of override impacts tokenization:

Char Class	Example Input	Default Tokenization	Override	Result
space	`10x40`	`10x40` (word)	`admin:database-tokenizer-override ("x","space")`	`10` (word) `x` (space) `40` (word)
word	`456-1111`	`456` (word) `-` (punc.) `1111` (word)	`admin:database-tokenizer-override ("-","word")`	`456-1111` (word)
symbol	`@1.2`	`@` (punc.) `1.2` (word)	`admin:database-tokenizer-override ("@","symbol")`	`@` (symbol) `1.2` (word)
remove	`456-1111`	`456` (word) `-` (punc.) `1111` (word)	`admin:database-tokenizer-override ("-","remove")`	`4561111` (word)

29.1.3 Using xdmp:describe to Explore Tokenization

You can use `xdmp:describe` with `cts:tokenize` to explore how the use of fields and tokenizer overrides affects tokenization. Passing the name of a field to `cts:tokenize` applies the overrides defined on the field to tokenization.

The following example shows the difference between the default tokenization rules (no field), and tokenization using a field named “dim” that defines tokenizer overrides. For configuration details on the field used in this example, see “Example: Searching Within a Word” on page 792.

```
xquery version "1.0-ml";
xdmp:describe(cts:tokenize("20x40"));
xdmp:describe(cts:tokenize("20x40",(),"dim"))

==>
cts:word("20x40")
(cts:word("20"), cts:space("x"), cts:word("40"))
```

You can use this method to test the effect of new overrides.

You can also include a language as the second argument to `cts:tokenize`, to explore language specific effects on tokenization. For example:

```
xdmp:describe(cts:tokenize("20x40","en","dim"))
```

For more details on the interaction between language and tokenization, see “Language Support in MarkLogic Server” on page 751.

29.1.4 Performance Impact of Using Tokenizer Overrides

Using tokenizer overrides can make indexing and searching take longer, so you should only use overrides when truly necessary. For example, if a custom override is in scope for a search, then filtered field queries require retokenization of every text node checked during filtering.

If you have a small number of tokenizer overrides, the impact should be modest.

If you define a custom tokenizer on a field with a very broad scope, expect a larger performance hit. Choosing to re-classify commonly occurring characters like “ “ (space) as a symbol or word character can cause index space requirements to greatly increase.

29.1.5 Defining a Custom Tokenizer Override

You can only define a custom tokenizer override on a field. You can configure a custom tokenizer override in the following ways:

- Programmatically, using the function `admin:database-add-field-tokenizer-override` in the Admin API.
- Interactively, using the Admin Interface. For details, see Configuring Fields in the *Administrator’s Guide*.

  Note: Even if reindexing is disabled, when you add tokenizer overrides to a field, those tokenization changes take effect immediately, so all new queries against the field will use the new tokenization (even if it is indexed with the previous tokenization).

For example, assuming the database configuration already includes a field named `phone`, the following XQuery adds a custom tokenizer override to the field that classifies "-" as a `remove` character and `"@"` as a symbol:

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

let $dbid := xdmp:database("myDatabase")
let $config :=
  admin:database-add-field-tokenizer-override(
    admin:get-configuration(), $dbid, "phone",
    (admin:database-tokenizer-override("-","remove"),
     admin:database-tokenizer-override("@", "symbol"))
  )
return admin:save-configuration($config)
```

29.1.6 Examples of Custom Tokenizer Overrides

This section contains the following examples related to using custom tokenizer overrides:

- Example: Configuring a Field with Tokenizer Overrides
- Example: Improving Accuracy of Wildcard-Enabled Searches
- Example: Data Normalization
- Example: Searching Within a Word
- Example: Using the Symbol Classification

29.1.6.1 Example: Configuring a Field with Tokenizer Overrides

The following query demonstrates creating a field, field range index, and custom tokenizer overrides using the Admin API. You can also perform these operations using the Admin Interface.

Use this example as a template if you prefer to use XQuery to configure the fields used in the remaining examples in this section. Replace the database name, field name, included element name, and tokenizer overrides with settings appropriate for your use case.

```
(: Create the field :)
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
    at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $dbid := xdmp:database("YourDatabase")
return admin:save-configuration(
  admin:database-add-field(
    $config, $dbid,
    admin:database-field("example", fn:false())
  )
);

(: Configure the included elements and field range index :)
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
    at "/MarkLogic/admin.xqy";

let $dbid := xdmp:database("YourDatabase")
let $config := admin:get-configuration()
let $config :=
  admin:database-add-field-included-element(
    $config, $dbid, "example",
    admin:database-included-element(
      "", "your-element", 1.0, "", "", "")
  )
let $config :=
  admin:database-add-range-field-index(
    $config, $dbid,
```

```
    admin:database-range-field-index(
      "string", "example",
      "http://marklogic.com/collation/", fn:false())
  )
return admin:save-configuration($config);

(: Define custom tokenizer overrides :)
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
    at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $dbid := xdmp:database("YourDatabase")
return admin:save-configuration(
  admin:database-add-field-tokenizer-override(
    $config, $dbid, "example",
    (admin:database-tokenizer-override("(", "remove"),
     admin:database-tokenizer-override(")", "remove"),
     admin:database-tokenizer-override("-", "remove"))
  )
);
```

29.1.6.2 Example: Improving Accuracy of Wildcard-Enabled Searches

This example demonstrates using custom tokenizers to improve the accuracy and efficiency of unfiltered search on phone numbers when three character searches are enabled on the database to support wildcard searches.

Run the following query in Query Console to load the sample data into the database.

```
xdmp:document-insert("/contacts/Abe.xml",
  <person>
    <phone>(202)456-1111</phone>
  </person>);
xdmp:document-insert("/contacts/Mary.xml",
  <person>
    <phone>(202)456-1112</phone>
  </person>);
xdmp:document-insert("/contacts/Bob.xml",
  <person>
    <phone>(202)111-4560</phone>
  </person>);
xdmp:document-insert("/contacts/Bubba.xml",
  <person>
    <phone>(456)202-1111</phone>
  </person>)
```

Use the Admin Interface or a query similar to the following to enable three character searches on the database to support wildcard searches.

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
```

```
    at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
return admin:save-configuration(
  admin:database-set-three-character-searches($config,
    xdmp:database("YourDatabase"), fn:true())
)
```

With three character searches enabled, the following unfiltered search returns false positives because the query text tokenizes into two word tokens, “202” and “456”, with the wildcard applying only to the “456” token.

```
xquery version "1.0-ml";
cts:search(
  fn:doc(),
  cts:word-query("(202)456*"),
  "unfiltered")//phone/text()

==>
(456)202-1111
(202)456-1111
(202)111-4560
(202)456-1112
```

To improve the accuracy of the search, define a field on the `phone` element and define tokenizer overrides to eliminate all the punctuation characters from the field values. Use the Admin Interface to define a field with the following characteristics, or modify the query in “Example: Configuring a Field with Tokenizer Overrides” on page 789 and run it in Query Console.

Field Property	Setting
Name	`phone`
Field type	`root`
Include root	`false`
Included elements	`phone` (no namespace)
Range field index	scalar type: `string` field name: `phone` collation: `http://marklogic.com/collation/` (default) range value positions: `false` (default) invalid values: `reject` (default)
Tokenizer overrides	`remove (` (left parenthesis) `remove )` (right parenthesis) `remove -` (hyphen)

This field definition causes the phone numbers to be indexed as single word tokens such as "4562021111" and "2021114560". If you perform the following search, the false positives are eliminated:

```
xquery version "1.0-ml";
cts:search(fn:doc(),
  cts:field-word-query("phone", "(202)456*"),
  "unfiltered")//phone/text()

==>
(202)456-1111
(202)456-1112
```

If the field definition did not include the tokenizer overrides, the field word query would include the same false positives as the initial word query.

29.1.6.3 Example: Data Normalization

In "Example: Improving Accuracy of Wildcard-Enabled Searches" on page 790, custom tokenizer overrides are used to remove punctuation from phone numbers of the form (202)456-1111. The overrides provide the additional benefit of normalizing query text because the tokenizer overrides apply to query text as well as content.

If you define "(", ")", "-", and " " (space) as `remove` characters, then a phone number such as (202)456-1111 is indexed as the single word 2024561111, and all the following query text examples will match exactly in an unfiltered search:

- (202)456-1111
- 202-456-1111
- 202 456-1111
- 2024561111

The same overrides also normalize indexing during ingestion: If the input data contains all of the above patterns in the `phone` element, the data is normalized to a single word token for indexing in all cases.

For sample input and details on configuring an applicable field, see "Example: Improving Accuracy of Wildcard-Enabled Searches" on page 790.

29.1.6.4 Example: Searching Within a Word

This example demonstrates using custom tokenizer overrides to create multiple tokens out of what would otherwise be considered a single word. This makes it possible to search successfully for a portion of the word.

Suppose you have input documents that include a `dimensions` element of the form MxN, where M and N are the number of feet. For example, "10x4" is the measurement of an area that is 10 feet by 4 feet. You cannot search for "all documents which includes at least one dimension of 10 feet" because `10x4` tokenizes as a single word.

To demonstrate, run the following query in Query Console to load the sample documents:

```
xquery version "1.0-ml";
xdmp:document-insert("/plots/plot1.xml",
  <plot>
    <dimensions>10x4</dimensions>
  </plot>);
xdmp:document-insert("/plots/plot2.xml",
  <plot>
    <dimensions>25x10</dimensions>
  </plot>);
xdmp:document-insert("/plots/plot3.xml",
  <plot>
     <dimensions>5x4</dimensions>
  </plot>)
```

Next, run the following word query against the database and observe that there are no matches:

```
xquery version "1.0-ml";
cts:search(fn:doc(), cts:word-query("10"),"unfiltered")
```

Use the Admin Interface to define a field with the following characteristics, or modify the query in "Example: Configuring a Field with Tokenizer Overrides" on page 789 and run it in Query Console.

Field Property	Setting
Name	`dim`
Field type	`root`
Include root	`false`
Included elements	`dimensions` (no namespace)
Range field index	scalar type: `string` field name: `dim` collation: `http://marklogic.com/collation/` (default) range value positions: `false` (default) invalid values: `reject` (default)
Tokenizer overrides	`space x`

The field divides each `dimension` text node into two tokens, split at "x". Therefore, the following field word query now finds matches in the example documents:

```
xquery version "1.0-ml";
cts:search(fn:doc(), cts:field-word-query("dim", "10"),"unfiltered")

==>
<plot>
  <dimensions>25x10</dimensions>
</plot>
<plot>
  <dimensions>10x4</dimensions>
</plot>
```

29.1.6.5 Example: Using the Symbol Classification

This example demonstrates the value of classifying some characters as `symbol`. Suppose you are working with Twitter data, where the appearance of `@word` in Tweet text represents a user and `#word` represents a topic identifier ("hash tag"). For this example, we want the following search semantics to apply:

- If you search for a naked term, such as `NASA`, the search should match occurrences of the naked term ("`NASA`") or topics ("`#NASA`"), but not users ("`@NASA`").
- If you search for a user (`@NASA`), the search should only match users, not naked terms or topics.
- If you search for a topic (`#NASA`), the search should only match topics, not naked terms or users.

The following table summarizes the desired search results:

Query Text	Should Match	Should Not Match
`NASA`	`NASA` `#NASA`	`@NASA`
`@NASA`	`@NASA`	`NASA` `#NASA`
`#NASA`	`#NASA`	`NASA` `@NASA`

If you do not define any token overrides, then the terms `NASA`, `@NASA`, and `#NASA` tokenize as follows:

- `NASA`: `cts:word("NASA")`
- `@NASA`: `cts:punctuation("@"), cts:word("NASA")`

- `#NASA`: `cts:punctuation("#"), cts:word("NASA")`

Assuming a punctuation-insensitive search, this means all three query strings devolve to searching for just `NASA`.

If you define a tokenizer override for `@` that classifies it as a word character, then `@NASA` tokenizes as a single word and will not match naked terms or topics. That is, `@NASA` tokenizes as:

```
cts:word("@NASA")
```

However, classifying `#` as a word character does not have the desired effect. It causes the query text `#NASA` to match topics, as intended, but it also excludes matches for naked terms. The solution is to classify `#` as a symbol. Doing so causes the following tokenization to occur:

```
cts:word("#"),cts:word("NASA")
```

Now, searching for `#NASA` matches adjacent occurrences of `#` and `NASA`, as in a topic, and searching for just `NASA` matches both topics and naked terms. Users (`@NASA`) continue to be excluded because of the tokenizer override for `@`.

29.2 User-Defined Lexer Plugins

You can use a user-defined lexer plug-in to affect how MarkLogic splits the text in document content and queries into parts. Create a user-defined lexer plugin in C++ by implementing a subclass of the `marklogic::LexerUDF` class and deploying it to MarkLogic as a native plugin. The `LexerUDF` class is a UDF (User Defined Function) interface.

MarkLogic also provides several built-in lexer plug-ins you can use to customize tokenization. For details, see “Customization Using a Built-In Lexer or Stemmer” on page 766.

This section covers the following topics:

- When to Consider a User-Defined Lexer
- LexerUDF Interface Summary
- Understanding User-Defined Lexer Control Flow
- Implementation Guidelines for User-Defined Lexers
- Creating and Deploying a User-Defined Lexer Plugin
- Registering a Custom Tokenizer with MarkLogic
- Testing a User-Defined Lexer
- Error Handling and Logging

29.2.1 When to Consider a User-Defined Lexer

MarkLogic provides several built-in lexers that you can configure for a language if you are not satisfied with the default lexer. The following are some use cases in which you might consider implementing a your own lexer:

- You need to tokenize a language that is not directly supported by MarkLogic.
- You want to use a specific 3rd party library for tokenization for a given language.
- Your data requires universal reassignment of certain characters to different tokenization classes. Tokenizer overrides, which fulfill a similar need, only apply within specific fields and come at the cost of retokenization.
- You require special format tokenization in the context of specific data fields where the requirements are more complicated than the simple reclassification provided by tokenizer overrides.

In some cases, you might also need a user-defined stemmer or custom dictionary. For example, if you're tokenizing a language not supported by MarkLogic and you wish to use stemmed searches on that language, then you would also deploy a custom stemmer. For details, see "Using a User-Defined Stemmer Plugin" on page 656 and "Custom Dictionaries for Tokenizing and Stemming" on page 665.

29.2.2 LexerUDF Interface Summary

You implement a user-defined lexer as a subclass of the `MarkLogic::LexerUDF` base C++ class. This class is defined in *`MARKLOGIC_INSTALL_DIR`*`/include/MarkLogic.h`. You can find detailed documentation about the class in the User-Defined Function API reference and in `MarkLogic.h`. You can find an example implementation in *`MARKLOGIC_INSTALL_DIR`*`/Samples/NativePlugins`.

The following table contains a brief summary of the key methods of `LexerUDF`. For a discussions of how MarkLogic uses these methods, see "Understanding User-Defined Lexer Control Flow" on page 797.

LexerUDF Method	Description
`initialize`	Initialize a `LexerUDF` after construction. This method is only called once per lexer object.
`reset`	Prepare the lexer to process a new text run. The first token should be available to the `token` method after calling this method.
`next`	Advance the lexer to the next token. Returns false if there are no more tokens.

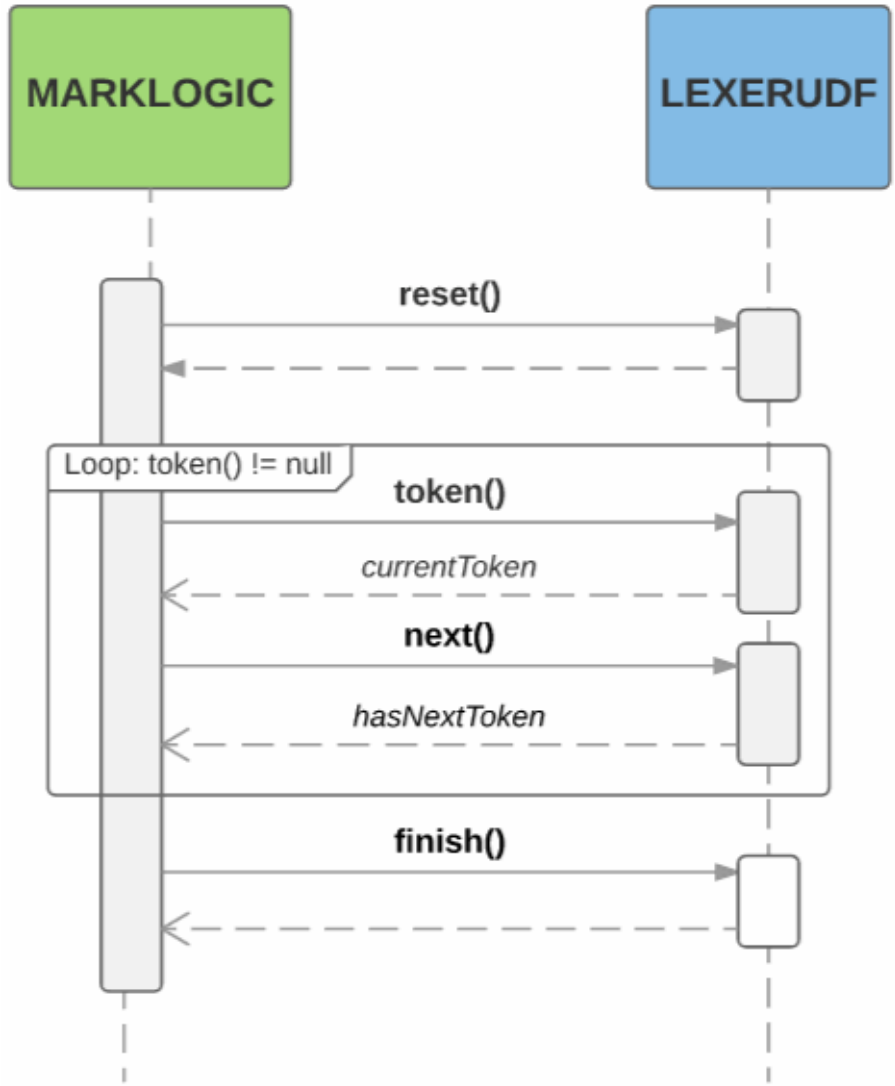
LexerUDF Method	Description
`token`	Return the current token. Returns null if there is no current token.
`finish`	Clean up from the current tokenization run.

29.2.3 Understanding User-Defined Lexer Control Flow

A user-defined lexer is implemented as a subclass of `marklogic::LexerUDF`. Once your lexer is installed as a native plugin and associated with a language, it will be applied automatically when loading content or processing query text in the context of the configured language.

MarkLogic maintains a per-language pool of lexer objects. When MarkLogic needs one of your `LexerUDF` objects, it first checks to see if one is available from the pool. If not, MarkLogic creates one using the object factor created during plugin registration. MarkLogic then calls the `initialize` method of the object.

The following diagram illustrates how MarkLogic interacts with a `LexerUDF` object during tokenization:



The lexer owns the memory allocated for the `Token` returned by the `token` method, and is responsible for releasing it when appropriate, such as when the `finish` method is called.

If it is necessary to re-tokenize a text run, MarkLogic invokes the lexer's `reset` method again.

When a tokenization run finishes, MarkLogic returns the lexer object to the pool. A lexer stays in the pool until it becomes stale. MarkLogic can choose to mark a lexer stale, or a lexer can flag itself as stale by returning `true` from its `isStale` method.

When a lexer is no longer needed, MarkLogic calls its `close` method. This enables the lexer to deallocate memory and release other resources, as needed.

29.2.4 Implementation Guidelines for User-Defined Lexers

When implementing a `LexerUDF` subclass, keep the following guidelines in mind.

> **Note:** Your UDF implementation runs in the same memory and process space as MarkLogic Server, so errors in your implementation can crash MarkLogic Server. Before deploying a user-defined lexer, you should read and understand Using Native Plugins in the *Application Developer's Guide*. See also "Testing a User-Defined Lexer" on page 801.

- You must implement a subclass of `LexerUDF` for each tokenization algorithm you want to use.
- Your lexer should partition its input into words, punctuation, and whitespace tokens with no gaps, overlaps, or reordering because MarkLogic stores the sequence of tokens, not the original input string.
- Tokenization is a low-level, inner-loop operation that MarkLogic performs during indexing (including document ingestion) and query evaluation. Your stemmer should introduce as little overhead as possible.
- Your lexer is not responsible for applying tokenizer overrides. If tokenizer overrides are configured, MarkLogic will apply them to the tokens returned by the lexer.
- Your implementation does not have to be thread safe. MarkLogic will instantiate a new lexer in each thread in which it wants to perform tokenization.
- You can include a Part of Speech indicator as part of `Token` object returned by `LexerUDF::token`. This indicator can sometimes improve the precision of the stemmer. The default stemmers only use this information for Japanese. When deciding whether and which part of speech to use, keep the following mind:
    - Including a Part of Speech is only useful when you use both a custom lexer and a custom stemmer that acts on the PoS. The default stemmer only uses PoS for Japanese.
    - You should usually return `UNSPECIFIED_POS` for tokens in short strings such as query text because there is not enough context to make a reliable classification. If you use a custom stemmer, MarkLogic recommends your stemmer return all possible stems for `UNSPECIFIED_POS`.
    - If you are sure of the POS classification, tagging a token with a specific part of speech can improve the precision of the stemmer or serve as a signal for stem ordering.
    - `UNSPECIFIED_POS` is more efficient in time and space for the stemmer. If you don't really need the extra precision or your stemmer plugin does not use POS, you should not use specific parts of speech. Using specific parts of speech makes the stemmed searches indexes bigger and adds to processing time.

- A single text run might need to be tokenized more than once. Your lexer should be idempotent across calls to its `restart` method.
- Report errors using the `Reporter` object that is passed to most `LexerUDF` methods, rather than by throwing exceptions. For details, see "Error Handling and Logging" on page 801.
- You might also want to support your language with a custom stemmer and/or a custom dictionary. To learn more about customization options, see "Stemming and Tokenization Customization" on page 762.

29.2.5 Creating and Deploying a User-Defined Lexer Plugin

Follow the steps below to create and deploy a lexer UDF in MarkLogic as a native plugin. A complete example is available in *MARKLOGIC_DIR*`/Samples/NativePlugins`.

1. Implement a subclass of the C++ class `marklogic::LexerUDF`. See *MARKLOGIC_DIR*`/include/MarkLogic.h` for interface details.

2. Implement an extern "C" function called `marklogicPlugin` to perform plugin registration. For details, see Registering a Native Plugin at Runtime in the *Application Developer's Guide*.

3. Build a dynamically linked library containing your UDF and registrationf unction. You should use the Makefile in *MARKLOGIC_DIR*`/Samples/NativePlugins` as the basis for building your plugin. For more details, see Building a Native Plugin Library in the *Application Developer's Guide*.

4. Following the directions in Using Native Plugins to package and install your plugin. See the note below about dependent libraries.

5. Configure your lexer as the lexer plugin for at least one language. For details, see "Configuring Tokenization and Stemming Plugins" on page 764.

The native plugin interface includes support for bundling dependent libraries in the native plugin zip file. However, many 3rd party natural language processing tools are large, complex, and have strict installation directory requirements. If you are using such a packge, you should install the 3rd party package package independently on each host in the cluster, rather than trying to include it inside your native plugin package.

29.2.6 Registering a Custom Tokenizer with MarkLogic

A native plugin becomes available for use once you install it, but it will not be loaded until there is a reason to use it. A plugin containing only a lexer UDF is only loaded if it is associated with at least one language, and the need to tokenize text in that language arises.

When MarkLogic loads a native plugin, it performs a registration handshake to obtain details about the plugin such as what UDFs the plugin provides. This handshake is performed through an `extern "C"` function named `marklogicPlugin` that must be part of every native plugin.

The following code is an example of a registration function for a plugin that registers only a single lexer capability. Assume the plugin implements a `LexerUDF` subclass named `MyLexerUDF`. The code registers the lexer with the plugin id "sample_lexer".

```
extern "C" PLUGIN_DLL void
marklogicPlugin(Registry& r)
{
  r.version();
  r.registerLexer<MyLexerUDF>("sample_lexer");
}
```

For details, see Registering a Native Plugin at Runtime in the *Application Developer's Guide*. For a complete example, see the code in *MARLOGIC_DIR*`/Samples/NativePlugins`.

29.2.7 Testing a User-Defined Lexer

You can test your `LexerUDF` implementation in the following ways:

- Create standalone test scaffolding.
- Use the `cts:tokenize` XQuery function or the `cts.tokenize` Server-Side JavaScript function to exercise your plugin after it is installed and configured for at least one language.

Testing your lexer standalone during development is highly recommended. It is much easier to debug your code in this setup. Also, since it is possible for native plugin code to crash MarkLogic, it is best to test and stabilize your code outside the server environment.

You can find example test scaffolding in *MARKLOGIC_DIR*`/Samples/NativePlugins/TestStemTok.cpp`. See the `main()` function for a starting point.

29.2.8 Error Handling and Logging

Use `marklogic::Reporter` to log messages and notify MarkLogic Server of fatal errors. Your code should not report errors to MarkLogic Server by throwing exceptions.

Report non-fatal errors and other messages using `marklogic::Reporter::log`. This method logs a message to the MarkLogic Server error log and returns control to your code. Most methods of `LexerUDF` accept a `marklogic::Reporter` input parameter.

Report **fatal** errors using `marklogic::Reporter::error`. You should reserve calls to `Reporter::error` for serious errors from which no recovery is possible. Reporting an error via `Reporter::error` has the following effects:

- If you report a fatal tokenization error during document insertion, the insertion transaction aborts.

- If you report a fatal tokenization error during reindexing, reindexing of the document fails.
- Control does not return to your code. Tokenization stops.
- MarkLogic Server returns `XDMP-UDFERR` to the application. Your error message is included in the `XDMP-UDFERR` error.

The following snippet reports an error and aborts tokenization:

```
#include "MarkLogic.h"
using namespace marklogic;
...
void ExampleUDF::next(Reporter& r)
{
  ...
  r.log(Reporter::Error, "Bad codepoint.");
}
```

For more details, see the `marklogic::Reporter` class in `MARKLOGIC_DIR/include/MarkLogic.h`.

30.0 Encodings and Collations

In addition to the language support described in "Language Support in MarkLogic Server" on page 751, 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

30.1 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. If the content cannot be translated, MarkLogic Server throws an exception indicating that there is non-UTF-8 content.

You can specify an explicit encoding in the following ways:

- If your content is ingested on behalf of an HTTP request, you can specify an encoding in the HTTP headers, such as setting the `charset` parameter of the Content-type header.
- Set the `encoding` option of the functions listed in the following table.

XQuery	JavaScript
`xdmp:document-load`	`xdmp.documentLoad`
`xdmp:document-get`	`xdmp.documentGet`
`xdmp:zip-get`	`xdmp.zipGet`
`xdmp:gunzip`	`xdmp.gunzip`
`xdmp:xslt-invoke`	`xdmp.xsltInvoke`

Encoding is determined using the following precedence, from highest to lowest:

- The encoding option of the ingestion function, if set.
- The encoding specified by the HTTP headers, if present.
- Otherwise, assume UTF-8.

If you set the `encoding` option to "auto", then MarkLogic tries to determine the encoding from the document content.

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.

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 content encoding, the result can be unpredictable: You might get an exception in some situations, but you might end up with the wrong characters in other situations.

For details on the syntax of the `encoding` option, see the *MarkLogic XQuery and XSLT Function Reference*.

30.2 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

30.2.1 Overview of Collations

> **Note:** Javascript does not have the concept of a prolog; therefore, there is no way to declare a default collation in Javascript the way it is done in XQuery.

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.

30.2.2 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 809), and is the system default. The second is the codepoint order collation, and was the default in pre-3.2 releases of MarkLogic Server.

30.2.3 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

30.2.3.1 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 809).

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 811.

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`

30.2.3.2 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.

Attribute	Legal Values	Descriptions
`<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.
`<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 809) 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.

30.2.4 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).

30.2.5 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/.

30.2.6 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.

30.2.7 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`).

30.3 Collations and Character Sets By Language

The following table lists the languages for which MarkLogic Server supports language-specific tokenization and stemming. It also lists some common collations and character sets for each language.

Note that some of the listed character set names can be ambiguous. MarkLogic uses the International Components for Unicode (ICU) library for character encoding and conversion. For best accuracy, refer to the ICU converter alias mapping at http://demo.icu-project.org/icu-bin/convexp.

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/EO	case sensitive	

Language	Base Collations		Character Sets
Norwegian (Nynorsk and Bokmål)	http://marklogic.com/collation/nn (Nynorsk)	case/diacritic sensitive	ISO-8859-1 cp1252
	http://marklogic.com/collation/nn/S1	case/diacritic insensitive	
	http://marklogic.com/collation/nn/S2	diacritic sensitive	
	http://marklogic.com/collation/nn/S1/EO	case sensitive	
	http://marklogic.com/collation/nb (Bokmål)	case/diacritic sensitive	
	http://marklogic.com/collation/nb/S1	case/diacritic insensitive	
	http://marklogic.com/collation/nb/S2	diacritic sensitive	
	http://marklogic.com/collation/nb/S1/EO	case sensitive	
Swedish	http://marklogic.com/collation/sv	case/diacritic sensitive	ISO-8859-1 cp1252
	http://marklogic.com/collation/sv/S1	case/diacritic insensitive	
	http://marklogic.com/collation/sv/S2	diacritic sensitive	
	http://marklogic.com/collation/sv/S1/EO	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 762. The language-specific collations are available to all languages, regardless of what languages are enabled in the license key.

31.0 Appendix: Query Options Reference

This appendix is a reference guide to the query options used for search and lexicon analysis by the XQuery Search API and the MarkLogic Client APIs (REST, Java, Node.js). This appendix contains the following topics:

- How to Use This Reference
- Options Summary
- additional-query
- concurrency-level
- constraint
- debug
- default-suggestion-source
- extract-document-data
- forest
- fragment-scope
- grammar
- operator
- page-length
- quality-weight
- result-decorator
- return-aggregates
- return-constraints
- return-facets
- return-frequencies
- return-metrics
- return-plan
- return-qtext
- return-query
- return-results
- return-similar
- return-values
- search-option

- searchable-expression
- sort-order
- suggestion-source
- term
- transform-results
- tuples
- values
- Term Options
- Facet Options
- Range Options
- Geospatial Point Query Options
- Geospatial Region Query Options
- Suggestion Options
- Values Options

31.1 How to Use This Reference

This reference describes the layout of the query options structure usable with search and lexicon query interfaces of the XQuery Search API (`search:search`, `search:values`, etc.) and the Client APIs (REST, Java, Node.js). Both XML and JSON representations are shown, but not all APIs support both representations; consult the reference for the API you're using.

Some of the APIs provide builders for constructing query options, so you do not need to know the syntax. However, this appendix can still be useful with a builder because it provides details on the meaning, defaults, and limits for specific option components.

The Syntax Summary for each option shows all possible components, but not all are required and some cannot be used together. Refer to the Component Description section for details on a given option.

Option components that can be specified as repeating XML elements are usually represented by array values in JSON. In many cases, you can omit the array wrapper if there is only one element.

31.2 Options Summary

A single options node or object can contain can contain options useful for document searches, lexicon and index queries, and/or search term completion suggestions. However, not all options are used by all operations. For example, the `values` and `tuples` options only apply to lexicon query operations (`search:values`), `default-suggestion-source` only applies to search term completion suggestion operations (`search:suggest`), and `extract-document-data` only applies to document searches (`search:search`).

A set of query options has the following structure. You can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<options>`
`  anyOption...`
`</options>`
	`{"options": {`
	`  anyOption, ...`
	`}`

Where *anyOption* is zero or more of the child components summarized in the following table. For more details about a given option, see the option-specific topic.

Key	Description
additional-query	Additional serialized cts:query's, as XML literals. This option has array value in JSON and can appear multiple times in XML.
concurrency-level	The maximum number of threads used to resolve facets.
constraint	Zero or more constraints that limit the scope of a search and/or define facets which can be returned as part of the search results. This option has array value in JSON and can appear multiple times in XML.
debug	Whether or not to enable debugging mode. The default is `false`.
default-suggestion-source	Defines the content to be used as the default source of suggestions (see `search:suggest`).
extract-document-data	Specify element, attribute, or JSON property content from the search matches to return.
forest	One or more forest IDs. This option has array value in JSON and can appear multiple times in XML.

Key	Description
fragment-scope	Controls the global fragment scope over which to search.
grammar	A custom search grammar definition.
operator	A list of state elements, each representing a unique run-time configuration option. This option has array value in JSON and can appear multiple times in XML.
page-length	The number of results to return per page. The default value is 10.
quality-weight	Specifies a weighting factor to use in the query. The default value is 1.0.
return-aggregates	Include the result of running a builtin or user-defined aggregate function. Applies only to queries against values or tuples.
return-constraints	Include the input constraint definitions in the results. The default is false.
return-facets	Include resolved facets in the results. The default is true.
return-frequencies	Include frequencies in the results. The default is true.
return-metrics	Include performance statistics in the results. The default is true.
return-plan	Include `xdmp:plan` output in the results. The default is false.
return-qtext	Include the original query text in the results. The default is true.
return-query	Include the XML query representation in the results. The default is false.
return-results	Include search results in the output. The default is true.
return-similar	Include with each search result a list of URLs of similar documents in the database. The default is false.
return-values	When querying a range index or values lexicon, whether or not to include the index/lexicon values in the results. Default: true.
search-option	For advanced users, one or more options to pass to the underlying query operation. This option has array value in JSON and can appear multiple times in XML.
searchable-expression	An XPath expression to be searched. Whatever expression is specified is returned from the search.

Key	Description
sort-order	Set the default sort order. The first such value is the primary sort order, the second is secondary sort order, and so on. This option has array value in JSON and can appear multiple times in XML.
suggestion-source	Specify a search term completion suggestion source.
transform-results	Specify a function to use to process a search result for the snippet output.
tuples	Define one or more value lexicons query against, matching value co-occurrences. That is, tuples of values, each of which appear in the same fragment.
values	Define one or more value lexicons to query against.

31.3 additional-query

Use this option to specify one or more additional cts queries to apply when searching documents and generating suggestions. The queries are AND'd with the input query. The query results are constrained by the `additional-query`(s).

Specify each additional query value as the serialized XML representation of a `cts:query`.

- Syntax Summary
- Component Description
- Examples
- See Also

31.3.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<additional-query>`
`  serialized-cts-query`
`</additional-query>`
	`"additional-query": [`
	`  "serialized-cts-query"`
	`]`

31.3.2 Component Description

The text value that represents each additional query is a serialized `cts:query`. In XML options, it is represented as XML. In JSON options, it is a string containing the serialized XML.

If your query options include multiple additional queries, they are AND'd together, as if with `cts:and-query`.

31.3.3 Examples

The following example constrains the results to the directory named `/my/directory/`. Whitespace and linebreaks are added for readability.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <additional-query>`
	`  <cts:directory-query xmlns:cts="http://marklogic.com/cts">`
	`  <cts:uri>/my/directory/</cts:uri>`
	`  </cts:directory-query>`
	`  </additional-query>`
	`</options>`

JSON	`{"options": {`
	`  "additional-query":[`
	`  "<directory-query xmlns='http://marklogic.com/cts'>`
	`  <uri>/my/directory/</uri>`
	`  </directory-query>"`
	`  ]`
	`}}`

31.3.4 See Also

For more details, see the following topics:

- "Serializations of cts:query Constructors" on page 284
- "Understanding cts:query" on page 248
- The built-in functions in the `cts` namespace in the *XQuery and XSLT Reference Guide*

31.4 concurrency-level

The maximum number of threads to use when resolving facets. The default is 8, which specifies that at most 8 threads will be used concurrently to resolve facets. The value of this option should be an integer value greater than 0. The default value is 8.

The following example specifies a concurrency level of 16.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <concurrency-level>16</concurrency-level>`
	`</options>`

JSON	`{"options": {`
	`  "concurrency-level": 16`
	`}}`

31.5 constraint

This option forms the outer container for a constraint definition. Use constraints to limit the scope of a search and/or define facets that can be returned as part of the search results. Constraints can also be applicable when generating search suggestions. No constraints are defined by default.

Each constraint must include a name that is unique among the options in scope for a search. The constraint name can be used as a term qualifier in a string query; for details, see "Searching Using String Queries" on page 67. The name can also be used to identify the constraint in some kinds of structured queries and QBE's.

This section includes the following high level topics about the constraint option:

- Syntax Summary
- Component Description
- Examples
- See Also

This section also includes detailed descriptions of each of the following constraint types and the more complex components of range and geospatial constraints, such as bucket, computed-bucket, and heatmap.

- range
- value
- word
- collection
- container
- element-query
- properties

- geo-attr-pair
- geo-elem
- geo-elem-pair
- geo-json-property
- geo-json-property-pair
- geo-path
- geo-region-path
- custom
- path-index

31.5.1 Syntax Summary

This option has the following structure. Each constraint must include a name and exactly one constraint specification. An options node can define multiple constraints. In JSON, the `constraint` array can be empty. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<constraint name="`*uniqueName*`">`
`  <annotation/>`
`  `*anyConstraintType*
`</constraint>`
	`"constraint": [`
	`  {"name": "`*uniqueName*`",`
	`  "annotation": `*string*`,`
	`  `*anyConstraintType*
	`  }`
	`]`

Where *anyConstraintType* is one of the following constraint specifications:

- range
- value
- word
- collection
- container
- element-query
- properties
- geo-attr-pair
- geo-elem

- geo-elem-pair
- geo-json-property
- geo-json-property-pair
- geo-path
- custom

31.5.2 Component Description

The components of this option have the following semantics. A constraint can contain the following child elements or properties.

Element, Attribute or Property Name	Description
`name`	Required. An identifier for the constraint. The name must be unique within the in-scope query options and may not contain whitespace.
`annotation`	Your comments. Annotations have no effect on a query.
`anyConstraintType`	A constraint specification. For a list of constraint types, see “Syntax Summary” on page 823.

31.5.3 Examples

The following example defines two constraints: a container constraint that limits matches to those that occur within an XML element named “TITLE”, and a value constraint that limits matches to the values in an XML element named “COST”. For more examples, refer to the individual constraint types.

Format	Example
XML	<options xmlns="http://marklogic.com/appservices/search"> <constraint name="title"> <container> <element ns="" name="TITLE" /> </container> </constraint> <constraint name="cost"> <value> <element ns="" name="COST" /> </value> </constraint> </options>

JSON	{"options": {
	  "constraint": [
	  { "name": "title",
	  "container": {
	  "element": {
	  "ns": "",
	  "name": "TITLE"
	  }}},
	  { "name": "cost",
	  "value": {
	  "element": {
	  "ns": "",
	  "name": "COST"
	  }}}
	  ]
	}}

31.5.4 See Also

For more details, see the following topics and the See Also topics for individual constraint types.

- “Constraint Options” on page 382

31.5.5 range

A component of a constraint option that specifies an XML element, XML element attribute, field, JSON property, or XPath expression on which to constrain by range, as in a range query.

There must be a range index of the specified type (and collation for string range indexes) defined for the specified element, attribute, JSON property, field, or path.

For best performance of range constraints that include `bucket` or `computed-bucket` specifications, define the buckets in a consistent, sorted order (ascending or descending). If the order is not consistent, then the bucketed results are returned in the order specified, regardless of any sorting `facet-option` in the specification.

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.5.1 Syntax Summary

This option has the following structure. The definition must include exactly one of `element`, `field`, `json-property`, or `path-index` descriptor. If there is an `element`, an `attribute` descriptor may also be included. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<range type="indexedType"`
`collation="collURI"`
`facet="boolean"`
`nullable="boolean">`
`<element ns="namespace" name="name"/>`
`<attribute ns="namespace" name="name"/>`
`<field name="name"/>`
`<json-property>name</json-property>`
`<path-index/>`
`<fragment-scope>scope</fragment-scope>`
`<bucket/>`
`<computed-bucket/>`
`<facet-option>option</facet-option>`
`<range-option>option</range-option>`
`<weight>double</weight>`
`</range>`
	`"range": {`
	`"type": string,`
	`"collation" : string,`
	`"facet": boolean,`
	
	`"element": {`
	`"ns": "namespace",`
	`"name": "name"`
	`},`
	`"attribute": {`
	`"ns": "namespace",`
	`"name": "name"`
	`},`
	`"field": {"name": "name"},`
	`"json-property": "name",`
	`"path-index": [path index desc],`
	`"fragment-scope": "scope",`
	`"bucket": [bucket descriptor],`
	`"computed-bucket": [bucket desc],`
	`"facet-option" : [ "option" ],`
	`"range-option": [ "option" ],`
	`"nullable" : boolean,`
	`"weight": double`
	`}`

31.5.5.2 Component Description

The components of this option have the following semantics. The definition must include exactly one of `element`, `json-property`, `field`, or `path-index`. All other components are optional.

Element, Attribute or Property Name	Description
`type`	The type of the values in the associated index. The database configuration must include a range index with this type. Use collation to further disambiguate range indexes of type `xs:string`.
`collation`	A collation URI. If this value is present, the database configuration should include a range index of type `xs:string` that uses this collation. If not specified, the default Root Collation is assumed. For details, see "Collations" on page 804.
`facet`	Whether or not to include facets based on this constraint in the query results.
`nullable`	Whether or not to allow null values in tuples when making a values query.
`element`	Defines an XML element to constrain by. If there is an `element`, there can also be an `attribute`. Specify the element local name in `name`. If the element is in a namespace, specify the namespace URI in `ns`. The database configuration must include a corresponding element or element attribute range index.
`attribute`	Defines an XML element attribute to constrain by. There must be an accompanying `element` descriptor. Specify the attribute local name in `name`. If the attribute is in a namespace, specify the namespace URI in `ns`. The database configuration must include a corresponding element attribute range index.
`json-property`	Defines a JSON property to constrain by. The database configuration must include a corresponding element range index. (JSON properties are indexed using element range indexes.)
`field`	Defines a field to constrain by. The database configuration must include a corresponding field range index.

Element, Attribute or Property Name	Description
`path-index`	Defines a path range index reference to constrain by. For details, see "path-index" on page 888. In XML, you can specify multiple indexes by including this component multiple times. In JSON, you can specify multiple indexes by setting the property value to an array of path index descriptors. The database configuration must include a corresponding path range index.
`fragment-scope`	Set a local fragment scope for this constraint. The local fragment scope overrides the global fragment scope. For example, a `fragment-scope` of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents. Allowed values: `properties`, `documents` (default).
`bucket`	Zero or more named ranges of static values. For details, see "bucket" on page 880.
`computed-bucket`	Zero or more named ranges of dynamic values. For details, see "computed-bucket" on page 884.
`facet-option`	Specify faceting options to apply when generating facets. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<facet-option>limit=5</facet-option>` in XML, or `"facet-option": ["limit=5"]` in JSON. For details, see "Facet Options" on page 950.
`range-option`	Specify range options that influence the interpretation of the constraint. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<range-option>min-occurs=2</range-option>` in XML, or `"range-option": ["min-occurs=2"]` in JSON. For details, see "Range Options" on page 951.
`weight`	Higher weights move search results up in the relevance order. The default is 1.0. The weight should be less than or equal to 64 and greater than or equal to -16 (between -16 and 64); weights greater than 64 will have the same effect as a weight of 64. 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.

31.5.5.3 Examples

This section contains an example contains one of each type of range constraint and an example that demonstrates the use of faceting options. For more examples, see “bucket” on page 880 and “computed-bucket” on page 884.

The following example includes one of each basic type of range constraint: element, element attribute, JSON property, field, and path index. The database configuration must include a range index that corresponds to each constraint.

Format	Example

XML	<options xmlns="http://marklogic.com/appservices/search">
	  <constraint name="elem">
	  <range type="xs:string">
	  <element ns="myNsURI" name="anElemName"/>
	  </range>
	  </constraint>
	  <constraint name="attr">
	  <range type="xs:string">
	  <element ns="myNsURI" name="anElemName"/>
	  <attribute ns="myNsURI" name="anAttrName" />
	  </range>
	  </constraint>
	  <constraint name="json-prop">
	  <range type="xs:string">
	  <json-property>aPropName</json-property>
	  </range>
	  </constraint>
	  <constraint name="field">
	  <range type="xs:string"
	  collation="http://marklogic.com/collation/codepoint">
	  <field name="aFieldName"/>
	  </range>
	  </constraint>
	  <constraint name="path">
	  <range type="xs:gYear" facet="true">
	  <path-index xmlns:my="http://example.com">
	  /publication/my:meta/my:year
	  </path-index>
	  </range>
	  </constraint>
	</options>

Format	Example

JSON	{"options": {
	  "constraint": [
	  { "name": "elem",
	  "range": {
	  "type": "xs:string",
	  "element": { "ns": "myNsURI", "name": "anElemName" }
	  }},
	  { "name": "attr",
	  "range": {
	  "type": "xs:string",
	  "element": { "ns": "myNsURI", "name": "anElemName" },
	  "attribute": { "ns": "myNsURI", "name": "anAttrName" }
	  }},
	  { "name": "json-prop",
	  "range": {
	  "type": "xs:string",
	  "json-property": { "name": "aPropName" }
	  }},
	  { "name": "field",
	  "range": {
	  "type": "xs:string",
	  "collation": "http://marklogic.com/collation/codepoint",
	  "field": { "name": "aFieldName" }
	  }},
	  { "name": "path",
	  "range": {
	  "type": "xs:gYear",
	  "facet": true,
	  "path-index": {
	  "namespaces": {"my": "http://example.com"},
	  "text": "/publication/my:meta/my:year"
	  }
	  }}
	  ]
	}}

The following example defines a constraint that is used to generate facets. The `facet-option` values are passed to the underlying lexicon calls.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="color">`
	`  <range type="xs:string" facet="true">`
	`  <element ns="myNsURI" name="bodycolor"/>`
	`  <facet-option>frequency-order</facet-option>`
	`  <facet-option>descending</facet-option>`
	`  </range>`
	`  </constraint>`
	`  <constraint name="owner">`
	`</options>`

JSON	`{"options": {`
	`  "constraint": [`
	`  { "name": "color",`
	`  "range": {`
	`  "type": "xs:string",`
	`  "facet": true,`
	`  "element": { "ns": "myNsURI", "name": "bodycolor" },`
	`  "facet-option": [ "frequence-order", "descending" ]`
	`  }}]`
	`}}`

31.5.5.4 See Also

For more details on using this option, see the following topics:

- “Constrained Searches and Faceted Navigation” on page 34
- “Constraint Options” on page 382
- “Using Relational Operators on Constraints” on page 72
- “Support for Multiple Query Types” on page 26

31.5.6 value

A component of a constraint option that specifies a XML element, XML attribute, JSON property, or field on which to constrain, as in a value query. For JSON property constraints, you can optionally specify a node using `value/@type`. If present, type must be one of `string` (default), `number`, `boolean`, or `null`. You cannot create facets from a value constraint.

- Syntax Summary
- Component Description
- Examples

- See Also

31.5.6.1 Syntax Summary

A value constraint definition has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<value type="jsonNodeType">`
`  <element ns="namespace" name="name"/>`
`  <attribute ns="namespace" name="name"/>`
`  <field name="fieldName"/>`
`  <json-property>name</json-property>`
`  <fragment-scope>scope</fragment-scope>`
`  <weight>double</weight>`
`  <term-option>option</term-option>`
`</value>`
	`"value": {`
	`  "type": "jsonNodeType"`
	`  "element": {`
	`  "ns": "namespace",`
	`  "name": "name"`
	`  },`
	`  "attribute": {`
	`  "ns": "namespace",`
	`  "name": "name"`
	`  },`
	`  "json-property": "name",`
	`  "field": {"name": "fieldName"},`
	`  "weight": double,`
	`  "fragment-scope": "scope",`
	`  "term-option": [ "option" ]`
	`}`

31.5.6.2 Component Description

The components of this option have the following semantics. The definition must include exactly one of `element`, `json-property`, or `field`. If there is an `element`, there can also be an `attribute`.

Element, Attribute or Property Name	Description
`type`	A JSON node type, one of `string` (default), `boolean`, `null`, `number`. Only meaningful for JSON content. Use `type` to constrain the matches to values in this node type. Non-JSON documents never contain nodes of these types. Optional.
`element`	Defines an XML element on which to constrain queries If there is an `element`, there can also be an `attribute`. Specify the element local name in `name`. If the element is in a namespace, specify the namespace URI in `ns`.
`attribute`	Defines an XML element attribute on which to constrain queries. There must be an accompanying `element` descriptor. Specify the attribute local name in `name`. If the attribute is in a namespace, specify the namespace URI in `ns`.
`json-property`	Defines a JSON property on which to constrain queries.
`field`	Defines a field on which to constrain queries.
`weight`	Higher weights move search results up in the relevance order. The default is 1.0. The weight should be less than or equal to 64 and greater than or equal to -16 (between -16 and 64); weights greater than 64 will have the same effect as a weight of 64. 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.
`fragment-scope`	Set a local fragment scope for this constraint. The local fragment scope overrides the global fragment scope. For example, a `fragment-scope` of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents. Allowed values: `properties`, `documents` (default).
`term-option`	Specify options to apply when generating facets. In XML, specify multiple options by including the element multiple times. If the option has a value, the element or array item value is of the form *`option=value`*. For example: `<term-option>lang=en</term-option>` in XML, or `"term-option": ["lang=en"]` in JSON. “Term Options” on page 950.

31.5.6.3 Examples

The following example includes a value constraint of each type: element, element attribute, JSON property, and field.

Format	Example
XML	<options xmlns="http://marklogic.com/appservices/search"> <constraint name="my-elem-value"> <value> <element ns="my-namespace" name="my-localname"/> </value> </constraint> <constraint name="my-attr-value"> <value> <attribute ns="" name="my-attribute"/> <element ns="my-namespace" name="my-localname"/> </value> </constraint> <constraint name="my-json-value"> <value type="number"> <json-property>myNumericPropName</json-property> </value> </constraint> <constraint name="my-field-value"> <value> <field name="fieldvalue"/> <weight>2.0</weight> </value> </constraint> </options>

Format	Example

JSON	<pre>{"options": {
	  "constraint": [
	  { "name": "my-elem-value",
	  "value": {
	  "element": {"ns": "my-namespace","name": "my-localname"}
	  }},
	  { "name": "my-attr-value",
	  "value": {
	  "element": {"ns": "my-namespace","name": "my-localname"},
	  "attribute": {"ns": "","name": "my-attribute"}
	  }},
	  { "name": "my-json-value",
	  "value": {
	  "type": "number",
	  "json-property": "myNumericPropName"
	  }},
	  { "name": "my-field-value",
	  "value": {
	  "field": {"name": "fieldName"},
	  "weight": 2.0
	  }}
	  ]
	}}</pre>

31.5.6.4 See Also

For more details on using this option, see the following topics:

- “Constraint Options” on page 382
- “Value Constraint Example” on page 389
- “Support for Multiple Query Types” on page 26

31.5.7 word

A component of a constraint option that specifies the XML element, XML element attribute, JSON property, or field on which to constrain by word. You cannot create facets from a word constraint.

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.7.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<word>`
`  <element ns="namespace" name="name"/>`
`  <attribute ns="namespace" name="name"/>`
`  <field name="name"/>`
`  <json-property>name</json-property>`
`  <fragment-scope>scope</fragment-scope>`
`  <weight>double</weight>`
`  <term-option>option</term-option>`
`</word>`
	`"word": {`
	`  "element": {`
	`  "ns": "namespace",`
	`  "name": "name"`
	`  },`
	`  "attribute": {`
	`  "ns": "namespace",`
	`  "name": "name"`
	`  },`
	`  "json-property": "name",`
	`  "field": {"name": "name"},`
	`  "weight": double,`
	`  "fragment-scope": "scope",`
	`  "term-option": [ "option" ]`
	`}`

31.5.7.2 Component Description

The child components of this option have the following semantics:

Element, Attribute or Property Name	Description
`element`	Defines an XML element on which to constrain queries If there is an `element`, there can also be an `attribute`. Specify the element local name in `name`. If the element is in a namespace, specify the namespace URI in `ns`.
`attribute`	Defines an XML element attribute on which to constrain queries. There must be an accompanying `element` descriptor. Specify the attribute local name in `name`. If the attribute is in a namespace, specify the namespace URI in `ns`.
`json-property`	Defines a JSON property on which to constrain queries.

Element, Attribute or Property Name	Description
`field`	Defines a field on which to constrain queries.
`weight`	Higher weights move search results up in the relevance order. The default is 1.0. The weight should be less than or equal to 64 and greater than or equal to -16 (between -16 and 64); weights greater than 64 will have the same effect as a weight of 64. 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.
`fragment-scope`	Set a local fragment scope for this constraint. The local fragment scope overrides the global fragment scope. For example, a `fragment-scope` of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents. Allowed values: `properties`, `documents` (default).
`term-option`	Specify options to apply when generating facets. In XML, specify multiple options by including the element multiple times. If the option has a value, the element or array item value is of the form *`option=value`*. For example: `<term-option>lang=en</term-option>` in XML, or `"term-option": ["lang=en"]` in JSON. For details, see “Term Options” on page 950.

31.5.7.3 Examples

The following example includes a word constraint of each possible type (element, element attribute, JSON property, and field). The JSON property constraint demonstrates the use of term options. The field constraint demonstrates the use of a custom weight.

Format	Example
XML	<options xmlns="http://marklogic.com/appservices/search"> <constraint name="my-elem-value"> <word> <element ns="my-namespace" name="my-localname"/> </word> </constraint> <constraint name="my-attr-value"> <word> <attribute ns="" name="my-attribute"/> <element ns="my-namespace" name="my-localname"/> </word> </constraint> <constraint name="my-json-value"> <word> <json-property>myPropName</json-property> <term-option>case-sensitive</term-option> <term-option>unstemmed</term-option> </word> </constraint> <constraint name="my-field-value"> <word> <field name="fieldvalue"/> <weight>2.0</weight> </word> </constraint> </options>

Format	Example

JSON	<pre>{"options": {
	  "constraint": [
	  { "name": "my-elem-value",
	  "word": {
	  "element": {"ns": "my-namespace","name": "my-localname"}
	  }},
	  { "name": "my-attr-value",
	  "word": {
	  "element": {"ns": "my-namespace","name": "my-localname"},
	  "attribute": {"ns": "","name": "my-attribute"}
	  }},
	  { "name": "my-json-value",
	  "word": {
	  "json-property": "myPropName",
	  "term-option": ["case-sensitive", "unstemmed"]
	  }},
	  { "name": "my-field-value",
	  "value": {
	  "field": {"name": "fieldName"},
	  "weight": 2.0
	  }}
	  ]
	}}</pre>

31.5.7.4 See Also

For more details on using this option, see the following topics:

- “Constraint Options” on page 382
- “Word Constraint Examples” on page 389
- “Support for Multiple Query Types” on page 26

31.5.8 collection

A component of a constraint option that constrains results by collection, matching either documents in the specified collections or in collections with the specified collection URI prefix.

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.8.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<collection prefix="URIpfx" facet="boolean">`
`  <facet-option>`*`option`*`</facet-option>`
`</collection>`
	`"collection": {`
	`  "prefix": "`*`URIprefix`*`",`
	`  "facet-option": [ "`*`option`*`" ],`
	`  "facet": `*`boolean`*
	`}`

31.5.8.2 Component Description

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`prefix`	Constrain matches to collections with collection URIs matching this prefix. Use this component to as shorthand when constraining by collections with a shared URI prefix. For details, see “Examples” on page 841.
`facet`	Whether or not to generate facets from this constraint. Default: true.
`facet-option`	Specify options to apply when generating facets. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form `option=value`. For example: `<facet-option>limit=5</facet-option>` in XML, or `"facet-option": ["limit=5"]` in JSON. “Facet Options” on page 950.

31.5.8.3 Examples

The following example defines a constraint that limits results to matches in documents in collections with the URI prefix "/my/collection/". The constraint also indicates facets should be generated using the collection (`facet` is true). The facet option `limit` specifies that at most 5 results are returned.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="mycoll" facet="true">`
	`  <collection prefix="/my/collection/"/>`
	`  <facet-option>limit=5</facet-option>`
	`  </collection>`
	`  </constraint>`
	`</options>`

JSON	`{"options": {`
	`  "constraint": [`
	`  { "name": "mycoll",`
	`  "collection": {`
	`  "prefix": "/my/collection/",`
	`  "facet": true,`
	`  "facet-option": [ "limit=5" ]`
	`  }}`
	`  ]`
	`}}`

For example, if your database includes documents in the collections "/my/collection/animals" and "/my/collections/edibles", then the following string query text matches documents containing the term "aardvark" that are also in the collection "/my/collection/animals".

```
mycoll:animals AND aardvark
```

Omit the prefix to constrain on complete collection URIs. For example, if you omit `prefix` from the above constraint definition, then you would find the same documents with the following query text:

```
mycoll:/my/collection/animals AND aardvark
```

31.5.8.4 See Also

For more details on using this option, see the following topics:

- "Collection Constraint Example" on page 390
- "Collections" on page 693
- "Constraint Options" on page 382
- `cts:collection-query`

31.5.9 container

A component of a constraint option that specifies a constraint that restricts a search to a specified XML element or JSON property container. You cannot create facets from a container constraint.

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.9.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<container>`
`  <element ns="`*`namespace`*`" name="`*`name`*`"/>`
`  <json-property>`*`name`*`</json-property>`
`  <fragment-scope>`*`scope`*`</fragment-scope>`
`</container>`
	`"container": {`
	`  "element": {`
	`  "ns": "`*`namespace`*`",`
	`  "name": "`*`name`*`"`
	`  },`
	`  "json-property": "`*`name`*`",`
	`  "fragment-scope": "`*`scope`*`"`
	`}`

31.5.9.2 Component Description

The components of this option have the following semantics. The constraint must include exactly one of `element` or `json-property`.

Element, Attribute or Property Name	Description
`element`	Defines an XML element on which to constrain queries. Specify the element local name in `name`. If the element is in a namespace, specify the namespace URI in `ns`.
`json-property`	Defines a JSON property on which to constrain queries.
`fragment-scope`	Set a local fragment scope for this constraint to further constrain where matches occur. The local fragment scope overrides the global fragment scope. For example, a `fragment-scope` of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents. Allowed values: `properties`, `documents` (default).

31.5.9.3 Examples

The following example includes two container constraints, one on an XML element and one on a JSON property.

Format	Example
XML	<options xmlns="http://marklogic.com/appservices/search"> <constraint name="ele-container-constraint"> <container> <element name="title" ns="http://my/namespace" /> </container> </constraint> <constraint name="json-container-constraint"> <container> <json-property>author</json-property> </container> </constraint> </options>

JSON	{"options": {
	  "constraint": [
	  { "name": "ele-container-constraint",
	  "container": {
	  "element": {"ns": "http://my/namespace","name": "title"}
	  }},
	  { "name": "json-container-constraint",
	  "container": {
	  "json-property": "author",
	  }}
	  ]
	}}

31.5.9.4 See Also

For more details on using this option, see the following topics:

- “Constraint Options” on page 382
- “Support for Multiple Query Types” on page 26

31.5.10 element-query

A component of a constraint option that specifies a constraint that restricts the search to the specified element. You cannot create facets from an `element-query` constraint.

> **Note:** This option is deprecated. Use container instead.

- Syntax Summary
- Component Description

- Examples

31.5.10.1Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<element-query ns="namespace" name="name"/>`
`  <fragment-scope>scope</fragment-scope>`
`</element-query>`
	`"element-query": {`
	`  "ns": "namespace",`
	`  "name": "name",`
	`  "fragment-scope": "scope"`
	`}`

31.5.10.2Component Description

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`ns`	The namespace of the element, if it is in a namespace.
`name`	Required. The local name of the element.
`fragment-scope`	Set a local fragment scope for this constraint to further constrain where matches occur. The local fragment scope overrides the global fragment scope. For example, a `fragment-scope` of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents. Allowed values: `properties`, `documents` (default).

31.5.10.3Examples

The following example demonstrates an element-query constraint on the element “title” in the namespace `http://my/namespace`. This option is deprecated; use container instead.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="elem-constraint">`
	`  <element-query name="title" ns="http://my/namespace" />`
	`  </constraint>`
	`</options>`

JSON	`{"options": {`
	`  "constraint": [`
	`  { "name": "elem-constraint",`
	`  "element-query": {`
	`  "ns": "http://my/namespace",`
	`  "name": "title"`
	`  }}`
	`  ]`
	`}}`

31.5.11 properties

A component of a constraint option that limits matches to those found in document properties. To constrain by JSON property, see “container” on page 842.

- Syntax Summary
- Examples
- See Also

31.5.11.1Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON
All elements are in the namespace `http://marklogic.com/appservices/search`. `<properties/>`	`{ "properties": null }`

31.5.11.2Examples

The following example limits matches to those found in document properties.

Format	Example

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

JSON	`{"options": {`
	`  "constraint": [{`
	`  "name": "prop-constraint",`
	`  "properties": null" }`
	`  }]`
	`}}`

31.5.11.3See Also

For more details on using this option, see the following topics:

- “Constraint Options” on page 382
- “Support for Multiple Query Types” on page 26
- `cts:properties-fragment-query`

31.5.12 geo-attr-pair

A component of a constraint option that models a geospatial index with coordinates stored as XML element attributes. That is, geospatial data of the following form:

```
<parent lat="value" lon="value">
```

For similar JSON data, use geo-json-property-pair.

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.12.1Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<geo-attr-pair>`
`  <parent ns="namespace" name="elemName" />`
`  <lat ns="namespace" name="attrName" />`
`  <lon ns="namespace" name="attrName" />`
`  <facet-option>option</facet-option>`
`  <heatmap>heatmap descriptor</heatmap>`
`  <fragment-scope>scope</fragment-scope>`
`  <geo-option>option</geo-option>`
`  <weight>double</weight>`
`</geo-attr-pair>`
	`"geo-attr-pair": {`
	`  "parent": [{`
	`  "ns": "namespace",`
	`  "name": "elemName",`
	`  }],`
	`  "lat": [{`
	`  "ns": "namespace",`
	`  "name": "attrName",`
	`  }],`
	`  "lon": [{`
	`  "ns": "namespace",`
	`  "name": "attrName",`
	`  }],`
	`  "facet-option": [ option ],`
	`  "heatmap": {heatmap desc},`
	`  "geo-option": [ option ],`
	`  "fragment-scope": "scope",`
	`  "weight": double`
	`}`

31.5.12.2Component Description

By default coordinates are stored as (latitude,longitude) points. To reverse the coordinate order, add the geo-option "`long-lat-point`" to the query options configuration. For more details, see `cts:element-attribute-pair-geospatial-query`.

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`parent`	Required. Identifies an element that can contain the latitude and longitude attributes. Use `ns` and `name` to define the namespace (if any) and local name of the element, respectively. If multiple parents are defined, the query matches if any one of them matches. This component can have array value in JSON.
`lat`	Required. Identifies the attribute of `parent` that contains the latitude value. Use `ns` and `name` to define the namespace (if any) and local name of the element, respectively. If multiple elements are defined, the query matches if any one of them matches. However, only the first matching latitude attribute in any point instance is checked. This component can have array value in JSON.
`lon`	Required. Identifies the attribute of `parent` that contains the longitude value. Use `ns` and `name` to define the namespace (if any) and local name of the element, respectively. If multiple elements are defined, the query matches if any one of them matches. However, only the first matching longitude attribute in any point instance is checked. This component can have array value in JSON.
`heatmap`	A model of a two-dimensional grid, used to categorize data along two dimensions for geospatial faceting. For details, see “heatmap” on page 877.

Element, Attribute or Property Name	Description
`facet-option`	Specify options to apply when generating facets. You can only include facet options when there is a `heatmap`. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<facet-option>limit=5</facet-option>` in XML, or `"facet-option": ["limit=5"]` in JSON. For details, see "Facet Options" on page 950.
`geo-option`	Specify options that customize the constraint, such as whether or not to include boundaries. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<geo-option>score-function=linear</geo-option>` in XML, or `"geo-option": ["score-function=linear"]` in JSON. For details, see "Geospatial Point Query Options" on page 952.
`fragment-scope`	Set a local fragment scope for this constraint to further constrain where matches occur. The local fragment scope overrides the global fragment scope. For example, a `fragment-scope` of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents. Allowed values: `properties`, `documents` (default).
`weight`	Higher weights move search results up in the relevance order. The default is 1.0. The weight should be less than or equal to 64 and greater than or equal to -16 (between -16 and 64); weights greater than 64 will have the same effect as a weight of 64. 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.

31.5.12.3Examples

The following example defines a geospatial element attribute pair constraint named "my-geo-attr-pair". Facets will be generated for the constraint, using the region defined in the `heatmap`.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="my-geo-attr-pair">`
	`  <geo-attr-pair>`
	`  <parent ns="ns1" name="my-elem"/>`
	`  <lat ns="ns2" name="attr1"/>`
	`  <lon ns="ns2" name="attr2"/>`
	`  <facet-option>empties</facet-option>`
	`  <heatmap s="23.2" w="-118.3" n="23.3" e="-118.2"`
	`  latdivs="4" londivs="4"/>`
	`  </geo-attr-pair>`
	`  </constraint>`
	`</options>`

JSON	`{"options":{`
	`  "constraint": [{`
	`  "name": "my-geo-attr-pair",`
	`  "geo-attr-pair": {`
	`  "parent": { "ns": "ns1", "name": "my-elem" },`
	`  "lat": { "ns": "ns2", "name": "attr1" },`
	`  "lon": { "ns": "ns3", "name": "attr2" },`
	`  "facet-option": [ "empties" ],`
	`  "heatmap": {`
	`  "s": 23.2, "w": -118.3, "n": 23.3, "e": -118.2,`
	`  "latdivs": 4, "londivs": 4`
	`  }`
	`  }`
	`  }]`
	`}}`

31.5.12.4See Also

For more details on using this option, see the following topics:

- `cts:element-attribute-pair-geospatial-query`
- "Constraint Options" on page 382
- "Geospatial Search Applications" on page 476

31.5.13 geo-elem

A component of a constraint option that models a geospatial index with coordinates stored in a single XML element. That is, geospatial data with the following structure. The specification of a parent element is optional.

```
<parent>
  <elem>lat-lon-values</elem>
</parent>
```

For similar JSON data, use geo-json-property.

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.13.1Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<geo-elem>`
`  <parent ns="namespace" name="elemName" />`
`  <element ns="namespace" name="elemName" />`
`  <facet-option>option</facet-option>`
`  <heatmap>heatmap descriptor</heatmap>`
`  <fragment-scope>scope</fragment-scope>`
`  <geo-option>option</geo-option>`
`  <weight>double</weight>`
`</geo-elem>`
	`"geo-attr-pair": {`
	`  "parent": [{`
	`  "ns": "namespace",`
	`  "name": "elemName",`
	`  }],`
	`  "element": [{`
	`  "ns": "namespace",`
	`  "name": "elemName",`
	`  }],`
	`  "facet-option": [ option ],`
	`  "heatmap": {heatmap desc},`
	`  "geo-option": [ option ],`
	`  "fragment-scope": "scope",`
	`  "weight": double`
	`}`

31.5.13.2Component Description

By default coordinates are stored as (latitude,longitude) points. To reverse the coordinate order, add the geo-option "`long-lat-point`" to the query options configuration. For more details, see `cts:element-geospatial-query`.

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`parent`	Optional. Identifies an element that can contain the latitude and longitude elements. Use `ns` and `name` to define the namespace (if any) and local name of the element, respectively. If multiple parents are defined, the query matches if any one of them matches. This component can have an array value in JSON, but need not if there is only one item.
`element`	Required. Identifies the element containing the latitude and longitude values. Use `ns` and `name` to define the namespace (if any) and local name of the element, respectively. If multiple elements are defined, the query matches if any one of them matches. This component can have an array value in JSON, but need not if there is only one item.
`heatmap`	A model of a two-dimensional grid, used to categorize data along two dimensions for geospatial faceting. For details, see “heatmap” on page 877.

Element, Attribute or Property Name	Description
`facet-option`	Specify options to apply when generating facets. You can only include facet options when there is a `heatmap`. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<facet-option>limit=5</facet-option>` in XML, or `"facet-option": ["limit=5"]` in JSON. For details, see "Facet Options" on page 950.
`geo-option`	Specify options that customize the constraint, such as whether or not to include boundaries. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<geo-option>score-function=linear</geo-option>` in XML, or `"geo-option": ["score-function=linear"]` in JSON. For details, see "Geospatial Point Query Options" on page 952.
`fragment-scope`	Set a local fragment scope for this constraint to further constrain where matches occur. The local fragment scope overrides the global fragment scope. For example, a `fragment-scope` of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents. Allowed values: `properties`, `documents` (default).
`weight`	Higher weights move search results up in the relevance order. The default is 1.0. The weight should be less than or equal to 64 and greater than or equal to -16 (between -16 and 64); weights greater than 64 will have the same effect as a weight of 64. 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.

31.5.13.3Examples

The following example defines two `geo-elem` constraints, one without a parent element specification and one with a parent element specification.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="my-geo-elem">`
	`  <geo-elem>`
	`  <element ns="ns1" name="my-elem"/>`
	`  <geo-option>type=long-lat-point</geo-option>`
	`  </geo-elem>`
	`  </constraint>`
	`  <constraint name="my-geo-elem-child">`
	`  <geo-elem>`
	`  <parent ns="ns1" name="the-parent"/>`
	`  <element ns="ns1" name="the-child"/>`
	`  <facet-option>empties</facet-option>`
	`  <heatmap s="23.2" w="-118.3" n="23.3" e="-118.2"`
	`  latdivs="4" londivs="4"/>`
	`  </geo-elem>`
	`  </constraint>`
	`</options>`

JSON	`{"options":{`
	`  "constraint": [`
	`  { "name": "my-geo-elem",`
	`  "geo-elem": {`
	`  "element": { "ns": "ns1", "name": "my-elem" },`
	`  "geo-option": [ "type=long-lat-point" ]`
	`  }},`
	`  { "name": "my-geo-elem-child",`
	`  "geo-elem": {`
	`  "parent": { "ns": "ns1", "name": "the-parent" },`
	`  "element": { "ns": "ns1", "name": "the-child" },`
	`  "facet-option": [ "empties" ],`
	`  "heatmap": {`
	`  "s": 23.2, "w": -118.3, "n": 23.3, "e": -118.2,`
	`  "latdivs": 4, "londivs": 4`
	`  }`
	`  }}`
	`  ]`
	`}}`

31.5.13.4See Also

For more details on using this option, see the following topics:

- `cts:element-child-geospatial-query`

- `cts:element-geospatial-query`
- "Geospatial Search Applications" on page 476

31.5.14 geo-elem-pair

A component of a constraint option that models a geospatial index with coordinates stored in 2 XML elements that are children of the same parent element. That is, geospatial data of the following form:

```
<parent>
  <lat>latitudeValue</lat>
  <lon>longitudeValue</lon>
</parent>
```

For similar JSON data, use geo-json-property-pair.

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.14.1Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<geo-elem-pair>`
`  <parent ns="namespace" name="elemName" />`
`  <lat ns="namespace" name="attrName" />`
`  <lon ns="namespace" name="attrName" />`
`  <facet-option>option</facet-option>`
`  <heatmap>heatmap descriptor</heatmap>`
`  <fragment-scope>scope</fragment-scope>`
`  <geo-option>option</geo-option>`
`  <weight>double</weight>`
`</geo-elem-pair>`
	`"geo-elem-pair": {`
	`  "parent": [{`
	`  "ns": "namespace",`
	`  "name": "elemName",`
	`  }],`
	`  "lat": [{`
	`  "ns": "namespace",`
	`  "name": "attrName",`
	`  }],`
	`  "lon": [{`
	`  "ns": "namespace",`
	`  "name": "attrName",`
	`  }],`
	`  "facet-option": [ option ],`
	`  "heatmap": {heatmap desc},`
	`  "geo-option": [ option ],`
	`  "fragment-scope": "scope",`
	`  "weight": double`
	`}`

31.5.14.2Component Description

By default coordinates are stored as (latitude,longitude) points. To reverse the coordinate order, add the geo-option "`long-lat-points`" to the query options configuration. For more details, see `cts:element-pair-geospatial-query`.

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`parent`	Required. Identifies an element that can contain the latitude and longitude elements. Use `ns` and `name` to define the namespace (if any) and local name of the element, respectively. If multiple parents are defined, the query matches if any one of them matches. This component can have an array value in JSON, but need not if there is only one item.
`lat`	Required. Identifies the element containing the latitude value. Use `ns` and `name` to define the namespace (if any) and local name of the element, respectively. If multiple elements are defined, the query matches if any one of them matches. However, only the first matching child in any point instance is checked. This component can have an array value in JSON, but need not if there is only one item.
`lon`	Required. Identifies the element containing the longitude value. Use `ns` and `name` to define the namespace (if any) and local name of the element, respectively. If multiple elements are defined, the query matches if any one of them matches. However, only the first matching child in any point instance is checked. This component can have an array value in JSON, but need not if there is only one item.
`heatmap`	A model of a two-dimensional grid, used to categorize data along two dimensions for geospatial faceting. For details, see “heatmap” on page 877.

Element, Attribute or Property Name	Description
`facet-option`	Specify options to apply when generating facets. You can only include facet options when there is a `heatmap`. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<facet-option>limit=5</facet-option>` in XML, or `"facet-option": ["limit=5"]` in JSON. For details, see "Facet Options" on page 950.
`geo-option`	Specify options that customize the constraint, such as whether or not to include boundaries. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<geo-option>score-function=linear</geo-option>` in XML, or `"geo-option": ["score-function=linear"]` in JSON. For details, see "Geospatial Point Query Options" on page 952.
`fragment-scope`	Set a local fragment scope for this constraint to further constrain where matches occur. The local fragment scope overrides the global fragment scope. For example, a `fragment-scope` of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents. Allowed values: `properties`, `documents` (default).
`weight`	Higher weights move search results up in the relevance order. The default is 1.0. The weight should be less than or equal to 64 and greater than or equal to -16 (between -16 and 64); weights greater than 64 will have the same effect as a weight of 64. 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.

31.5.14.3Examples

The following example defines a geospatial element pair constraint named “my-geo-elem-pair”.

Format	Example
XML	<options xmlns="http://marklogic.com/appservices/search"> <constraint name="my-geo-elem-pair"> <geo-elem-pair> <parent ns="ns1" name="the-parent"/> <lat ns="ns2" name="child1"/> <lon ns="ns3" name="child2"/> <geo-option>boundaries-excluded</geo-option> <facet-option>empties</facet-option> <heatmap s="23.2" w="-118.3" n="23.3" e="-118.2" latdivs="4" londivs="4"/> </geo-elem-pair> </constraint> </options>

JSON	{"options":{
	  "constraint": [{
	  "name": "my-geo-elem-pair",
	  "geo-elem": {
	  "parent": { "ns": "ns1", "name": "the-parent" },
	  "lat": { "ns": "ns2", "name": "child1" },
	  "lon": { "ns": "ns3", "name": "child2" },
	  "geo-option": ["boundaries-excluded"],
	  "facet-option": ["empties"],
	  "heatmap": {
	  "s": 23.2, "w": -118.3, "n": 23.3, "e": -118.2,
	  "latdivs": 4, "londivs": 4
	  }
	  }
	  }]
	}}

31.5.14.4See Also

For more details on using this option, see the following topics:

- “Geospatial Search Applications” on page 476
- `cts:element-pair-geospatial-query`

31.5.15 geo-json-property

A component of a constraint option that models a geospatial index with coordinates stored in a single JSON property. That is, geospatial data with the either of the following structures:

```
"parentProperty": { "property": lat-lon-values }

"property": lat-lon-values
```

This topic has the following sections:

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.15.1Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<geo-json-property>`
`  <parent-property>`*name*`</parent-property>`
`  <json-property>`*name*`</json-property>`
`  <facet-option>`*option*`</facet-option>`
`  <heatmap>`*heatmap descriptor*`</heatmap>`
`  <fragment-scope>`*scope*`</fragment-scope>`
`  <geo-option>`*option*`</geo-option>`
`  <weight>`*double*`</weight>`
`</geo-json-property>`
	`"geo-json-property": {`
	`  "parent-property": ["`*name*`"],`
	`  "json-property": ["`*name*`"],`
	`  "facet-option": [ `*option*` ],`
	`  "heatmap": {`*heatmap desc*`},`
	`  "geo-option": [ `*option*` ],`
	`  "fragment-scope": "`*scope*`",`
	`  "weight": `*double*
	`}`

31.5.15.2Component Description

By default coordinates are stored as (latitude,longitude) points. To reverse the coordinate order, add the geo-option "`long-lat-points`" to the query options configuration. For more details, see `cts:json-property-geospatial-query`.

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`parent-property`	Optional. Identifies the parent JSON property that can contain the latitude and longitude property. If multiple parents are defined, the query matches if any one of them matches. This component can have an array value in JSON, but need not if there is only one item.
`json-property`	Required. Identifies the JSON property containing the latitude and longitude values. If multiple properties are defined, the query matches if any one of them matches. This component can have an array value in JSON, but need not if there is only one item.
`heatmap`	A model of a two-dimensional grid, used to categorize data along two dimensions for geospatial faceting. For details, see “heatmap” on page 877.
`facet-option`	Specify options to apply when generating facets. You can only include facet options when there is a `heatmap`. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<facet-option>limit=5</facet-option>` in XML, or `"facet-option": ["limit=5"]` in JSON. For details, see “Facet Options” on page 950.
`geo-option`	Specify options that customize the constraint, such as whether or not to include boundaries. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<geo-option>score-function=linear</geo-option>` in XML, or `"geo-option": ["score-function=linear"]` in JSON. For details, see “Geospatial Point Query Options” on page 952.

Element, Attribute or Property Name	Description
`fragment-scope`	Set a local fragment scope for this constraint to further constrain where matches occur. The local fragment scope overrides the global fragment scope. For example, a `fragment-scope` of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents. Allowed values: `properties`, `documents` (default).
`weight`	Higher weights move search results up in the relevance order. The default is 1.0. The weight should be less than or equal to 64 and greater than or equal to -16 (between -16 and 64); weights greater than 64 will have the same effect as a weight of 64. 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.

31.5.15.3Examples

The following example defines two geospatial JSON property constraints, one with a parent property and one without. The second constraint also illustrates the use of the `geo-option`, `facet-option`, and `heatmap` components.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="my-geo-json-child">`
	`  <geo-json-property>`
	`  <parent-property>parent<parent-property>`
	`  <json-property>child</json-property>`
	`  </geo-json-property>`
	`  </constraint>`
	`  <constraint name="my-geo-json-property">`
	`  <geo-json-property>`
	`  <json-property>location</json-property>`
	`  <geo-option>boundaries-excluded</geo-option>`
	`  <facet-option>empties</facet-option>`
	`  <heatmap s="23.2" w="-118.3" n="23.3" e="-118.2"`
	`  latdivs="4" londivs="4"/>`
	`  </geo-json-property>`
	`  </constraint>`
	`</options>`

Format	Example

JSON	{"options":{
	"constraint": [
	{ "name": "my-geo-json-child",
	"geo-json-property": {
	"parent-property": "parent",
	"json-property": "child",
	}},
	{ "name": "my-geo-json",
	"geo-json-property": {
	"json-property": "location",
	"geo-option": ["boundaries-excluded"],
	"facet-option": ["empties"],
	"heatmap": {
	"s": 23.2, "w": -118.3, "n": 23.3, "e": -118.2,
	"latdivs": 4, "londivs": 4
	}
	}},
	]
	}}

31.5.15.4See Also

For more details on using this option, see the following topics:

- “Geospatial Search Applications” on page 476
- `cts:json-property-geospatial-query`
- `cts:json-property-child-geospatial-query`

31.5.16 geo-json-property-pair

A component of a constraint option that models a geospatial index with coordinates stored in a pair of JSON properties that are children of a specific parent property. That is, geospatial data of the following form:

```
"parentProperty": {
  "latProperty": lat-value,
  "lonProperty": lon-value
}
```

This topic has the following sections:

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.16.1Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<geo-json-property-pair>`
`  <parent-property>`*`name`*`</parent-property>`
`  <lat-property>`*`name`*`</lat-property>`
`  <lon-property>`*`name`*`</lon-property>`
`  <facet-option>`*`option`*`</facet-option>`
`  <heatmap>`*`heatmap descriptor`*`</heatmap>`
`  <fragment-scope>`*`scope`*`</fragment-scope>`
`  <geo-option>`*`option`*`</geo-option>`
`  <weight>`*`double`*`</weight>`
`</geo-json-property-pair>`
	`"geo-json-property-pair": {`
	`  "parent-property": ["`*`name`*`"],`
	`  "lat-property": ["`*`name`*`"],`
	`  "lon-property": ["`*`name`*`"],`
	`  "facet-option": [ `*`option`*` ],`
	`  "heatmap": {`*`heatmap desc`*`},`
	`  "geo-option": [ `*`option`*` ],`
	`  "fragment-scope": "`*`scope`*`",`
	`  "weight": `*`double`*
	`}`

31.5.16.2Component Description

By default coordinates are stored as (latitude,longitude) points. To reverse the coordinate order, add the geo-option "`long-lat-points`" to the query options configuration. For more details, see `cts:json-property-pair-geospatial-query`.

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`parent-property`	Required. Identifies the parent JSON property that can contain the latitude and longitude property. If multiple parents are defined, the query matches if any one of them matches. This component can have an array value in JSON, but need not if there is only one item.
`lat-property`	Required. Identifies the JSON property containing the latitude. If multiple properties are defined, the query matches if any one of them matches. However, only the first matching child in any point instance is checked. This component can have an array value in JSON, but need not if there is only one item.

Element, Attribute or Property Name	Description
`lon-property`	Required. Identifies the JSON property containing the longitude. If multiple properties are defined, the query matches if any one of them matches. However, only the first matching child in any point instance is checked. This component can have an array value in JSON, but need not if there is only one item.
`heatmap`	A model of a two-dimensional grid, used to categorize data along two dimensions for geospatial faceting. For details, see "heatmap" on page 877.
`facet-option`	Specify options to apply when generating facets. You can only include facet options when there is a `heatmap`. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<facet-option>limit=5</facet-option>` in XML, or `"facet-option": ["limit=5"]` in JSON. For details, see "Facet Options" on page 950.
`geo-option`	Specify options that customize the constraint, such as whether or not to include boundaries. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<geo-option>score-function=linear</geo-option>` in XML, or `"geo-option": ["score-function=linear"]` in JSON. For details, see "Geospatial Point Query Options" on page 952.
`fragment-scope`	Set a local fragment scope for this constraint to further constrain where matches occur. The local fragment scope overrides the global fragment scope. For example, a `fragment-scope` of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents. Allowed values: `properties`, `documents` (default).
`weight`	Higher weights move search results up in the relevance order. The default is 1.0. The weight should be less than or equal to 64 and greater than or equal to -16 (between -16 and 64); weights greater than 64 will have the same effect as a weight of 64. 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.

31.5.16.3Examples

The following example defines a geospatial JSON property pair constraint, including the use of the `geo-option`, `facet-option`, and `heatmap` components.

Format	Example
XML	<options xmlns="http://marklogic.com/appservices/search"> <constraint name="my-geo-json-pair"> <geo-json-property-pair> <parent-property>parent</parent-property> <lat-property>child1</lat-property> <lon-property>child2</lon-property> <geo-option>boundaries-excluded</geo-option> <facet-option>empties</facet-option> <heatmap s="23.2" w="-118.3" n="23.3" e="-118.2" latdivs="4" londivs="4"/> </geo-json-property-pair> </constraint> </options>

JSON	{"options":{
	  "constraint": [
	  { "name": "my-geo-json-pair",
	  "geo-json-property": {
	  "parent-property": "parent",
	  "lat-property": "child1",
	  "lon-property": "child2",
	  "geo-option": ["boundaries-excluded"],
	  "facet-option": ["empties"],
	  "heatmap": {
	  "s": 23.2, "w": -118.3, "n": 23.3, "e": -118.2,
	  "latdivs": 4, "londivs": 4
	  }
	  }},
	  ]
	}}

31.5.16.4See Also

For more details on using this option, see the following topics:

- “Geospatial Search Applications” on page 476
- `cts:json-property-pair-geospatial-query`

31.5.17 geo-path

A component of a constraint option that models a geospatial index with coordinates stored in an XML element, XML attribute, or JSON property described by an XPath expression.

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.17.1Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<geo-path>`
`  <path-index/>`
`  <facet-option>`*`option`*`</facet-option>`
`  <heatmap>`*`heatmap descriptor`*`</heatmap>`
`  <fragment-scope>`*`scope`*`</fragment-scope>`
`  <geo-option>`*`option`*`</geo-option>`
`  <weight>`*`double`*`</weight>`
`</geo-path>`
	`"geo-path": {`
	`  "path-index": [`*`path index desc`*`],`
	`  "facet-option": [ `*`option`*` ],`
	`  "heatmap": {`*`heatmap desc`*`},`
	`  "geo-option": [ `*`option`*` ],`
	`  "fragment-scope": `*`"scope"`*`,`
	`  "weight": `*`double`*
	`}`

31.5.17.2Component Description

By default coordinates are stored as (latitude,longitude) points. To reverse the coordinate order, add the geo-option "`long-lat-points`" to the query options configuration. For more details, see `cts:path-geospatial-query`.

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`path-index`	Required. A geospatial range index references to constrain by; for details, see “path-index” on page 888. The path should reference a point index; for region path indexes, see “geo-region-path” on page 871. You can specify more than one `path-index`. The query matches if any one of them matches. This component can have an array value in JSON, but need not if there is only one item. The database configuration must include a geospatial region path index based on the same path. The path expression and namespace URIs must match the index configuration; namespace prefixes do not have to match.
`heatmap`	A model of a two-dimensional grid, used to categorize data along two dimensions for geospatial faceting. For details, see “heatmap” on page 877.
`facet-option`	Specify options to apply when generating facets. You can only include facet options when there is a `heatmap`. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<facet-option>limit=5</facet-option>` in XML, or `"facet-option": ["limit=5"]` in JSON. For details, see “Facet Options” on page 950.
`geo-option`	Specify options that customize the constraint, such as whether or not to include boundaries. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<geo-option>score-function=linear</geo-option>` in XML, or `"geo-option": ["score-function=linear"]` in JSON. For details, see “Geospatial Point Query Options” on page 952.

Element, Attribute or Property Name	Description
`fragment-scope`	Set a local fragment scope for this constraint to further constrain where matches occur. The local fragment scope overrides the global fragment scope. For example, a `fragment-scope` of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents. Allowed values: `properties`, `documents` (default).
`weight`	Higher weights move search results up in the relevance order. The default is 1.0. The weight should be less than or equal to 64 and greater than or equal to -16 (between -16 and 64); weights greater than 64 will have the same effect as a weight of 64. 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.

31.5.17.3Examples

The following example illustrates a geospatial path constraint for an XML element or JSON property addressable with the XPath Expression "/a/b".

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="my-geo-path">`
	`  <geo-path>`
	`  <path-index>/a/b</path-index>`
	`  <geo-option>boundaries-excluded</geo-option>`
	`  <facet-option>empties</facet-option>`
	`  <heatmap s="23.2" w="-118.3" n="23.3" e="-118.2"`
	`  latdivs="4" londivs="4"/>`
	`  </geo-path>`
	`  </constraint>`
	`</options>`

JSON	`{"options":{`
	`  "constraint": [`
	`  { "name": "my-geo-path",`
	`  "geo-path": {`
	`  "path-index": { "text" : "/a/b" },`
	`  "geo-option": [ "boundaries-excluded" ],`
	`  "facet-option": [ "empties" ],`
	`  "heatmap": {`
	`  "s": 23.2,`
	`  "w": -118.3,`
	`  "n": 23.3,`
	`  "e": -118.2,`
	`  "latdivs": 4,`
	`  "londivs": 4`
	`  }`
	`  }},`
	`  ]`
	`}}`

31.5.17.4See Also

For more details on using this option, see the following topics:

- "Geospatial Search Applications" on page 476
- `cts:path-geospatial-query`

31.5.18 geo-region-path

A component of a constraint option that models a geospatial region index with the location of region coordinates described by an XPath expression.

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.18.1Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<geo-region-path coord="coord-sys">`
`  <path-index/>`
`  <fragment-scope>scope</fragment-scope>`
`  <geo-option>option</geo-option>`
`  <weight>double</weight>`
`</geo-region-path>`
	`"geo-region-path": {`
	`  "path-index": [path index desc],`
	`  "coord": "coord-sys",`
	`  "geo-option": [ option ],`
	`  "fragment-scope": "scope",`
	`  "weight": double`
	`}`

31.5.18.2Component Description

By default, coordinates are assumed to be (latitude,longitude) points. To reverse the coordinate order, add the geo-option "`long-lat-points`" to the query options configuration. For more details, see `cts:geospatial-region-query`.

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`path-index`	Required. A geospatial region path range index descriptor for an XML element or JSON property whose contents represent a region. Define any required namespace bindings as part of the `path-index` element or property. For details, see "path-index" on page 888. For geospatial point constraints, see "geo-path" on page 867. You can specify more than one `path-index`. The query matches if any one of them matches. This component can have an array value in JSON, but need not if there is only one item. The database configuration must include a geospatial region path index based on the same path. The path expression and namespace URIs must match the index configuration; namespace prefixes do not have to match.
`coord`	The coordinate system of the region path index. If present, this value must match the index configuration. If the coordinate system and precision are not explicitly specified, "wgs84" (single precision) is assumed. For a list of allowed values, see the options for `cts:geospatial-region-path-reference`.
`geo-option`	Specify options that customize the constraint, such as the units to be used for computations. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<geo-option>units=feet</geo-option>` in XML, or `"geo-option": ["units=feet"]` in JSON. For details, see "Geospatial Region Query Options" on page 953.

Element, Attribute or Property Name	Description
`fragment-scope`	Set a local fragment scope for this constraint to further constrain where matches occur. The local fragment scope overrides the global fragment scope. For example, a `fragment-scope` of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents. Allowed values: `properties`, `documents` (default).
`weight`	Higher weights move search results up in the relevance order. The default is 1.0. The weight should be less than or equal to 64 and greater than or equal to -16 (between -16 and 64); weights greater than 64 will have the same effect as a weight of 64. 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.

31.5.18.3Examples

The following example defines a geospatial region constraint for an XML element or JSON property addressable with the XPath Expression "/a/b".

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="my-geo-region">`
	`  <geo-region-path coord="wgs84">`
	`  <path-index>/a/b</path-index>`
	`  <geo-option>units=feet</geo-option>`
	`  </geo-region-path>`
	`  </constraint>`
	`</options>`

JSON	`{"options":{`
	`  "constraint": [`
	`  { "name": "my-geo-region",`
	`  "geo-region-path": {`
	`  "path-index": {"text": "/a/b"},`
	`  "coord": "wgs84",`
	`  "geo-option": [ "units=feet" ]`
	`  }}`
	`  ]`
	`}}`

31.5.18.4See Also

For more details on using this option, see the following topics:

- "Geospatial Search Applications" on page 476
- `cts:geospatial-region-query` (XQuery)
- `cts.geospatialRegionQuery` (Server-Side JavaScript)

31.5.19 custom

A component of a constraint that defines a custom constraint along with the functions that implement it.

- Syntax Summary
- Component Description
- Examples
- See Also

Warning Custom constraints and other hooks on the Search API cannot be implemented as JavaScript MJS modules.

31.5.19.1Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<custom facet="boolean">`
`  <parse apply="funcName"`
`  ns="namespace"`
`  at="/path/to/module.xqy"/>`
`  <start-facet apply="funcName"`
`  ns="namespace"`
`  at="/path/to/module.xqy"/>`
`  <finish-facet apply="funcName"`
`  ns="namespace"`
`  at="/path/to/module.xqy"/>`
`  <facet-option>option</facet-option>`
`  <term-option>option</term-option>`
`</custom>`
	`"custom": {`
	`  "facet": boolean`
	`  "parse": {`
	`  "apply": "funcName",`
	`  "ns": "namespace",`
	`  "at": "/path/to/module.xqy"`
	`  },`
	`  "start-facet": {`
	`  "apply": "funcName",`
	`  "ns": "namespace",`
	`  "at": "/path/to/module.xqy"`
	`  },`
	`  "finish-facet": {`
	`  "apply": "funcName",`
	`  "ns": "namespace",`
	`  "at": "/path/to/module.xqy"`
	`  },`
	`  "facet-option": [ option ],`
	`  "term-option": [ option ]`
	`}`

31.5.19.2Component Description

The components of this option have the following semantics. The functions that implement the constraint behavior are identified by a name (`apply`) and optional namespace (`ns`), and the full path to the module that contains the function implementation (`at`).

Element, Attribute or Property Name	Description
`facet`	Required. Whether or not to use this constraint for faceting. If `facet` is false, you do not need to specify `start-facet` or `finish-facet` functions. If facet is true, you must specify a finish-facet function and can specify a `start-facet` function.
`parse`	Required. Identifies the function used to parse the input query text or structured query.
`start-facet`	Identifies a function that makes lexicon API calls to return the values and counts used to construct facets from this constraint. Required if `facet` is true and you use the `concurrent` facet option; optional otherwise.
`finish-facet`	Identifies a function that accepts input from the `start-facet` function (if used) and constructs a facet element.
`facet-option`	Specify options to apply when generating facets. In XML, specify multiple options by including the element multiple times. The element or array item value is of the form *`option=value`*. For example: `<facet-option>limit=5</facet-option>` in XML, or `"facet-option": ["limit=5"]` in JSON. For details, see “Facet Options” on page 950.
`term-option`	Specify options to apply when generating facets. In XML, specify multiple options by including the element multiple times. If the option has a value, the element or array item value is of the form *`option=value`*. For example: `<term-option>lang=en</term-option>` in XML, or `"term-option": ["lang=en"]` in JSON. For details, see “Term Options” on page 950.

31.5.19.3Examples

The following example illustrates a custom constraint that supplies `parse`, `start-facet`, and `finish-facet` functions. All three functions are in the namespace "my-namespace", implemented in an XQuery module installed as /my/module.xqy. For a complete example that includes function implementations, see "Creating a Custom Constraint" on page 42.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="my-custom">`
	`  <custom facet="true">`
	`  <parse apply="my-parse-function"`
	`  ns="my-namespace" at="/my/module.xqy"/>`
	`  <start-facet apply="my-start-function"`
	`  ns="my-namespace" at="/my/module.xqy"/>`
	`  <finish-facet apply="my-finish-function"`
	`  ns="my-namespace" at="/my/module.xqy"/>`
	`  <facet-option>concurrent</facet-option>`
	`  </custom>`
	`  </constraint>`
	`</options>`

JSON	`{"options": {`
	`  "constraint": [`
	`  { "name": "my-custom",`
	`  "custom": {`
	`  "facet": true,`
	`  "parse": {`
	`  "apply": "my-parse-function",`
	`  "ns": "my-namespace",`
	`  "at": "/my/module.xqy"`
	`  },`
	`  "start-facet": {`
	`  "apply": "my-start-function",`
	`  "ns": "my-namespace",`
	`  "at": "/my/module.xqy"`
	`  },`
	`  "finish-facet": {`
	`  "apply": "my-finish-function",`
	`  "ns": "my-namespace",`
	`  "at": "/my/module.xqy"`
	`  },`
	`  "facet-option": [ "concurrent" ],`
	`  }},`
	`  ]`
	`}}`

31.5.19.4See Also

For more details on using this option, see the following topics:

- "Creating a Custom Constraint" on page 42

31.5.20 heatmap

A component of a geospatial constraint that models a two-dimensional grid used to categorize data along two dimensions. Use with geospatial indexes and queries to generate geospatial facets in the form of boxes, similar to the boxes created by `cts:geospatial-boxes`.

A `heatmap` can only occur as a child of a geospatial constraint, such as geo-elem or geo-json-property. A heatmap is required for generating facets from a geospatial constraint.

A heatmap is a bounding box defined by 4 cooridnates (n, w, e, s) and the number of latitudinal and longitudinal divisions in which to subdivide the region for faceting purposes. The bounding box is divided equally into the requested number of buckets. The bounding coordinates of the buckets have single point float precision.

A geospatial facets takes the form of a geospatial box with a count of the number of matches within the box. By default, each such box facet is the minimum bounding box of all points in the enclosing bucket. Use the "gridded" facet option to return the bucket coordinates defined by the lat and lon divisions instead. Empty buckets return no facets by default; use the "empties" facet option to return empty bucket boxes.

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.20.1Syntax Summary

This component has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<heatmap s="double" w="double"`
`n="double" e="double"`
`latdivs="unsignedInt",`
`londivs="unsignedInt" />`
	`"heatmap": {`
	`"n": double,`
	`"s": double,`
	`"e": double,`
	`"w": double,`
	`"latdivs": unsignedInt,`
	`"londivs": unsignedInt`
	`}`

31.5.20.2Component Description

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`n`	Required. Heatmap bounding box north coordinate.
`s`	Required. Heatmap bounding box south coordinate.
`e`	Required. Heatmap bounding box east coordinate.
`w`	Required. Heatmap bounding box west coordinate.
`latdivs`	Required. The number of latitude divisions to apply to the bounding box.
`londivs`	Required. The number of longitude divisions to apply to the bounding box.

31.5.20.3Examples

The following example defines a geospatial JSON property pair constraint, with latitude values in `event/latitude` and longitude values in `event/longitude`. Since the constraint defines a `heatmap`, geospatial facets are generated by default. Since the constraint includes the "gridded" facet option, the boxes generated as facets use the dimensions defined by the lat and lon divs defined by the heatmap, rather than the smallest box within each that encompasses all matching points.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="geo-pair">`
	`  <geo-json-property-pair>`
	`  <parent-property>event</parent-property>`
	`  <lat-property>latitude</lat-property>`
	`  <lon-property>longitude</lon-property>`
	`  <facet-option>gridded</facet-option>`
	`  <heatmap s="23.2" w="-118.3" n="23.3" e="-118.2"`
	`  latdivs="4" londivs="4"/>`
	`  </geo-json-property-pair>`
	`  </constraint>`
	`</options>`

JSON	`{"options": {`
	`  "constraint": [{`
	`  "name": "geo-pair",`
	`  "geo-json-property-pair": {`
	`  "parent-property": "event",`
	`  "lat-property": "latitude",`
	`  "lon-property": "longitude",`
	`  "facet-option": [ "gridded" ],`
	`  "heatmap": {`
	`  "s": 24.0, "n": 49.0,`
	`  "e": -67.0,"w": -125.0,`
	`  "latdivs": 5, "londivs": 4`
	`  }}`
	`  }]`
	`}}`

Applied to a search, the heatmap causes the search response to include a `boxes` component, similar to the following:

Format	Search Response Excerpt

XML	`...`
	`<boxes name="geo-pair">`
	`  <box count="2" s="34" w="-125" n="39" e="-110.5"/>`
	`  <box count="12" s="34" w="-110.5" n="39" e="-96"/>`
	`</boxes>`
	`...`

JSON	`...`
	`"facets": {`
	`  "geo-pair": {`
	`  "boxes": [`
	`  { "count": 2,`
	`  "s": 34, "w": -125,`
	`  "n": 39, "e": -110.5`
	`  },`
	`  { "count": 12,`
	`  "s": 34, "w": -110.5,`
	`  "n": 39, "e": -96`
	`  }`
	`  ]`
	`...`

31.5.20.4See Also

For more details on using this option, see the following topics:

- “Geospatial Constraint Example” on page 393
- “Geospatial Search Applications” on page 476
- “Creating Geospatial Facets” on page 525

31.5.21 bucket

A component of a range constraint that defines a range of static values within the constraint that can be used in range query expressions and for generating facets. For dynamic value ranges, refer to “computed-bucket” on page 884.

Values assigned to a bucket meet the following criteria:

```
ge <= value < lt
```

Where `ge` and `lt` represent the "ge" and "lt" values specified in the bucket descriptor.

- Syntax Summary

- Component Description
- Examples
- See Also

31.5.21.1Syntax Summary

This component has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API. The bucket definition must contain either a `lt` or `ge` value and may contain both.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<bucket name="name"`
`ge="value" lt="value">`
`displayLabel`
`</bucket>`
	`"bucket": [{`
	`"name": "name",`
	`"label": "displayLabel",`
	`"lt": value,`
	`"ge": value`
	`}]`

31.5.21.2Component Description

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`name`	Required. The bucket identifier. The name should be unique within the context of the enclosing constraint. The bucket name can be used in string query terms to represent values that fall into the bucket. For details, see “Examples” on page 886.
`label (JSON)` `fn:data() (XML)`	Text that can be used to label the bucket when displaying facets. In XML, this is the text data contained in the `<bucket/>` element. Optional. The display label has no functional use in searches. If present, it is returned in facets so that applications can use it for display purposes.
`lt`	The upper bound for values assigned to this bucket. Optional, but a bucket must include at least one boundary value. Values assigned to this bucket must be less than this value. Must be an atomic value.
`ge`	The lower bound for values assigned to this bucket. Optional, but a bucket must include at least one boundary value. Value assigned to this bucket must be greater than or equal to this value. Must be an atomic value.

31.5.21.3Examples

The following example defines a path range constraint that includes 3 buckets for faceting, corresponding to matches in the ranges ($x < 5$), ($5 <= x < 10$), and ($10 <= x < 15$). For display purposes, these buckets are labeled “less than 5”, “5 to 9”, and “10 to 15”.

You can use the bucket names in string query range expressions. For example, a string search for “pindex:low” finds values less than 5 in nodes with the path `/Employee/fn`.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="pindex">`
	`  <range type="xs:string" facet="true">`
	`  <path-index>/Employee/fn</path-index>`
	`  <bucket name="low" lt="5">less than 5</bucket>`
	`  <bucket name="medium" lt="10" ge="5">5 to 9</bucket>`
	`  <bucket name="high" lt="15" ge="10">10 to 15</bucket>`
	`  </range>`
	`  </constraint>`
	`</options>`

JSON	`{"options": {`
	`  "constraint": [{`
	`  "name": "pindex",`
	`  "range": {`
	`  "type": "xs:string",`
	`  "facet": true,`
	`  "path-index": {"text": "/Employee/fn"},`
	`  "bucket": [`
	`  { "name": "low",`
	`  "lt": "5",`
	`  "label": "less than 5"`
	`  },`
	`  { "name": "medium",`
	`  "ge": "5",`
	`  "lt": "10",`
	`  "label": "5 to 9"`
	`  },`
	`  { "name": "high",`
	`  "ge": "10",`
	`  "lt": "15",`
	`  "label": "10 to 15"`
	`  }`
	`  ]`
	`  }`
	`  }]`
	`}}`

31.5.21.4See Also

For more details on using this option, see the following topics:

- “range” on page 825
- “Buckets Example” on page 62
- “Example: Path Range Index Constraint Query Options” on page 416

31.5.22 computed-bucket

A component of a range constraint that defines a range of dynamic values within the constraint that can be used in range query expressions and for generating facets. For static value ranges, refer to “bucket” on page 880.

Values assigned to a given computed bucket meet the following criteria:

```
(anchor + ge) <= value < (anchor + lt)
```

Where `anchor`, `ge`, and `lt` are values specified in the bucket descriptor.

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.22.1Syntax Summary

This component has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

The bucket definition must include at least one of (`ge`, `lt`) and may include both. You must include at least one anchor value. The anchor value can be shared by both boundaries or be boundary-specific. For example, if the bucket definition includes a `lt` value, then it must include either an `anchor` or `lt-anchor` value.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<computed-bucket name="name"`
`ge="value" lt="value">`
`anchor="anchorValue"`
`lt-anchor="anchorValue"`
`ge-anchor="anchorValue"`
`displayLabel`
`</computed-bucket>`
	`"computed-bucket": [{`
	`"name": "name",`
	`"label": "displayLabel",`
	`"lt": value,`
	`"ge": value,`
	`"anchor": "anchorValue",`
	`"lt-anchor": "anchorValue",`
	`"gt-anchor": "anchorValue"`
	`}]`

31.5.22.2Component Description

The components of this option have the following semantics.

Element, Attribute or Property Name	Description
`name`	Required. The bucket identifier. The name should be unique within the context of the enclosing constraint. The bucket name can be used in string query terms to represent values that fall into the bucket. For details, see “Examples” on page 886.
`label (JSON)` `data() (XML)`	Text that can be used to label the bucket when displaying facets. In XML, this is the text data contained in the `<bucket/>` element. Optional. The display label has no functional use in searches. If present, it is returned in facets so that applications can use it for display purposes.
`lt`	The upper bound for values assigned to this bucket. Optional, but a bucket must include at least one boundary definition. Values assigned to this bucket must be less than this value. Must be an atomic value.

Element, Attribute or Property Name	Description
`ge`	The lower bound for values assigned to this bucket. Optional, but a bucket must include at least one boundary definition. Value assigned to this bucket must be greater than or equal to this value. Must be an atomic value.
`anchor`	A dynamic anchor literal value for the bucket. The bucket bounds are relative to this dynamic value. Optional if you include a boundary-specific anchor (`lt-anchor`, `ge-anchor`) for the included boundary values. Allowed values: `now`, `start-of-day`, `start-of-month`, `start-of-year`.
`lt-anchor`	Overrides `anchor` for computing the bucket upper bound. Optional. Allowed values: `now`, `start-of-day`, `start-of-month`, `start-of-year`.
`ge-anchor`	Overrides `anchor` for computing the bucket lower bound. Optional. Allowed values: `now`, `start-of-day`, `start-of-month`, `start-of-year`.

31.5.22.3Examples

The following example defines an element range constraint on `xs:dateTime` values that includes 4 buckets, corresponding to matches for the values in dynamic ranges relative to “now”.

Format	Example

XML	<options xmlns="http://marklogic.com/appservices/search">
	  <constraint name="made">
	  <range type="xs:dateTime">
	  <element ns="http://example.com" name="manufactured"/>
	  <computed-bucket name="today" ge="P0D" lt="P1D"
	  anchor="now">Today</computed-bucket>
	  <computed-bucket name="30-days" ge="-P30D" lt="P1D"
	  anchor="now">Last 30 days</computed-bucket>
	  <computed-bucket name="60-days" ge="-P60D" lt="P1D"
	  anchor="now">Last 60 Days</computed-bucket>
	  <computed-bucket name="year" ge="-P1Y" lt="P1D"
	  anchor="now">Last Year</computed-bucket>
	  </range>
	  </constraint>
	</options>

Format	Example

JSON	<pre>{ "options": {
	  "constraint": [{
	  "name": "made",
	  "range": {
	  "type": "xs:dateTime",
	  "element": {
	  "ns": "http://example.com",
	  "name": "manufactured"
	  },
	  "computed-bucket": [
	  { "name": "today",
	  "ge": "P0D",
	  "lt": "P1D",
	  "anchor": "now",
	  "label": "Today"
	  },
	  { "name": "30-days",
	  "ge": "-P30D",
	  "lt": "P1D",
	  "anchor": "now",
	  "label": "Last 30 days"
	  },
	  { "name": "60-days",
	  "ge": "-P60D",
	  "lt": "P1D",
	  "anchor": "now",
	  "label": "Last 60 Days"
	  },
	  { "name": "year",
	  "ge": "-P1Y",
	  "lt": "P1D",
	  "anchor": "now",
	  "label": "Last Year"
	  }
	  ]
	  }
	  }]
	}}</pre>

31.5.22.4See Also

For more details on using this option, see the following topics:

- “range” on page 825
- “Computed Buckets Example” on page 64
- “Example: Element Attribute Range Constraint Query Options” on page 418

31.5.23 path-index

This component identifies a path range index for use in a constraint definition.

- Syntax Summary
- Component Description
- Examples
- See Also

31.5.23.1Syntax Summary

This component has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<path-index xmlns:nsPrefix="nsURI">`
`  pathExpression`
`</path-index>`
	`"path-index": {`
	`  "text": "pathExpression"`
	`  "namespaces": {`
	`  "nsPrefix": "nsURI"`
	`  }`
	`}`

31.5.23.2Component Description

In XML, specify the XPath expression as the `path-index` element value. Define any namespace prefix bindings used in the path expression as `xmlns` attributes of the `path-index` element.

In JSON, specify the XPath expression as the value of the "text" property. Define any namespace prefix bindings used in the path expression in the "namespaces" property. The value of a `path-index` can be an array in contexts where you can specify multiple path index references.

The database configuration must include a corresponding path range index.

The path expression is limited to the subset of XPath that can be used to define a path range index. For details, see Path Field and Path-Based Range Index Configuration in *XQuery and XSLT Reference Guide*.

31.5.23.3Examples

The following option defines a path index based range constraint and a path index based values specification. The range constraint references two path indexes, one of which uses namespace prefixes. Path range indexes corresponding to the paths `/data/a` and `/ns1:data/ns2:b` must exist.

Format	Example
XML	<options xmlns="http://marklogic.com/appservices/search"> <constraint name="myPathConstraint"> <range type="xs:int"> <path-index>/data/a</path-index> <path-index xmlns:ns1="/my/ns1" xmlns:ns2="/my/ns2">/ns1:data/ns2:b</path-index> </range> </constraint> <values name="myValuesSpec"> <range type="xs:int"> <path-index>/data/a</path-index> </range> </values> </options>

JSON	{ "options": {
	  "constraint": [{
	  "name": "myPathConstraint",
	  "range": {
	  "type": "xs:int",
	  "path-index": [
	  { "text": "/data/a" },
	  { "text": "/ns1:data/ns2:b",
	  "namespaces": {
	  "ns1": "/my/ns1",
	  "ns2": "/my/ns2"
	  }
	  }
	  ]
	  }
	  }],
	  "values": [
	  { "name": "myValuesSpec",
	  "range": {
	  "type": "xs:int",
	  "path-index": { "text": "/data/a" }
	  }
	  }
	  ]
	} }

31.5.23.4See Also

For more details on using this option, see the following topics:

- “constraint” on page 822
- Defining Path Range Indexes in the *Administrator’s Guide*

31.6 debug

Whether or not to enable debug mode during a search or lexicon query. Allowed values: `true`, `false` (default). When debug mode is enabled, additional report elements are included in the search results summary.

The following example enables debug mode.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <debug>true</debug>`
	`</options>`

JSON	`{"options": {`
	`  "debug": true`
	`}}`

31.7 default-suggestion-source

This option defines a constraint source for generating suggestions for naked terms with `search:suggest`. Suggestions are often used for type-ahead suggestions in a search user interface. For terms qualified by a constraint prefix, such as “tag:value”, use the “suggestion-source” on page 928 option. For more details, see default-suggestion-source Option in the *Search Developer’s Guide*.

- Syntax Summary
- Component Description
- Examples
- See Also

31.7.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

Either use `ref` to specify the name of a constraint defined elsewhere in the in-scope options, or omit `ref` and define a single range, word, or collection constraint or word lexicon.

Your options can only include one default suggestion source.

> **Note:** The use of `word-lexicon` (the database-wide word lexicon) is not recommended as best practice; collection and range lexicons yield the best performance.

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<default-suggestion-source ref="constraint">`
`  <range/>`
`  <word/>`
`  <collection/>`
`  <word-lexicon collation="collURI">`
`  <fragment-scope>scope</fragment-scope>`
`  </word-lexicon>`
`  <suggestion-option>option</suggestion-option>`
`</default-suggestion-source>`
	`"default-suggestion-source" : {`
	`  "ref": "constraintName",`
	`  "range": { constraint defn },`
	`  "word": { constraint defn },`
	`  "collection": {constr. defn},`
	`  "word-lexicon": {`
	`  "collation": "collURI",`
	`  "fragment-scope": "scope"`
	`  },`
	`  "suggestion-option": [option]`
	`}`

31.7.2 Component Description

The components of this option have the following semantics. Note that the `range`, `word`, `collection`, and `word-lexicon` components are mutually exclusive of each other.

Element, Attribute or Property Name	Description
`ref`	The name of a constraint defined in the same set of query options. Optional. If a `ref` value is present and a matching constraint exists, that constraint takes precedence over any `range`, `word`, `collection` or `word-lexicon` defined within this option.
`range`	Defines a range constraint that limits suggestions to values that match this constraint.
`word`	Defines a word constraint that limits suggestions to words that match this constraint.
`collection`	Defines a collection constraint that limits suggestions to collection names. If a prefix is present on the constraint definition, constrain suggestions to collection names with the given prefix.
`word-lexicon`	Limit suggestions to words found in the word lexicon.
`suggestion-option`	Query options to use in generating suggestions. For details, see "Suggestion Options" on page 953.

31.7.3 Examples

The following example constrains suggestions for naked terms to those values in the XML element (or JSON property, in the JSON example) with the name “beast”. The constraint is defined external to the `default-suggestion-source`. A range index must be configured on “beast”. If you request suggestions for the partial search term “an”, the “animal:” constraint prefix is also included in the suggestions because constraint names are always included.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="animal">`
	`  <range type="xs:string">`
	`  <element name="beast"/>`
	`  </range>`
	`  </constraint>`
	`  <default-suggestion-source ref="animal"/>`
	`</options>`

JSON	`{"options": {`
	`  "constraint": [{`
	`  "name": "animal",`
	`  "range": {`
	`  "type": "xs:string",`
	`  "json-property": "beast"`
	`  }}],`
	`  "default-suggestion-source" : {`
	`  "ref": "animal"`
	`  }`
	`}}`

The following example has the same effect as the previous one, but the constraint is defined inside the `default-suggestion-source`.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`<default-suggestion-source>`
	`<range type="xs:string">`
	`<element name="beast"/>`
	`</range>`
	`</default-suggestion-source>`
	`</options>`

JSON	`{"options": {`
	`"default-suggestion-source" : {`
	`"range": {`
	`"type": "xs:string",`
	`"json-property": "beast"`
	`}}`
	`}}`

The following example generates suggestions from collection names with the prefix “/subject/”. For example if you have documents in the collections "/subject/math" and "/subject/science" and you request suggestions for the partial query text “sc”, then “science” is returned as a suggestion.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`<default-suggestion-source>`
	`<collection prefix="/subject/" />`
	`</default-suggestion-source>`
	`</options>`

JSON	`{"options": {`
	`"default-suggestion-source" : {`
	`"collection": { "prefix": "/subject/" }`
	`}`
	`}}`

31.7.4 See Also

For more details on using this option, see the following topics:

- Search Term Completion Using search:suggest in the *Search Developer’s Guide*
- Java: Generating Search Term Completion Suggestions in the *Java Application Developer’s Guide*

- REST: Generating Search Term Completion Suggestions in the *REST Application Developer's Guide*
- Node.js: Generating Search Term Completion Suggestions in the *Node.js Application Developer's Guide*

31.8 extract-document-data

Use `extract-document-data` to select one or more XML elements, XML attributes, or JSON properties from each matching document to return in the results of a document search. Selection is via an absolute XPath expression. The XPath expression in `extract-path` is limited to a subset of XPath; for details, see Restricted XPath in the *XQuery and XSLT Reference Guide*.

- Syntax Summary
- Component Description
- Examples
- See Also

Specify the elements, attributes, or properties as absolute XPath expressions in `extract-path`. Use the `selected` property to indicate what data to return in the selected document components. If `extract-path` is absent or contains no path expressions, the entire content of each matching document is included in the result set.

The manner in which the extracted content is returned depends on the calling context. Using the option in the following calling context returns the extracted content inside the search result summary:

- XQuery: `search:search` or `search:resolve` functions
- REST Client API: `GET:/v1/search` or `POST:/v1/search` with an Accept header of `application/json` or `application/xml`
- Node.js: `DatabaseClient.documents.query` that returns a search result summary instead of documents (`queryBuilder.withOptions({categories: 'none'})`) and includes a result slice with an `extract` clause.
- Java: `QueryManager.search` returns the extracted content in the search results. Access the content using `MatchDocumentSummary.getExtracted`.

Using the option in the following calling contexts returns the extracted content as a sequence of documents:

- XQuery: `search:resolve-nodes`
- REST Client API: `GET:/v1/search` or `POST:/v1/search` with an Accept header of `multipart/mixed` (a multi-document read)
- Node.js: `DatabaseClient.documents.query` that returns documents and includes a result slice with an `extract` clause.

- Java: The `DocumentPage` returned by `DocumentManager.search` contains an extracted document instead of a full document for each match.

31.8.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<extract-document-data selected=value>`
`  <extract-path xmlns:pfx="nsURI">`
`  abs-xpath-expr`
`  </extract-path>`
`</extract-document-data>`
	`"extract-document-data": {`
	`  "selected": value,`
	`  "extract-path": [ abs-xpath-expr ]`
	`}`

31.8.2 Component Description

The components of this option have the following semantics. Setting `selected` to "all" or failing to specify at least one `extract-path` XPath expression means to "extract" the entire document.

Element, Attribute or Property Name	Description
`selected`	How to handle the content selected by `extract-path`. Allowed values: • `include` (default): Include the targeted item(s) in the results. • `include-with-ancestors`: Include the targeted item(s) and all containing ancestors. • `exclude`: Include everything except the targeted item(s). • `all`: Include the entire document For examples of using each setting, see Extracting a Portion of Matching Documents in the *Search Developer's Guide*.
`extract-path`	Optional. The absolute XPath expression of an XML element, XML element attribute, or JSON property that is a target for extraction. You can specify multiple paths. The path expressions are limited to a subset of XPath. For details, see The extract-document-data Query Option in the *XQuery and XSLT Reference Guide*. If a path requires namespace prefixes, define the binding on the `extract-path` XML element, or predefine the binding using a capability such as the REST Client API method `PUT:/v1/config/namespaces` or Java Client API `NamespaceManager` interface.

31.8.3 Examples

The following example targets two items for extraction with the XPath expressions "/my:location" and "/who/userName". Since `selected` is "include", just these elements (or JSON properties) will be include in the extraction results.

A namespace binding is required for "/my:location". In XML options, the binding can be specified on the enclosing `extract-path`, as shown below. When defining options using JSON or when using an API that does not provide this level of control, you must predefine the namespace binding.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <extract-document-data selected="include">`
	`  <extract-path xmlns:my="/my/namespace">/my:location</extract-path>`
	`  <extract-path>/who/userName</extract-path>`
	`  </extract-document-data>`
	`</options>`

JSON	`{"options": {`
	`  "extract-document-data": {`
	`  selected: "include",`
	`  extract-path: [ "/my:location", "/who/userName" ]`
	`  }`
	`} }`

31.8.4 See Also

For more details on using this option, see the following topics:

- Extracting a Portion of Matching Documents in the *Search Developer's Guide*
- REST: Using Namespace Bindings in the *REST Application Developer's Guide*
- Java:
  - Extracting a Portion of Matching Documents in the *Java Application Developer's Guide*
  - Namespaces in the *Java Application Developer's Guide*
- Node.js: Extracting a Portion of Each Matching Document in the *Node.js Application Developer's Guide*

31.9 forest

This option specifies forests to which a search or lexicon query will be constrained. Identify each forest by an unsigned long forest id. If no forest ids are specified, all forests in the database are searched, which is the default behavior.

Note: In the JSON representation, represent the forest ids as a strings because forest ids can exceed the maximum number value supported by JSON.

The following example limits the search to two forests.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <forest>13081837393370312889</forest>`
	`  <forest>13081837393370234811</forest>`
	`</options>`

JSON	`{"options": {`
	`  "forest": ["13081837393370312889","13081837393370234811"]`
	`}}`

For additional examples, see `cts:search`.

31.10 fragment-scope

This option controls the global fragment scope over which to search. The global scope applies to what the search returns (that is, if it returns results from document fragments or from property fragments) and is inherited by any constraints that do not explicitly override the `fragment-scope`.

Allowed values: `properties`, `documents` (default)

You can override the global fragment scope by setting a local fragment scope inside a constraint or term definition. For example, a `fragment-scope` of `properties` on a range constraint enables faceting on a value stored in a property, even if you are searching over documents.

For details, see “Fragment Scope Option” on page 399.

The following example sets the global fragment scope to “properties”.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <fragment-scope>properties</fragment-scope>`
	`</options>`

JSON	`{"options": {`
	`  "fragment-scope": "properties"`
	`}}`

31.11 grammar

Warning Use of this option for grammar customization is deprecated as of MarkLogic 9. You should use a 3rd party library if you require a custom string query grammar. For details, see Search API Grammar Customization Deprecated in the *Release Notes*.

The wrapper element for a custom search grammar definition. The default grammar defines "Google-style" parsing; for details, see “The Default String Query Grammar” on page 68.

- Syntax Summary
- Component Description
- Examples

See the following topics for a detailed description of the starter and joiner components of a grammar.

- starter
- joiner

31.11.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<grammar>`
`  <quotation>quoteChar</quotation>`
`  <implicit options=">ctsQuery</implicit>`
`  <starter/>`
`  <joiner/>`
`</grammar>`
	`"grammar": {`
	`  "quotation": "quoteChar",`
	`  "implict": "ctsQuery",`
	`  "starter": [...],`
	`  "joiner": [...]`
	`}`

31.11.2 Component Description

The components of this option have the following semantics. The grammar should contain at least one `implicit`, `starter`, or `joiner`. If the grammar is empty, then query text is parsed according to the term options.

Element, Attribute or Property Name	Description
`quotation`	The string to use to delimit the start and end of a phrase. You cannot specify a search that includes quotation character. For example, in the default grammar, you cannot search for a double quote (") because this is the `quotation` character.
`implicit`	A serialized `cts:query` used to join two search terms when they are not separated by an explicit operator. By default, the Search API uses a `cts:and-query`,. For example, in the default grammar, “cat dog” is equivalent to “cat AND dog”. Your grammar should include at most one `implicit` component.
`starter`	Define a unary operator or a pair of group delimiters. For example, the default grammar includes a starter for the unary operator “-” that is bound to `cts:not-query` and a starter for “(“ and “)” for grouping. For details, see “starter” on page 903. Your grammar can include zero or more starters.
`joiner`	Define a binary operator that joins two query expressions. For example, the default grammar defines joiners such as “AND”, “OR”, and “LT”. For details, see “joiner” on page 905. Your grammar can include zero or more joiners.

31.11.3 Examples

The following example is a subset of the default grammar. You can obtain the complete grammar by calling `search:get-default-options`.

Format	Example

XML	<options xmlns="http://marklogic.com/appservices/search">
	  <grammar>
	<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>
	  </grammar>
	</options>

JSON	{"options": {
	  "grammar": {
	  "starter": [
	  {
	  "strength": 30,
	  "apply": "grouping",
	  "delimiter": ")",
	  "label": "("
	  },
	  {
	  "strength": 40,
	  "apply": "prefix",
	  "element": "cts:not-query",
	  "label": "-"
	  },
	  ],
	  "joiner": [{
	  "strength": 10,
	  "apply": "infix",
	  "element": "cts:or-query",
	  "tokenize": "word",
	  "label": "OR"
	  }],
	  "quotation": "\"",
	  "implicit": "<cts:and-query strength=\"20\"
	xmlns=\"http://marklogic.com/appservices/search\"
	xmlns:cts=\"http://marklogic.com/cts\"/>"
	  }
	  }
	}

31.11.4 starter

A `starter` is a child of a grammar option that defines a unary operator or a group operator. For more details, see “grammar” on page 900.

Use the apply-at-ns pattern described in Search Customization Via Options and Extensions in *Search Developer's Guide* to define the XQuery library function that implements the starter. The function must produce a cts:query element of the type named by `element`.

- Syntax Summary
- Component Description
- Examples

31.11.4.1Syntax Summary

The `starter` component of a grammar option has the following structure. The operator token is the text data of the `starter` element in XML and the value of the `label` property in JSON. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<starter strength="int"`
`apply="funcName"`
`ns="namespaceURI"`
`at="modulePath"`
`element="qName"`
`options="optList"`
`delimiter="delimChar"`
`tokenize="value">`
`opToken`
`</starter>`
	`"starter": [{`
	`"label": "opToken",`
	`"strength": number,`
	`"apply": "funcName",`
	`"ns": "namespaceURI",`
	`"at": "modulePath",`
	`"element": "qName",`
	`"options": "optList",`
	`"tokenize": "value",`
	`"delimiter": "delimChar",`
	`}]`

31.11.4.2Component Description

The `starter` component of a grammar option has the following semantics:

Element, Attribute or Property Name	Description
`fn:data() (XML)` `label (JSON)`	Required. The starter operator token. For example, the default grammar includes a starter with the minus sign (-) as the operator, which enables query text such as "-cat". If defining a pair of grouping delimiters such as "(" and ")", place set this component to the group start delimiter ("(") and set `delimiter` to the group end deliminter (")").
`strength`	Required. The order of precedence of this starter relative to other operators in this grammar. Higher strength tokens or groups are processed before lower ones.
`apply`	The local-name of a function that parses expressions using this starter. This can be one of the functions pre-defined by the default grammar or a custom function; for a custom function, you must also include `ns` and `at` values.
`ns`	The XQuery module that contains the `apply` function, if using a user-defined function.
`at`	The module from which the `apply` function is imported, if using a user-defined function.
`element`	A `cts:query` element name that identifies the type of cts:query element produced by the parsing function. For example, the default grammar defines a negation starter using the minus sign, which produces a `cts:not-query`.
`options`	A space-separated list of options that are passed through to the underlying `apply` function.
`tokenize`	Allowed values: "word" or "default".
`delimiter`	When defining a pair of grouping deliminters, the string to use as a the delimiter of the end of the grouping. For example, if defining "(" and ")" as group delimiters, set `delimiter` to ")".

31.11.4.3Examples

The following example shows the definition of the "-" (not) unary operator and "( )" grouping operator defined by the default grammar.

Format	Example

XML	`<starter strength="30" apply="grouping" delimiter=")">(</starter>`
	`<starter strength="40" apply="prefix"`
	`  element="cts:not-query">-</starter>`

JSON	`"starter": [`
	`  { "strength": 30,`
	`  "apply": "grouping",`
	`  "delimiter": ")",`
	`  "label": "("`
	`  },`
	`  { "strength": 40,`
	`  "apply": "prefix",`
	`  "element": "cts:not-query",`
	`  "label": "-"`
	`  }`
	`]`

31.11.5 joiner

A `joiner` is a child of a grammar option that defines a binary operator. For more details, see "grammar" on page 900.

Use the apply-at-ns pattern described in Search Customization Via Options and Extensions in *Search Developer's Guide* to define the XQuery library function that implements the joiner. The function must produce a cts:query element of the type named by `element`.

- Syntax Summary
- Component Description
- Examples

31.11.5.1Syntax Summary

The `joiner` component of a grammar option has the following structure. The operator token is the text data of the `starter` element in XML and the value of the `label` property in JSON. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<joiner strength="int"`
`apply="funcName"`
`ns="namespaceURI"`
`at="modulePath"`
`element="qName"`
`options="optList"`
`compare="operator"`
`consume="nTokens"`
`tokenize="value">`
`opToken`
`</joiner>`
	`"joiner": [{`
	`"label": "opToken",`
	`"strength": number,`
	`"apply": "funcName",`
	`"ns": "namespaceURI",`
	`"at": "modulePath",`
	`"element": "qName",`
	`"options": "optList",`
	`"compare": "operator",`
	`"consume": nTokens`
	`"tokenize": "value"`
	`}]`

31.11.5.2Component Description

The `joiner` component of a grammar option have the following semantics:

Element, Attribute or Property Name	Description
`fn:data() (XML)` `label (JSON)`	Required. The joiner operator token. For example, the default grammar includes a joiner with “AND” as the operator, which enables query text such as “cat AND dog”.
`strength`	Required. The order of precedence of this operator relative to other operators in this grammar. Higher strength tokens or groups are processed before lower ones.
`apply`	The local-name of a function that parses expressions using this operator. This can be one of the functions pre-defined by the default grammar or a custom function; for a custom function, you must also include `ns` and `at` values.
`ns`	The XQuery module that contains the `apply` function, if using a user-defined function.

Element, Attribute or Property Name	Description
`at`	The module from which the `apply` function is imported, if using a user-defined function.
`element`	A `cts:query` element name that identifies the type of cts:query element produced by the parsing function. For example, the default grammar defines AND as a joiner that produces a `cts:and-query`.
`options`	A space-separated list of options that are passed through to the underlying `apply` function.
`compare`	The range query comparison operator to apply when defining a relational operator. Allowed values: `LT`, `LE`, `GT`, `GE`, `NE`, `EQ`.
`consume`	The number of tokens to consume when parsing the right operand of the joiner. For example, the `near/` operator defined by the default grammar consumes one token when parsing "A NEAR B", but it consumes two tokens when parsing the expression "A NEAR/3 B" (th count, 3, and B).
`tokenize`	Allowed values: "word" or "default".

31.11.5.3Examples

The following examples from the default grammar define the AND and LT operators. For more examples, see the default grammar.

Format	Example

XML	`joiner strength="20" apply="infix" element="cts:and-query"`
	`tokenize="word">AND</joiner>`
	`<joiner strength="50" apply="constraint" compare="LT"`
	`tokenize="word">LT</joiner>`

JSON	`"joiner": [`
	`{ "strength": 20,`
	`"apply": "infix",`
	`"element": "cts:and-query",`
	`"tokenize": "word",`
	`"label": "label"`
	`},`
	`{ "strength": 50,`
	`"apply": "constraint",`
	`"compare": "LT",`
	`"tokenize": "word",`
	`"label": "LT"`
	`},`
	`]`

31.12 operator

A named wrapper for one or more state elements, each representing a unique run-time configuration option. An operator defines a set of options that can be applied at query time when "*opName*:*state*" is encountered in the query text.

For example, if an operator with the name "sort" is defined, then the query text "sort:foo" will select the "foo" of the "sort" operator at query runtime, using the options specified on that state element. Options affecting query parsing (such as `constraint`, `grammar`, `term`, `empty`) may not be configured via operators.

- Syntax Summary
- Component Description
- Examples
- Examples

31.12.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<operator name="opName">`
`  <state name="stateName">`
`  <additional-query>`
`  ctsQuery`
`  </additional-query>`
`  <debug>boolean</debug>`
`  <forest>forestId</forest>`
`  <page-length>int</page-length>`
`  <quality-weight>double</quality-weight>`
`  <search-option>option</search-option>`
`  <searchable-expression>`
`  pathExpr`
`  </searchable-expression>`
`  <sort-order/>`
`  <transform-results/>`
`  </state>`
`</operator>`
	`"operator: {`
	`  "name": "opName",`
	`  "state": [ {`
	`  "name": "stateName",`
	`  "additional-query": "ctsQuery",`
	`  "debug": boolean,`
	`  "forest": forestId,`
	`  "page-length": integer,`
	`  "quality-weight": double,`
	`  "search-option": [ "option" ],`
	`  "searchable-expression":`
	`  "pathExpr",`
	`  "sort-order": [ ... ],`
	`  "transform-results": ...`
	`  } ]`
	`}`

31.12.2 Component Description

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`name`	Required. The name of this operator. The name must be unique across all operators and constraints in scope.
`state`	One or more state definitions for the operator.
`state/name`	Required. The name of this state.
`state/additional-query`	A cts:query to evaluate when the operator is in this state.
`state/debug`	Whether or not to enable debug logging. Default: false.
`state/forest`	A forest id. This must be an unsigned long value.
`state/page-length`	The page length into which results should be chunked.

Element, Attribute or Property Name	Description
`state/quality-weight`	Specifies a weighting factor to use in the query. The default value is 1.0.
`state/search-option`	Search options passed to the `additional-query`. You can include zero or more options.
`state/searchable-expression`	Specifies an XPath expression to search. For details, see “searchable-expression” on page 921.
`state/sort-order`	Specifies a sort order to apply. For details, see “sort-order” on page 923.
`state/transform-results`	Define a transformation to use when applying this state. For details, see “transform-results” on page 936.

31.12.3 Examples

The following example defines 2 operators, named “sort” and “page-length”. The “sort” operator has 2 states, “down” and “up”. The “nresults” operator has 2 states, “few” and “many”. This allows you, for example, to include a search term of the form `sort:up` to request results in ascending order. Similarly, a search term of the form `nresults:many` cause 50 matches to be returned.

Format	Example
XML	<options xmlns="http://marklogic.com/appservices/search"> <operator name="sort"> <state name="up"> <sort-order> <direction>ascending</direction> <score/> </sort-order> </state> <state name="down"> <sort-order> <direction>descending</direction> <score/> </sort-order> </state> </operator> <operator name="nresults"> <state name="few"> <page-length>10</page-length> </state> <state name="many"> <page-length>50</page-length> </state> </operator> </options>

Format	Example

JSON	{"options": {
	  "operator": [
	  { "name": "sort",
	  "state": [
	  { "name": "down",
	  "sort-order": [{
	  "direction": "descending",
	  "score-order": null
	  }]
	  },
	  { "name": "up",
	  "sort-order": [{
	  "direction": "ascending",
	  "score-order": null
	  }]
	  }
	  ]
	  },
	  { "name": "nresults",
	  "state": [
	  { "name": "few",
	  "page-length": 10
	  },
	  { "name": "many",
	  "page-length": 50
	  }
	  ]
	  }]
	} }

31.12.4 See Also

For more details on using this option, see the following topics:

- “Operator Options” on page 395

31.13 page-length

This option specifies the number of search results to include in each page of returned results. The default value is 10.

The following example sets the page length to 20.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <page-length>20</page-length>`
	`</options>`

JSON	`{"options": {`
	`  "page-length": 20`
	`}}`

31.14 quality-weight

This option specifies a document quality weight to use when computing scores. The value should be a double. The default value is 1.0.

The following example sets the quality weight to 2.0.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <quality-weight>2.0</quality-weight>`
	`</options>`

JSON	`{"options": {`
	`  "quality-weight": 2.0`
	`}}`

31.15 result-decorator

Defines a custom search result decorator XQuery function that is used to decorate each search `result` with additional information. Before performing a search that uses this option, the XQuery library module containing the function must be installed in the App Server modules database at the XPath given by `at`.

- Syntax Summary
- Component Description
- Examples

Warning Result-decorator and other hooks on the Search API cannot be implemented as JavaScript MJS modules.

31.15.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<result-decorator apply="`*`funcName`*`"`
`ns="`*`namespaceURI`*`"`
`at="`*`modulePath`*`" />`
	`"result-decorator": {`
	`"apply": "funcName",`
	`"ns": "`*`namespaceURI`*`",`
	`"at": "`*`modulePath`*`"`
	`}`

31.15.2 Component Description

The components of this option have the following semantics. The module identified by the option must be installed under the App Server root or int he modules database at the location identified by the `at` XPath expression.

Element, Attribute or Property Name	Description
`apply`	The local name of a custom result decorator function. You must also specify `ns` and `at`.
`ns`	The namespace of the function in apply. This must match the namespace declaration of the XQuery library module containing the `apply` function.
`at`	The XPath to the XQuery library module containing the `apply` function.

31.15.3 Examples

The following example specifies the function `decorator` in the XQuery library module located at `/ext/my.domain/decorator.xqy`. assuming the module namespace is `http://marklogic.com/example/my-lib`.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <result-decorator apply="decorator"`
	`  ns="http://marklogic.com/example/my-lib"`
	`  at="/ext/my.domain/decorator.xqy"/>`
	`</options>`

JSON	`{"options": {`
	`  "result-decorator": {`
	`  "apply": "decorator",`
	`  "ns": "http://marklogic.com/example/my-lib",`
	`  "at": "/ext/my.domain/decorator.xqy"`
	`  }`
	`} }`

31.16 return-aggregates

This option specifies whether or not to include the result of running a builtin or user-defined aggregate function in the result of a lexicon query, such as `search:values`. Aggregates are not applicable to document searches (`search:search`).

Default: `false` (do not include). This option applies only to queries against values or tuples.

The following example sets the option to true.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <return-aggregates>true</return-aggregates>`
	`</options>`

JSON	`{"options": {`
	`  "return-aggregates": true`
	`}}`

For details, see the following topics:

- “Return Options” on page 398
- “Using Aggregate Functions” on page 463

- XQuery: `search:values`
- REST: Analyzing Lexicons and Range Indexes With Aggregate Functions in the *REST Application Developer's Guide*

31.17 return-constraints

This option specifies whether or not to include original constraint definitions in the result summary (`search:response`) of a document search. Default: `false` (do not include). For details, see "Return Options" on page 398.

The following example sets the option to true.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <return-constraints>true</return-constraints>`
	`</options>`

JSON	`{"options": {`
	`  "return-constraints": true`
	`}}`

31.18 return-facets

This option specifies whether or not to include facets in the result summary (`search:response`) from a document search. Default: `true` (include).

The following example sets the option to false.

Format	Example

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

JSON	`{"options": {`
	`  "return-facets": false`
	`}}`

For details, see the following topics:

- "Return Options" on page 398
- "Constrained Searches and Faceted Navigation" on page 34

31.19 return-frequencies

This option specifies whether or not to include term frequencies in the search result summary for a values or tuples lexicon query. Default: `true` (include).

For details, see “Return Options” on page 398.

The following example sets the option to false.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <return-frequencies>false</return-frequencies>`
	`</options>`

JSON	`{"options": {`
	`  "return-frequencies": false`
	`}}`

31.20 return-metrics

This option specifies whether or not to include performance metrics in the search result summary (`search:response`). Default: `true` (include).

For details, see “Return Options” on page 398.

The following example sets the option to false.

Format	Example

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

JSON	`{"options": {`
	`  "return-metrics": false`
	`}}`

31.21 return-plan

This option specifies whether or not to include a query plan in the result summary (`search:response`) of a document search. Default: `false` (do not include). The query plan enables you to examine the evaluation plan for a query, which can aid query tuning. For example, you can determine whether or not your range indexes are being used as you expect them to be.

For details, see “Return Options” on page 398 and `xdmp:plan`.

The following example sets the option to true.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <return-plan>true</return-plan>`
	`</options>`

JSON	`{"options": {`
	`  "return-plan": true`
	`}}`

31.22 return-qtext

This option specifies whether or not to include the original query text in the search result summary (`search:response`). Default: `true` (include).

For details, see “Return Options” on page 398.

The following example sets the option to true.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <return-qtext>true</return-qtext>`
	`</options>`

JSON	`{"options": {`
	`  "return-qtext": true`
	`}}`

31.23 return-query

This option specifies whether or not to include the final representation of your query as a serialized cts:query in the result summary (`search:response`) of a document search. Default: `false` (do not include).

For details, see “Return Options” on page 398.

The following example sets the option to true.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <return-query>true</return-query>`
	`</options>`

JSON	`{"options": {`
	`  "return-query": true`
	`}}`

31.24 return-results

This option specifies whether or not to include search result details (`search:result`) in the result summary (`search:response`) of a document search. Default: `true` (include). You can use the transform-results option to control the formatting of the results.

For details, see “Return Options” on page 398.

The following example sets the option to false.

Format	Example

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

JSON	`{"options": {`
	`  "return-results": false`
	`}}`

31.25 return-similar

This option specifies whether or not to include a list of similar documents in each search result in the result summary (`search:response`) of a document search. Default: `false` (do not include).

For details, see “Return Options” on page 398.

The following example sets the option to true.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <return-similar>true</return-similar>`
	`</options>`

JSON	`{"options": {`
	`  "return-similar": true`
	`}}`

31.26 return-values

This option specifies whether or not to include the index/lexicon values in the search results summary when performing a lexicon or index query, such as with `search:values`. Default: `true` (include).

For details, see “Return Options” on page 398.

The following example sets the option to false.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <return-values>false</return-values>`
	`</options>`

JSON	`{"options": {`
	`  "return-values": false`
	`}}`

31.27 search-option

Advanced users can use this option to pass options to the underlying cts query option. For example, use this option to pass through options such as “filtered”, “unfiltered”, and “score-logtfidf”. See the XQuery function `cts:search` for a list of possible values.

This option can appear multiple times in XML and has array value in JSON.

The following example specifies two advanced search options.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <search-option>filtered</search-option>`
	`  <search-option>format-text</search-option>`
	`</options>`

JSON	`{"options": {`
	`  "search-option": ["filtered", "format-text"]`
	`}}`

31.28 searchable-expression

Note: Due to security and performance considerations, beginning in MarkLogic 9.0-10, the `searchable-expression` property/element in query options is deprecated. Please see Search API searchable-expression Deprecated in the Release Notes for more information.

This option specifies an XPath expression to be searched when performing a document search. For example, if you specify `//p`, then `p` elements that match the search criteria are returned.

The expression must be an inline fully searchable XPath expression, and all necessary namespaces must be declared. The default value is `fn:collection()`, which searches all documents in the database.

This option does not affect facet results. To constrain facets, use the `additional-query` option.

- Syntax Summary
- Component Description
- Examples
- See Also

31.28.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<searchable-expression xmlns:`*prefix*`="`*nsURI*`">`
`  `*xpathExpr*
`</searchable-expression>`
	`"searchable-expression": {`
	`  "text": "`*xpathExpr*`",`
	`  "namespaces" : {`
	`  "`*prefix*`": "`*nsURI*`"`
	`  }`
	`}`

31.28.2 Component Description

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`fn:data() (XML)` `text (JSON)`	A fully searchable XPath expression. In XML, the expression is simply the text data of the `searchable-expression` element. If the expression uses namespace prefixes, you must define the prefixes in the option.
`@xmlns:`*prefix* `(XML)` `namespaces (JSON)`	Define a binding between a prefix used in the searchable XPath expression and a namespace URI. In JSON, each child property of the namespaces component is of the form "*prefix*": "*namespaceURI*". In XML, define the binding using the standard xmlns attribute syntax.

31.28.3 Examples

The following example searches over elements addressed by the XPath expression "/ex:orders/com:company".

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <searchable-expression xmlns:ex="http:example.com"`
	`  xmlns:com="http://company.com">`
	`  /ex:orders/com:company`
	`  </searchable-expression>`
	`</options>`

JSON	`{"options": {`
	`  "searchable-expression": {`
	`  "text": "/ex:orders/com:company",`
	`  "namespaces": {`
	`  "ex": "http:example.com",`
	`  "com": "http://company.com"`
	`  }`
	`  }`
	`}}`

31.28.4 See Also

For more details on using this option, see the following topics:

- "Searchable Expression Option" on page 398

31.29 sort-order

Defines the sort order of an XML element, XML element attribute, field, or JSON property during a document search.

A set of query options can contain multiple `sort-order` specifiers. The first such specification is the primary sort order, the second is the secondary sort order, and so on. The default sort order is to sort by score, descending.

- Syntax Summary
- Component Description
- Examples

31.29.1 Syntax Summary

A `sort-order` option must contain exactly one `element`, `field`, or `json-property` child. If there is an `element` child it can optionally have an `attribute` sibling (to specify an attribute of the preceding element). Both the `element` and `attribute` must have `ns` and `name` attributes to specify the namespace and local-name of the specified element and attribute.

A `sort-order` option can contain at most one of the ordering specifiers `confidence-order`, `document-order`, `fitness-order`, `quality-order`, `score-order`, `unordered`, or `score`. Note that some of these ordering algorithms are indistinguishable from each. For example, sorting by fitness and sorting by score yield the same ordering.

The database configuration must include a range index on any XML element, XML element attribute, JSON property, or field used to control sort order.

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<sort-order type="xsType" collation="collURI" direction="value">`
`  <element ns="namespace" name="name"/>`
`  <attribute ns="namespace" name="name"/>`
`  <field name="name"/>`
`  <json-property>name</json-property>`
`  <path-index/>`
`  <confidence-order/>`
`  <document-order/>`
`  <fitness-order/>`
`  <quality-order/>`
`  <undordered/>`
`  <score-order/>`
`  <score/>`
`</sort-order>`
	`"sort-order": {`
	`  "type": "xsType",`
	`  "collation" : "collURI",`
	`  "direction" : "value",`
	`  "element": {`
	`  "ns": "namespace",`
	`  "name": "name"`
	`  },`
	`  "attribute": {`
	`  "ns": "namespace",`
	`  "name": "name"`
	`  },`
	`  "field": {"name": "name"},`
	`  "json-property": "name",`
	`  "path-index": [path index desc],`
	`  "confidence-order": null,`
	`  "document-order": null,`
	`  "fitness-order": null,`
	`  "quality-order": null,`
	`  "unordered": null,`
	`  "score-order": null,`
	`  "score": null`
	`}`

31.29.2 Component Description

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`element`	Identifies an XML element to which this sort order applies, or the element containing the attribute to which this sort order applies. If specifying only an element, you must set `type` to match the range index type of this element. If an `element` is present, you cannot include a `field`, `json-property`, or `path-index` child.
`attribute`	Identifies the element attribute to which this sort order applies. You must set `type` to match the range index type of this attribute. If an `attribute` property is present, you cannot also specify a `field`, `json-property`, or `path-index`, and you must specify `element`.
`field`	Identifies the field to which this sort order applies. Cannot be used with `element`, `attribute`, `path-index`, or `json-property`. You should not apply `sort-order` to a field that has more than one included element.
`json-property`	Identifies the JSON property to which this sort order applies. Cannot be used with `element`, `attribute`, `path-index`, or `field`.
`path-index`	Defines a path range index reference to which this sort order applies. For details, see "path-index" on page 888. Cannot be used with `element`, `attribute`, `json-property`, or `field`. The database configuration must include a corresponding path range index.
`type`	If you are sorting by an element, an attribute, or a JSON property, you must specify a type property with a value corresponding to the range index type of that element, attribute, or property. For example, `xs:string`, `xs:dateTime`, and so on.
`collation`	A collation URI. Optionally specify a collation when `type` is xs:string; otherwise, use the collation of the query.
`direction`	Specify the sorting direction. Default: `ascending`, except for `score`, which defaults to `descending`. Allowed values: `ascending`, `descending`.
`confidence-order`	Sort by confidence order, ascending. Use the `direction` specifier to override the default direction. For details, see Relevance Scores: Understanding and Customizing in the *Search Developer's Guide*. You can define at most one of the ordering specifiers `confidence-order`, `document-order`, `fitness-order`, `quality-order`, `score-order`, `unordered`, or `score`.

Element, Attribute or Property Name	Description
`document-order`	Sort by document order, ascending. Use the `direction` specifier to override the default direction. For details, see Relevance Scores: Understanding and Customizing in the *Search Developer's Guide*. You can define at most one of the ordering specifiers `confidence-order`, `document-order`, `fitness-order`, `quality-order`, `score-order`, `unordered`, or `score`.
`fitness-order`	Sort by fitness order, ascending. Use the `direction` specifier to override the default direction. For details, see Relevance Scores: Understanding and Customizing in the *Search Developer's Guide*. You can define at most one of the ordering specifiers `confidence-order`, `document-order`, `fitness-order`, `quality-order`, `score-order`, `unordered`, or `score`.
`quality-order`	Sort by quality order, ascending. Use the `direction` specifier to override the default direction. For details, see Relevance Scores: Understanding and Customizing in the *Search Developer's Guide*. You can define at most one of the ordering specifiers `confidence-order`, `document-order`, `fitness-order`, `quality-order`, `score-order`, `unordered`, or `score`.
`score-order`	Sort by score order, ascending. Use the `direction` specifier to override the default direction. For details, see Relevance Scores: Understanding and Customizing in the *Search Developer's Guide*. You can define at most one of the ordering specifiers `confidence-order`, `document-order`, `fitness-order`, `quality-order`, `score-order`, `unordered`, or `score`.
`unordered`	Do not apply an ordering. For details, see Relevance Scores: Understanding and Customizing in the *Search Developer's Guide*. You can define at most one of the ordering specifiers `confidence-order`, `document-order`, `fitness-order`, `quality-order`, `score-order`, `unordered`, or `score`.
`score`	Sort by score, descending. This selection is equivalent to using `sort-order` with `direction` set to `descending`. For details, see Relevance Scores: Understanding and Customizing in the *Search Developer's Guide*. You can define at most one of the ordering specifiers `confidence-order`, `document-order`, `fitness-order`, `quality-order`, `score-order`, `unordered`, or `score`.

31.29.3 Examples

The following example specifies a primary sort order using the element value for `my-element` and a secondary sort order of score descending. The database configuration must include a range index on `my-element` with the specified collation.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <sort-order type="xs:string"`
	`  collation="http://marklogic.com/collation/"`
	`  direction="ascending">`
	`  <element ns="my-namespace" name="my-element"/>`
	`  </sort-order>`
	`  <sort-order direction="ascending">`
	`  <score/>`
	`  </sort-order>`
	`</options>`

JSON	`{"options": {`
	`  "sort-order": [`
	`  { "direction": "descending",`
	`  "type": "xs:string",`
	`  "collation": "http://marklogic.com/collation/",`
	`  "element": {`
	`  "name": "my-element",`
	`  "ns": "my-namespace",`
	`  }`
	`  },`
	`  { "direction": "ascending",`
	`  "score": null`
	`  }`
	`  ]`
	`} }`

31.30 suggestion-source

This option defines a search term completion suggestion source for a constraint-qualified search term using `search:suggest` (or an equivalent operation). That is, terms of the form "animal:cat". Suggestions are often used for type-ahead suggestions in a search user interface.

By default, suggestions for constraint-qualified terms are generated based on the qualifying constraint. Use this option to specify an alternative source. This option is useful when you use a named constraint for searching and facets, but you want to use a different source for type-ahead suggestions without re-parsing your search terms.

For naked terms that are unqualified by a constraint prefix, such as "cat", use the default-suggestion-source option. For more details, see default-suggestion-source Option in the *Search Developer's Guide*.

- Syntax Summary
- Component Description
- Examples
- See Also

31.30.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

You must include ref, and it should refer to a named constraint defined elsewhere in your options.

> **Note:** The use of `word-lexicon` (the database-wide word lexicon) is not recommended as best practice; collection and range lexicons yield the best performance.

XML	JSON

All elements are in the namespace
`http://marklogic.com/appservices/search`.
 	
`<suggestion-source ref="constraint">`
`  <range/>`
`  <word/>`
`  <collection/>`
`  <word-lexicon collation="collURI">`
`  <fragment-scope>scope</fragment-scope>`
`  </word-lexicon>`
`  <suggestion-option>option</suggestion-option>`
`</suggestion-source>`
	`"suggestion-source" : [{`
	`  "ref": "constraintName",`
	`  "range": { constraint defn },`
	`  "word": { constraint defn },`
	`  "collection": {constr. defn},`
	`  "word-lexicon": {`
	`  "collation": "collURI",`
	`  "fragment-scope": "scope"`
	`  },`
	`  "suggestion-option": [option]`
	`}]`

31.30.2 Component Description

The components of this option have the following semantics. Use `ref` to identify the constraint to override. Use `range`, `word`, `collection`, or `word-lexicon` to define the overriding suggestion source. Note that the `range`, `word`, `collection`, and `word-lexicon` components are mutually exclusive of each other.

A constraint defined within a `suggestion-source` can include a `collation` value, which specifies the collation of the value lexicon used during query evaluation. If no collation is specified, then the query uses the default collation for the context in which the query is evaluated.

Element, Attribute or Property Name	Description
`ref`	Required. The name of the constraint overridden by this option. The named constraint must be defined elsewhere in the options. If the option specifies only ref with no `range`, `word`, `collection`, or `word-lexicon`, then no suggestions are generated when the named constraint is applied.
`range`	Defines a range constraint that limits suggestions to values that match this constraint.
`word`	Defines a word constraint that limits suggestions to words that match this constraint.
`collection`	Defines a collection constraint that limits suggestions to collection names. If a prefix is present on the constraint definition, constrain suggestions to collection names with the given prefix.
`word-lexicon`	Limit suggestions to words found in the database-wide word lexicon. The use of `word-lexicon` is not recommended as best practice; collection and range lexicons yield the best performance.
`suggestion-option`	Query options to use in generating suggestions. For details, see “Suggestion Options” on page 953.

31.30.3 Examples

The following options define 3 constraints (“color”, “price”, and “pets”), along with a `suggestion-source` option applicable to each. The suggestion source for “color” specifies suggestions matches should be case-sensitive. The suggestion source for “price” specifies that no suggestions should be generated from “price”. The suggestion source for “pets” specifies that suggestions should be drawn from an alternative source, rather than the collection defined by the original constraint

Format	Example
XML	<options xmlns="http://marklogic.com/appservices/search"> <constraint name="color"> <word> <json-property>color</json-property> </word> </constraint> <suggestion-source ref="color"> <suggestion-option>case-sensitive</suggestion-option> </suggestion-source> <constraint name="price"> <range type="xs:unsignedInt"> <element ns="" name="price"/> </range> </constraint> <suggestion-source ref="price"/> <constraint name="pets"> <collection prefix="/my/collection/" facet="true" /> </constraint> <suggestion-source ref="pets"> <collection prefix="/some/collection/subset/" </suggestion-source> </options>

Format	Example

JSON	{"options": {
	  "constraint": [
	  { "name": "color",
	  "word": { "json-property": "color" }
	  },
	  { "name": "price",
	  "range": {
	  "type": "xs:unsignedInt",
	  "element": { "ns": "", "name": "price" }
	  }
	  },
	  { "name": "pets",
	  "collection": { "prefix": "/my/collection", "facet": true}
	  }
	  ],
	  "suggestion-source": [
	  { "ref": "color", "suggestion-option": ["case-sensitive"] },
	  { "ref": "price" },
	  { "ref": "pets",
	  "collection": { "prefix": "/my/collcetion/subset" }
	  }
	  ]
	} }

31.30.4 See Also

For more details on using this option, see the following topics:

- Search Term Completion Using search:suggest in the *Search Developer's Guide*
- Java: Generating Search Term Completion Suggestions in the *Java Application Developer's Guide*
- REST: Generating Search Term Completion Suggestions in the *REST Application Developer's Guide*
- Node.js: Generating Search Term Completion Suggestions in the *Node.js Application Developer's Guide*

31.31 term

This option defines the handling of empty searches and controls options for how individual terms (that is, terms not associated with a constraint) will be represented when parsing the search terms. This option only applies to document searches and search term suggestions.

An “empty” search when you pass an empty query text string to `search:search` or an equivalent operation. To control how empty searches (that is, the empty string passed into `search:search`) are resolved, specify an `empty` child element with an `apply` attribute. The value of the `apply` attribute specifies the behavior for empty searches: a value of `all-results` specifies that empty searches return everything in the database, a value of `no-results` (the default) specifies that an empty search returns nothing.

You can also specify a `default` child element which determines special handling to all terms, so you can have all terms apply a set of rules such as query weighting, specifying a element or attribute value or word query to be used as a default, or specifying a constraint to be applied by default.

Include a `fragment-scope` element to limit the scope of term queries to document or fragment scope. The local definition overrides a global fragment scope on the enclosing `options` element as long as the term definition does not include a constraint definition or reference, or a custom function specification.

A custom term function has the same signature as a custom constraint function. For details about the signature, see Implementing a Structured Query parse Function.

- Syntax Summary
- Component Description
- Examples

31.31.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<term apply="funcName" ns="namespaceURI" at="modulePath">`
`  <default ref="constraintName">`
`  <word/>`
`  <value/>`
`  <range/>`
`  </default>`
`  <empty apply="value" />`
`  <weight>double</weight>`
`  <term-option>option</term-option>`
`  <fragment-scope>scope</fragment-scope>`
`</term>`
	`"term": {`
	`"default": {`
	`  "ref": "constraintName",`
	`  "word": ...,`
	`  "value": ...,`
	`  "range": ...`
	`  },`
	`  "empty": {`
	`  "apply": "value",`
	`  },`
	`  "weight": double,`
	`  "fragment-scope": "scope"`
	`  "apply": "funcName",`
	`  "ns": "namespaceURI",`
	`  "at": "modulePath"`
	`  "term-option": [ "option" ]`
	`}`

31.31.2 Component Description

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`apply`	The local-name of a function that parses expressions using this starter. This can be one of the functions pre-defined by the default grammar or a custom function; for a custom function, you must also include `ns` and `at` values.
`ns`	The XQuery module that contains the `apply` function, if using a user-defined function.
`at`	The module from which the `apply` function is imported, if using a user-defined function.
`default`	Specify default handling of naked terms. Either use `ref` to provide the name of a constraint defined elsewhere in the options, or define a `range`, `value`, or `word` constraint child to apply.

Element, Attribute or Property Name	Description
`empty`	Specify how to resolve an empty, such as when an empty string query is passed to `search:search`. Set apply to either `"all-results"` or `"no-results"`. Default: `all-results`.
`weight`	A weighting factor to apply to individual terms. Default: 1.0.
`term-option`	Zero or more options to apply to term matching. For details, see "Term Options" on page 950.
`fragment-scope`	Specifies a local fragment scope for evaluating term. The local `fragment-scope` overrides any global `fragment-scope`. However, this local scope is ignored if the containing `term` definition includes a constraint definition, constraint reference, or a custom function (via `apply`). Allowed values: `properties`, `documents` (default).

31.31.3 Examples

The following example specifies that an empty search should return no results, and that terms are matched diacritic-insensitive and unwildcarded.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <term>`
	`  <empty apply="no-results" />`
	`  <term-option>diacritic-insensitive</term-option>`
	`  <term-option>unwildcarded</term-option>`
	`  </term>`
	`<options>`

JSON	`{"options": {`
	`  "term": {`
	`  "empty": {"apply": "no-results"},`
	`  "term-options": ["diacritic-insensitive", "unwildcarded"]`
	`  }`
	`}}`

The following example specifies a custom term handling function.

Format	Example
XML	<options xmlns="http://marklogic.com/appservices/search"> <term apply="my-handler" ns="/my/namespace" at="/my/custom.xqy"> <empty apply="all-results" /> </term> <options>

JSON	{"options": {
	  "term": {
	  "apply": "my-handler",
	  "ns": "/my/namespace",
	  "at": "/my/custom.xqy",
	  "empty": {"apply": "no-results"},
	  }
	}}

The following example specifies that terms with no constraint qualifier should be parsed as a weighted word query on the element with QName `foo:bar`:

Format	Example
XML	<options xmlns="http://marklogic.com/appservices/search"> <term> <default> <word> <element ns="foo" name="bar" /> <weight>2.0</weight> </word> </default> </term> <options>

JSON	{"options": {
	  "term": {
	  "default": {
	  "word": {
	  "element": { "ns": "foo", "name": "bar" },
	  "weight" 2.0
	  }
	  }
	  }
	}}

The following example specifies that a constraint defined elsewhere in the options should be applied to terms:

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="my-constraint">...</constraint>`
	`  <term>`
	`  <default ref="my-constraint" />`
	`  </term>`
	`<options>`

JSON	`{"options": {`
	`  "constraint": { "name": "my-constraint", ... },`
	`  "term": {`
	`  "default": {`
	`  "ref": { "my-constraint" }`
	`  }`
	`  }`
	`}}`

31.32 transform-results

Specifies a function to use to process a search result for the snippet output. The default is that each result is formatted using the built-in default snippetting function. For details, see “Modifying Your Snippet Results” on page 401.

You can also use one of the other pre-defined snippeting functions or define your own custom snippet generation function. See the description of `apply` in “Component Description” on page 938. When you define a custom function, you can pass in parameters using the param child elements or properties.

- Syntax Summary
- Component Description
- Examples
- See Also

31.32.1 Syntax Summary

Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

When using a built-in snippeting function, this option can take one of the following forms:

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<transform-results apply="snippet">`
`  <preferred-matches>`
`  <element ns="namespace" name="elemName" />`
`  <json-property>name</json-property>`
`  </preferred-matches>`
`  <per-match-tokens>value</per-match-tokens>`
`  <max-matches>value</max-matches>`
`  <max-snippet-chars>value</max-snippet-chars>`
`</transform-results>`
 	
`<transform-results apply="metadata-snippet">`
`  <preferred-matches>`
`  <element ns="namespace" name="elemName" />`
`  <json-property>name</json-property>`
`  </preferred-matches>`
`</transform-results>`
 	
`<transform-results apply="raw" />`
 	
`<transform-results apply="empty-snippet" />`
	`"transform-results": {`
	`  "apply": "snippet",`
	`  "per-match-tokens": number,`
	`  "max-matches": number,`
	`  "max-snippet-chars": number,`
	`  "preferred-matches": {`
	`  "element": [{`
	`  "ns": "namespaceURI",`
	`  "name": "elemName"`
	`  }],`
	`  "json-property": ["name"]`
	`  }`
	`}`
	
	`"transform-results": {`
	`  "apply": "metadata-snippet",`
	`  "preferred-matches": {`
	`  "element": [{`
	`  "ns": "namespaceURI",`
	`  "name": "elemName"`
	`  }],`
	`  "json-property": ["name"]`
	`  }`
	`}`
	
	`"transform-results": {`
	`  "apply": "raw",`
	`}`
	
	`"transform-results": {`
	`  "apply": "empty-snippet",`
	`}`

To use a custom snippeting function, use the following form of the option:

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<transform-results apply="customFunc"`
`ns="namespaceURI" at="modulePath">`
`*userDefinedParams*`
`</transform-results>`
	Using the built-in snippeters:
	
	`"transform-results": {`
	`"apply": "*funcName*",`
	`"ns": "*namespaceURI*",`
	`"at": "*modulePath*",`
	`"*userDefinedParam*": value`
	`}`

31.32.2 Component Description

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`apply`	The local-name of a built-in or custom function* with which to produce snippet output for a search result. The following are the builtin function names; any other value is taken to be the name of a custom snippeting function. • `snippet` (default) • `raw`: return the whole node • `empty-snippet`: return an empty snippet • `metadata-snippet`: return the snippet from the specified element in the properties document
`preferred-matches`	When using the `snippet` or `metadata-snippet` built-in function, the snippet alogorithm looks for matches first in the specified XML element or JSON property nodes in each snippet. If no matches are found in the preferred elements and/or properties, the algorithm falls back to default content.
`per-match-tokens`	When using the `snippet` built-in function, the maximum number of tokens (typically words) per matching node that surround the highlighted term(s) in the snippet. Default: 30.
`max-matches`	When using the `snippet` built-in function, the maximum number of nodes containing a highlighted term that will display in the snippet. Default: 4.

Element, Attribute or Property Name	Description
`max-snippet-chars`	When using the `snippet` built-in function, limit total snippet size to this many characters. Default: 200.
`ns`	When using a custom snippeting function, the namespace URI of the containing module.
`at`	When using a custom snippeting function, the path to the module containing the function.
`userDefinedParams`	When using a custom snippeting function, specify parameters to pass to the custom function as additional XML elements or JSON properties on the option.

Warning *Custom constraints, decorators, and other hooks on the Search API cannot be implemented as JavaScript MJS modules.

31.32.3 Examples

The following example modifies the characteristics of the default snippeting function (`apply=snippet` is implicit because no function is named).

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <transform-results>`
	`  <max-matches>5</max-matches>`
	`  <max-snippet-chars>100</max-snippet-chars>`
	`  <preferred-matches>`
	`  <element ns="/my/namespace" name="some-element" />`
	`  <json-property>myProperty</json-property>`
	`  </preferred-matches>`
	`  </transform-results>`
	`</options>`

JSON	`{"options": {`
	`  "transform-results": {`
	`  "max-matches": 5,`
	`  "max-snippet-chars": 100,`
	`  "preferred-matches": {`
	`  "element": [{"ns": "/my/namespace", "name": "some-element"}],`
	`  "json-property": ["myProperty"]`
	`  }`
	`  }`
	`}}`

The following example illustrates how to use the “raw” built-in snippeter. Use the same form for “empty-snippet”, but change the value of `apply`.

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <transform-results apply="raw">`
	`  <max-matches>5</max-matches>`
	`  </transform-results>`
	`</options>`

JSON	`{"options": {`
	`  "transform-results": {"apply": "raw"}`
	`}}`

The following example demonstrates how to specify a custom snippet function and pass in some parameter values:

Format	Example

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <transform-results apply="my-snippeter"`
	`  ns="/my/namespace" at="/my-library.xqy">`
	`  <my-param1>42</my-param1>`
	`  <my-param2>abc</my-param2>`
	`  </transform-results>`
	`</options>`

JSON	`{"options": {`
	`  "transform-results": {`
	`  "apply": "my-snippeter",`
	`  "ns": "/my/namespace",`
	`  "at": "/my-library.xqy",`
	`  "my-param1": 42,`
	`  "my-param2": "abc"`
	`  }`
	`}}`

31.32.4 See Also

For more details on using this option, see the following topics:

- “Modifying Your Snippet Results” on page 401
- Customizing Search Snippets in the *REST Application Developer’s Guide*

31.33 tuples

The tuples option identifies values range indexes or lexicons in which to find value co-occurrences. Co-occurrences take the form of a `values-response` in the query response. Use this option with `search:values` and equivalent lexicon query interfaces.

You can specify range and geospatial indexes or the collection or URI lexicons. Include an `aggregate` child to specify an aggregate builtin or user-defined function to apply to the values. You can further tailor your results using top level options such as return-frequencies, return-values, and return-aggregates.

- Syntax Summary
- Component Description
- Examples
- See Also

31.33.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<tuples name="name" style="value">`
`  <range/>`
`  <geo-elem/>`
`  <geo-elem-pair/>`
`  <geo-attr-pair/>`
`  <geo-json-property/>`
`  <geo-json-property-pair/>`
`  <geo-path>`
`  <collection/>`
`  <uri/>`
`  <aggregate apply="funcName"`
`  udf="udfName" />`
`  <values-option>option</values-option>`
`</tuples>`
	`"tuples": {`
	`  "style": "value",`
	`  "name": "name",`
	`  "collection": ...,`
	`  "range": ...,`
	`  "geo-elem": ...,`
	`  "geo-elem-pair": ...,`
	`  "geo-attr-pair": ...,`
	`  "geo-json-property": ...,`
	`  "geo-json-property-pair": ...,`
	`  "geo-path": ...,`
	`  "collection": null,`
	`  "uri": null,`
	`  "aggregate": [{`
	`  "apply": "funcName",`
	`  "udf": string`
	`  }],`
	`"values-option": [ "option" ]`
	`}`

31.33.2 Component Description

The components of this option have the following semantics:

Element, Attribute or Property Name	Description
`name`	Required. The name identifying this tuples definition.
`style`	How to format the results. Allowed values: `default`, `consistent`. When set to `consistent`, the output is structured the same whether returning single values or tuples. When set to `default`, the layout differs between values and tuples results.
`range`	A range constraint with which to extract values from a range index or value lexicon.
`geo-elem`	A geospatial element constraint with which to extract values from a geospatial index.
`geo-elem-pair`	A geospatial element pair constraint with which to extract values from a geospatial index.
`geo-attr-pair`	A geospatial element attribute constraint with which to extract values from a geospatial index.
`geo-json-property`	A geospatial JSON property constraint with which to extract values from a geospatial index.
`geo-json-property-pair`	A geospatial JSON property pair constraint with which to extract values from a geospatial index.
`geo-path`	A geospatial path constraint with which to extract values from a geospatial index.

Element, Attribute or Property Name	Description
`collection`	Extract values from the collection lexicon.
`uri`	Extract values from the URI lexicon.
`aggregate`	Specify builtin or user-defined aggregate functions to apply to lexicon values or value co-occurrences. The `apply` child is required and names the built-in or user-defined aggregate function to apply. When using a UDF, use `udf` to specify the path to the native plugin library containing the implementation of the function. In XML, specify this element multiple times to compute multiple aggregates.
`values-option`	Specify zero or more options that affect the values under consideration. For details, see "Values Options" on page 954.

31.33.3 Examples

The following example specifies co-occurrences between editor and author element values.

Format	Example

XML	<options xmlns="http://marklogic.com/appservices/search">
	  <tuples name="editor-author">
	  <range type="xs:string"
	  collation="http://marklogic.com/collation">
	  <element name="editor" ns="" />
	  </range>
	  <range type="xs:string"
	  collation="http://marklogic.com/collation">
	  <element name="author" ns="" />
	  </range>
	  <values-option>limit=5</values-option>
	  </tuples>
	</options>

JSON	{"options": {
	  "tuples": [{
	  "name": "editor-author",
	  "indexes": [
	  {"range": {
	  "type": "xs:string",
	  "collation": "http://marklogic.com/collation",
	  "element": {
	  "name": "editor",
	  "ns": ""
	  }
	  } },
	  {"range": {
	  "type": "xs:string",
	  "collation": "http://marklogic.com/collation",
	  "element": {
	  "name": "author",
	  "ns": ""
	  }
	  } }
	  ],
	  "values-option": ["limit=5"]
	  }]
	} }

31.33.4 See Also

For details on using this option, see the following topics:

- “Browsing With Lexicons” on page 445

- "Returning Lexicon Values With search:values" on page 54
- "Using Aggregate Functions" on page 463
- Java: Apply Dynamic Query Options to Document Searches in the *Java Application Developer's Guide*
- Node.js: Querying Lexicons and Range Indexes in the *Node.js Application Developer's Guide*
- REST: Querying Lexicons and Range Indexes in the *REST Application Developer's Guide*

31.34 values

The `values` option constrains a values search to one or more range indexes or lexicons. Matched values take the form of a `values-response` in the query response. Use this option with `search:values` and equivalent interfaces.

You can specify range and geospatial indexes or the collection or URI lexicons. Include an `aggregate` child to specify an aggregate builtin or user-defined function to apply to the values. You can further tailor your results using top level options such as return-frequencies, return-values, and return-aggregates.

- Syntax Summary
- Component Description
- Examples
- See Also

31.34.1 Syntax Summary

This option has the following structure. Note that you can only use the JSON form with selected Client APIs, such as the REST Client API.

XML	JSON

All elements are in the namespace `http://marklogic.com/appservices/search`.
 	
`<values name="name" style="value">`
`  <range/>`
`  <geo-elem/>`
`  <geo-elem-pair/>`
`  <geo-attr-pair/>`
`  <geo-json-property/>`
`  <geo-json-property-pair/>`
`  <geo-path>`
`  <collection/>`
`  <uri/>`
`  <aggregate apply="funcName"`
`  udf="udfName" />`
`  <values-option>option</values-option>`
`</values>`
	`"values": {`
	`  "style": "value",`
	`  "name": "name",`
	`  "range": ...,`
	`  "geo-elem": ...,`
	`  "geo-elem-pair": ...,`
	`  "geo-attr-pair": ...,`
	`  "geo-json-property": ...,`
	`  "geo-json-property-pair": ...,`
	`  "geo-path": ...,`
	`  "collection": null,`
	`  "uri": null,`
	`  "aggregate": [{`
	`  "apply": "funcName",`
	`  "udf": string`
	`  }],`
	`"values-option": [ "option" ]`
	`}`

31.34.2 Component Description

The components of this option have the following semantics. You can only include one index/lexicon specification.

Element, Attribute or Property Name	Description
`name`	Required. The name identifying this tuples definition.
`style`	How to format the results. Allowed values: `default`, `consistent`. When set to `consistent`, the output is structured the same whether returning single values or tuples. When set to `default`, the layout differs between values and tuples results.
`range`	A range constraint with which to extract values from a range index or value lexicon.
`geo-elem`	A geospatial element constraint with which to extract values from a geospatial index.

Element, Attribute or Property Name	Description
`geo-elem-pair`	A geospatial element pair constraint with which to extract values from a geospatial index.
`geo-attr-pair`	A geospatial element attribute constraint with which to extract values from a geospatial index.
`geo-json-property`	A geospatial JSON property constraint with which to extract values from a geospatial index.
`geo-json-property-pair`	A geospatial JSON property pair constraint with which to extract values from a geospatial index.
`geo-path`	A geospatial path constraint with which to extract values from a geospatial index.
`collection`	Extract values from the collection lexicon.
`uri`	Extract values from the URI lexicon.
`aggregate`	Specify builtin or user-defined aggregate functions to apply to the lexicon values. The `apply` child is required and names the built-in or user-defined aggregate function to apply. When using a UDF, use `udf` to specify the path to the native plugin library containing the implementation of the function.In XML, specify this element multiple times to compute multiple aggregates.
`values-option`	Specify zero or more options that affect the values under consideration. For details, see “Values Options” on page 954.

31.34.3 Examples

The following example illustrates several kinds of values query specification: An element range index spec that uses `values-option`, a URI lexicon spec, a collection lexicon spec, and a JSON property range index spec that applies a custom aggregate function to index values. For more examples, see `search:values`.

Format	Example
XML	<options xmlns="http://marklogic.com/appservices/search"> <values name="editor-author"> <range type="xs:string" collation="http://marklogic.com/collation"> <element name="editor" ns="" /> </range> <values-option>limit=5</values-option> </values> <values name="uris"> <uri/> </values> <values name="coll"> <collection/> </values> <values name="aggregate"> <range type="xs:double"> <json-property>test</json-property> </range> <aggregate apply="mycalc" udf="sampleplugin" /> </values> </options>

Format	Example

JSON	<pre>{"options": {
	  "values": [
	  { "name": "editor-author",
	  "range": {
	  "type": "xs:string",
	  "collation": "http://marklogic.com/collation",
	  "element": { "name": "editor", "ns": "" }
	  },
	  "values-option": "limit=5"
	  },
	  { "name": "uris",
	  "uri": null
	  },
	  { "name": "coll",
	  "collection": null
	  },
	  { "name": "aggregate",
	  "range": {
	  "type": "xs:double",
	  "json-property": "test"
	  },
	  "aggregate": [{
	  "apply": "mycalc",
	  "udf": "sampleplugin"
	  }]
	  }
	  ]
	} }</pre>

31.34.4 See Also

For details on using this option, see the following topics:

- The XQuery function `search:values`
- “Browsing With Lexicons” on page 445
- “Returning Lexicon Values With search:values” on page 54
- “Using Aggregate Functions” on page 463
- Java: Apply Dynamic Query Options to Document Searches in the *Java Application Developer’s Guide*
- Node.js: Querying Lexicons and Range Indexes in the *Node.js Application Developer’s Guide*
- REST: Querying Lexicons and Range Indexes in the *REST Application Developer’s Guide*

31.35 Term Options

The following options can be specified as the value of a `term-option` child in options that support them, such as a value or word constraint. By default, a query uses the same default options as the underlying cts:query constructors, and the defaults change based on your index configuration.

The following term options are supported:

- `case-sensitive`
- `case-insensitive`
- `diacritic-sensitive`
- `diacritic-insensitive`
- `punctuation-sensitive`
- `punctuation-insensitive`
- `whitespace-sensitive`
- `whitespace-insensitive`
- `stemmed`
- `unstemmed`
- `wildcarded`
- `unwilcarded`
- `exact`
- `lang=`*`iso639code`*
- `distance-weight`
- `lexicon-expand=full`
- `lexicon-expand=prefix-postfix`
- `lexicon-expand=off`
- `lexicon-expand=heuristic`

Both `distance-weight` and `lexicon-expand` options are new with MarkLogic version 8. The following are new with MarkLogic version 9:

- `lexicon-expansion-limit`
- `limit-check`
- `no-limit-check`

31.36 Facet Options

The following options can be specified as the value of a `facet-option` child in constraint types that can be used as a facet (any constraints except word, value, element-query, or property).

Legal values for a `facet-option` are generally any option that can be passed into the underlying query or lexicon API. The following list enumerates the `facet-option` values, but be aware that some options are only available with some range types. For more detail on these options, see the documentation for the underlying range or lexicon APIs.

The following facet options are supported:

- `ascending`
- `descending`
- `empties`
- `any`
- `document`
- `properties`
- `locks`
- `frequency-order`
- `item-order`
- `fragment-frequency`
- `item-frequency`
- `gridded (geospatial constraints only)`
- `type=type`
- `timezone=TZ`
- `limit=N`
- `sample=N`
- `truncate=N`
- `skip=N`
- `score-logtfidf`
- `score-logtf`
- `score-simple`
- `score-random`
- `checked`
- `unchecked`
- `eager`
- `lazy`
- `concurrent`
- `map`

The `concurrent` and `map` options are removed in the latest version of MarkLogic version 7 and all newer releases. The `any`, `document`, `locks`, and `properties` options are new with MarkLogic version 8. The `lazy` and `eager` options are new with MarkLogic version 9.

31.37 Range Options

The following options can be specified as the value of a `range-option` child in options that support them, such as a range constraint.

- `score-function=zero (default)`
- `score-function=reciprocal`
- `score-function=linear`

- `slope-factor=`*number*
- `max-occurs=`*number*
- `min-occurs=`*number*
- `cached`
- `uncached`
- `cached-incremental`
- `synonym`

For details, see Including a Range or Geospatial Query in Scoring and `cts:element-range-query` (XQuery) or `cts.elementRangeQuery` (JavaScript), or equivalent functions for other constraint types. The `cached-incremental` option is new with MarkLogic version 9.

31.38 Geospatial Point Query Options

The following options can be specified as the value of a `geospatial-option` child in options that support them, such as a geo-elem or geo-json-property.

- `coordinate-system=`*coord_sys/precision*
- `coordinate-system=raw`
- `coordinate-system=wgs84/double`
- `coordinate-system=etrs89/double`
- `coordinate-system=raw/double`
- `cached-incremental`
- `precision=float`
- `precision=double`
- `units=`*units*
- `boundaries-included`
- `boundaries-excluded`
- `boundaries-latitude-excluded`
- `boundaries-longitude-excluded`
- `boundaries-south-excluded`
- `boundaries-north-excluded`
- `boundaries-east-excluded`
- `boundaries-west-excluded`
- `boundaries-circle-excluded`
- `type=`*point*
- `type=`*long-lat-point*
- `score-function=zero` (default)

- `score-function=reciprocal`
- `score-function=linear`
- `slope-factor=`*number*
- `cached`
- `uncached`
- `synonym`

31.39 Geospatial Region Query Options

The following options can be specified as the value of a `geospatial-option` child in a geospatial region query, such as a geo-region-path. For more information about the options, see `cts:geospatial-region-query`.

- `score-function=zero` (default)
- `score-function=reciprocal`
- `score-function=linear`
- `slope-factor=`*number*
- `synonym`
- `tolerance=`*distance*
- `units=`*units*

31.40 Suggestion Options

The following options can be specified as the value of a `suggestion-option` child in a default-suggestion-source or suggestion-source option.

- `case-sensitive`
- `case-insensitive`
- `diacritic-sensitive`
- `diacritic-insensitive`
- `ascending`
- `descending`
- `frequency-order`
- `item-order`
- `fragment-frequency`
- `item-frequency`

- `type=`*type*
- `timezone=`*TZ*
- `sample=`*N*
- `truncate=`*N*
- `score-logtfidf`
- `score-logtf`
- `score-simple`
- `score-random`
- `checked`
- `unchecked`
- `any`
- `document`
- `locks`
- `properties`

The `any`, `document`, `locks`, and `properties` options are new with MarkLogic version 8.

31.41 Values Options

The following options can be specified as the value of a `values-option` child in a tuples or values option. For a description of these options, see `cts:values` and `cts:value-tuples` in the *XQuery and XSLT Reference Guide*. Some options can only be used when defining a values configurations; others can only be used when defining tuples.

- `ascending`
- `descending`
- `any`
- `document`
- `properties`
- `locks`
- `frequency-order`
- `item-order`
- `fragment-frequency`
- `item-frequency`
- `timezone=`*TZ*
- `limit=`*N*
- `skip=`*N*
- `sample=`*N*
- `truncate=`*N*
- `score-logtfidf`

- `score-logtf`
- `score-simple`
- `score-random`
- `score-zero`
- `checked`
- `unchecked`
- `eager`
- `lazy`
- `concurrent`
- `map (for values only)`
- `ordered (for tuples only)`
- `proximity=`*`N`*` (for tuples only)`
- `too-many-positions-error`

The `any`, `document`, `locks`, `too-many-positions-error` and `properties` options are new with MarkLogic version 8. The eager and lazy options are new with MarkLogic version 9.

32.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at http://help.marklogic.com to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the Support Handbook for instructions on registering support contacts and on working with the MarkLogic Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at http://developer.marklogic.com. For technical questions, we encourage you to ask your question on Stack Overflow.

33.0 Copyright

MarkLogic Server 10.0 and supporting products.
Last updated: February, 2022

Copyright © 2022 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, US 8,892,599, and US 8,935,267.

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.

For all copyright notices, including third-party copyright notices, see the Combined Product Notices for your version of MarkLogic.