MarkLogic サーバー

『Search Developer's Guide』

MarkLogic 8ea
2015 年初頭

最終更新：8.0-ea、2014 年 11 月

Copyright © 2014 MarkLogic Corporation.All rights reserved.

目次

『Search Developer's Guide』

1.0 MarkLogic サーバーでの検索アプリケーションの開発 15
1.1 MarkLogic サーバーの検索機能の概要 15
1.1.1 ハイパフォーマンスな全文検索 15
1.1.2 XQuery、Java、および REST API 16
1.1.3 複数のクエリタイプのサポート 17
1.1.4 XQuery での XPath 検索のフルサポート 19
1.1.5 レキシコンとレンジインデックスに基づく API 19
1.1.6 ステミング（語幹処理)、ワイルドカード、スペリングほか多数の機能 19
1.1.7 アラート API とビルトイン 20
1.2 検索情報の入手先 20

2.0 search API：概要と使い方 22
2.1 search API の概要 22
2.1.1 XQuery ライブラリモジュール 23
2.1.2 シンプルな search:search の例とレスポンスの出力 23
2.1.3 自動クエリテキストパーシング機能および文法 24
2.1.4 制約付き検索とファセットナビゲーション 26
2.1.5 ビルトインのスニペット作成 27
2.1.6 検索ターム入力支援 28
2.1.7 オプションおよび拡張による検索のカスタマイズ 28
2.1.8 スピードと精度 29
2.2 クエリオプションによる検索の制御 30
2.3 search:suggest による検索ターム入力支援 30
2.3.1 default-suggestion-source オプション 31
2.3.2 suggestion-source オプションを使用した候補の選択 32
2.3.3 search:suggest への複数のクエリテキスト入力の使用 33
2.3.4 カーソル位置に基づいた候補の作成 33
2.3.5 search:suggest の例 34
2.4 カスタム制約の作成 34
2.4.1 parse 関数の実装 35
2.4.2 start-facet 関数の実装 38
2.4.3 finish-facet 関数の実装 39
2.4.4 例：シンプルなカスタム制約の作成 40
2.4.5 例：構造化クエリ用のカスタム制約の作成 41
2.4.6 例：カスタム制約位置情報ファセットの作成 43
2.5 検索文法 47

2.6 search:values を使用してレキシコン値を返す 47
2.6.1 入力レキシコンの指定 48
2.6.2 結果の制約およびフィルタリング 49
2.6.3 例：結果を制約するためのクエリの使用 50
2.6.4 例：開始値、制限、ページ長を使用したフィルタリング 51
2.6.5 例：値の共起の検出 53
2.6.6 追加のインターフェイス 54
2.7 search API での JSON のサポート 54
2.8 search API のその他の例 56
2.8.1 バケットの例 56
2.8.2 計算されるバケットの例 58
2.8.3 ソート順序の例 59

3.0 文字列クエリを使用した検索 60
3.1 文字列クエリの概要 60
3.2 デフォルトの文字列クエリの文法 61
3.2.1 クエリのコンポーネントと演算子 61
3.2.2 演算子の優先順位 65
3.2.3 制約での関係演算子の使用 65
3.2.4 文字列クエリの例 66
3.3 文字列クエリの文法の修正と拡張 66
3.3.1 starter 69
3.3.2 joiner 71
3.3.3 quotation 72
3.3.4 implicit 73

4.0 構造化クエリを使用した検索 74
4.1 構造化クエリの概要 74
4.2 構造化クエリの概念 75
4.2.1 主なクエリカテゴリ 76
4.2.2 タームクエリとワードクエリの違い 77
4.2.3 包含とは 77
4.2.4 完全一致と不完全一致 79
4.2.5 構造化クエリのサブクエリのタクソノミー 80
4.3 構造化クエリの構築 82
4.4 シンタックスの概要 83
4.5 構造化クエリの例 84
4.5.1 例：シンプルな構造化検索 84
4.5.2 例：テキストによる制約参照がある構造化検索 85
4.5.3 例：制約参照を使用した構造化検索 85
4.6 シンタックスリファレンス 87
4.6.1 query 88
4.6.2 term-query 90
4.6.3 and-query 91
4.6.4 or-query 93

4.6.5 and-not-query 95
4.6.6 not-query 97
4.6.7 not-in-query 99
4.6.8 near-query 100
4.6.9 boost-query 102
4.6.10 properties-fragment-query 104
4.6.11 directory-query 107
4.6.12 collection-query 108
4.6.13 container-query 109
4.6.14 document-query 111
4.6.15 document-fragment-query 112
4.6.16 locks-fragment-query 114
4.6.17 range-query 115
4.6.18 value-query 119
4.6.19 word-query 123
4.6.20 geo-elem-query 126
4.6.21 geo-elem-pair-query 129
4.6.22 geo-attr-pair-query 133
4.6.23 geo-path-query 137
4.6.24 geo-json-property-query 141
4.6.25 geo-json-property-pair-query 145
4.6.26 range-constraint-query 149
4.6.27 value-constraint-query 152
4.6.28 word-constraint-query 155
4.6.29 collection-constraint-query 157
4.6.30 container-constraint-query 160
4.6.31 element-constraint-query 162
4.6.32 properties-constraint-query 164
4.6.33 custom-constraint-query 166
4.6.34 geospatial-constraint-query 168
4.6.35 lsqt-query 171
4.6.36 period-compare-query 173
4.6.37 period-range-query 175
4.6.38 operator-state 177

5.0 Query By Example を使用した検索 181
5.1 QBE の概要 181
5.1.1 ドキュメントの構造に基づいた検索条件 183
5.1.2 論理演算子 187
5.1.3 比較演算子 188
5.1.4 値または単語（ワード）によるクエリ 189
5.1.5 検索結果のカスタマイズ 190
5.1.6 検索動作を制御するためのオプション 190
5.2 例 190
5.2.1 XML の例 191
5.2.2 JSON の例 193

5.3 QBE のサブクエリタイプの概要 195
5.3.1 値クエリ 195
5.3.2 ワードクエリ 197
5.3.3 レンジクエリ 198
5.3.4 複合クエリ 200
5.3.5 コンテナクエリ 201
5.4 検索条件のクイックリファレンス 204
5.4.1 XML 検索条件のクイックリファレンス 204
5.4.2 JSON 検索条件のクイックリファレンス 206
5.4.3 ドキュメント全体の検索 209
5.5 QBE 構造リファレンス 211
5.5.1 最上位レベルの構造 212
5.5.2 クエリのコンポーネント 213
5.5.3 レスポンスのコンポーネント 216
5.5.4 XML 固有の考慮事項 216
5.5.5 JSON 固有の考慮事項 218
5.6 インデックス付けがクエリに与える影響 224
5.7 QBE へのオプションの追加 225
5.7.1 XML でのオプションの指定 225
5.7.2 JSON でのオプションの指定 225
5.7.3 オプションリスト 226
5.7.4 永続的なクエリオプションの使用 229
5.8 検索結果のカスタマイズ 230
5.8.1 クエリにレスポンスを含める状況 231
5.8.2 スニペットフォーマッタの使用 232
5.8.3 抽出フォーマッタの使用 234
5.8.4 例：検索のカスタマイズ 235
5.9 ドキュメントタイプによる検索のスコープ設定 237
5.10 QBE の複合クエリへの変換 238

6.0 cts:query 式の構成 239
6.1 cts:query について 239
6.1.1 cts:query とは 240
6.1.2 検索を絞り込むための使い方 241
6.1.3 cts:element-query とは 241
6.1.4 cts:element-word-query とは 241
6.1.5 フィールドワードクエリコンストラクタおよび値クエリコンストラクタとは 242
6.1.6 レンジクエリコンストラクタとは 242
6.1.7 リバースクエリコンストラクタとは 243
6.1.8 位置クエリコンストラクタとは 243
6.1.9 cts:query での言語の指定 243
6.2 複数の cts:query 式の組み合わせ 244
6.2.1 cts:and-query および cts:or-query の使い方 244
6.2.2 cts:near-query を使用した近接クエリ 244
6.2.3 制限付きの cts:query 式の使い方 245

6.2.4 いかなるものともマッチしない / すべてのものとマッチする 245
6.3 cts:properties-query または cts:document-fragment-query による ドキュメントおよびプロパティの結合 246
6.4 検索のスピードを向上させるための cts:query 式の登録 247
6.4.1 クエリ API の登録 247
6.4.2 フィルタリングなしでの使用が必要な場合 247
6.4.3 登録はシステムの再起動で削除される 248
6.4.4 登録済みクエリの ID の格納 248
6.4.5 登録済みクエリおよび関連度の計算 249
6.4.6 例：cts:query 式の登録と使い方 249
6.5 cts:query 式への関連度の情報の追加 249
6.6 cts:query コンストラクタの XML シリアライゼーション 250
6.6.1 cts:query の XML へのシリアライズ 250
6.6.2 cts:annotate による任意の注釈の追加 251
6.6.3 XML から cts:query を構築する関数 251
6.7 例：cts:query のパーサーの作成 252

7.0 クエリオプションを使用した検索のカスタマイズ 255
7.1 はじめに 255
7.2 デフォルトのクエリオプションの取得 256
7.3 クエリオプションにエラーがないかどうかをチェックする 256
7.4 制約オプション 257
7.4.1 値制約の例 263
7.4.2 単語（ワード）制約の例 264
7.4.3 コレクション制約の例 265
7.4.4 バケットレンジ制約の例 266
7.4.5 完全一致（バケット化されていない）レンジ制約の例 267
7.5 演算子オプション 268
7.6 戻り値のオプション 270
7.7 検索可能な式のオプション 271
7.8 フラグメントスコープのオプション 271
7.9 結果のスニペットの修正 272
7.9.1 transform-results のオプションの指定 273
7.9.2 transform-results での独自のコードの指定 275
7.10 デコレータによる検索結果のカスタマイズ 275
7.10.1 検査結果デコレータとは 276
7.10.2 カスタムの検索結果デコレータの記述 277
7.10.3 カスタムの検索結果デコレータのインストール 278
7.10.4 カスタムの検索結果デコレータの使い方 278
7.11 その他の検索オプション 279
7.12 クエリオプションの例 279
7.12.1 例：値およびタプルクエリオプション 279
7.12.2 例：フィールド制約クエリオプション 282
7.12.3 例：コレクション制約クエリオプション 282
7.12.4 例：パスレンジインデックス制約クエリオプション 283
7.12.5 例：要素属性レンジ制約クエリオプション 285

7.12.6 例：位置情報制約クエリオプション 287

8.0 関連度スコア：概要とカスタマイズ 289
8.1 スコアと関連度の計算方法の概要 289
8.1.1 log(tf)*idf の計算 290
8.1.2 log(tf) の計算 291
8.1.3 シンプルなタームマッチ計算 291
8.1.4 ランダムなスコア計算 291
8.1.5 ターム頻度の正規化 292
8.2 フラグメント化およびインデックスオプションがスコアに与える影響 292
8.3 重みでスコアに影響を与える方法 293
8.4 distance-weight オプションによる近接ブースト 294
8.4.1 シンプルな近接ブーストの例 294
8.4.2 cts:and-query セマンティックによる近接ブーストの使い方 295
8.4.3 cts:near-query を使用した近接ブースト 296
8.5 セカンダリクエリによる関連度スコアのブースト 297
8.6 スコア計算にレンジクエリや位置クエリを含める 298
8.6.1 レンジクエリのスコアへの影響 299
8.6.2 レンジクエリのスコアへの影響のユースケース 299
8.6.3 レンジクエリのスコアへの影響をオンにする 300
8.6.4 傾き係数とは 301
8.6.5 パフォーマンスに関する考慮事項 304
8.6.6 レンジクエリのスコア計算の例 304
8.7 スコアとクオリティの関係 309
8.8 cts:score、cts:confidence、cts:fitness の使い方 309
8.9 cts:search の関連度順と XPath のドキュメント順の比較 310
8.10 関連度スコアの計算を調べる 311
8.11 cts:search 式の例 313
8.11.1 ドキュメントのスコアブーストをクオリティで拡大する 313
8.11.2 あるタームのスコアを上げ、他のタームのスコアを下げる 313

9.0 レキシコンを利用した閲覧 314
9.1 レキシコンについて 314
9.2 レキシコンの作成 315
9.3 ワードレキシコン 316
9.3.1 データベース全体のワードレキシコン 317
9.3.2 要素 / 要素属性ワードレキシコン 317
9.3.3 フィールドワードレキシコン 317
9.4 要素 / 要素属性 / パス値レキシコン 318
9.5 フィールド値レキシコン 318
9.6 値共起レキシコン 319
9.7 位置情報レキシコン 321
9.8 レンジレキシコン 322
9.9 URI レキシコンとコレクションレキシコン 323

9.10 レキシコンベースのクエリの実行 323
9.10.1 レキシコン API 324
9.10.2 cts:query 式へのレキシコン検索の制約 324
9.10.3 マッチレキシコン API の使い方 325
9.10.4 レキシコンのタームを含んでいるフラグメントの個数を求める 326

10.0 cts:query 式でのレンジクエリの使用 328
10.1 レンジクエリの概要 328
10.1.1 レンジクエリの使い方 328
10.1.2 レンジクエリを使用するための要件 329
10.1.3 レンジクエリのパフォーマンスとコーディング上の利点 329
10.2 レンジクエリの cts:query コンストラクタ 330
10.3 レンジクエリの例 330

11.0 集計関数の使用 332
11.1 集計関数の概要 332
11.2 ビルトイン集計関数の使用 332
11.3 集計ユーザー定義関数の使用 335

12.0 検索タームのマッチ箇所の強調表示 338
12.1 cts:highlight の概要 338
12.1.1 マッチするすべてのターム（語幹や大文字を含む） 339
12.2 一般的な検索と置換の関数 339
12.3 cts:highlight のビルトイン変数 340
12.3.1 $cts:text 変数を使用して、マッチしたテキストにアクセスする 340
12.3.2 $cts:node 変数を使用して、マッチのコンテキストにアクセスする 341
12.3.3 $cts:queries 変数を使用して、クエリに基づくロジックを提供する 341
12.3.4 $cts:start を使用して、文字数による位置を取得する 342
12.3.5 $cts:action を使用して、強調表示を停止する 342
12.4 cts:highlight を使用して、スニペットを作成する 342
12.5 cts:walk と cts:highlight の比較 344
12.6 一般的な使用上の注意 344
12.6.1 入力はノード 1 つだけであること 344
12.6.2 xdmp:set 使用時の cts:highlight による副作用 345
12.6.3 cts:similar-query または cts:element-attribute-*-query では強調表示はできない 346

13.0 位置検索アプリケーション 347
13.1 MarkLogic サーバーの位置データの概要 347
13.1.1 用語 347

13.1.2 座標系 348
13.1.3 位置クエリのタイプ 349
13.1.4 位置クエリ用の XQuery プリミティブ型とコンストラクタ 349
13.1.5 WKT（Well-Known Text）マークアップ言語 350
13.2 位置情報の座標と領域の概要 351
13.2.1 基本的な座標とポイントの概要 352
13.2.2 位置情報のボックスの概要 352
13.2.3 位置情報の多角形の概要：多角形、複雑な多角形、ラインストリング 354
13.2.4 位置情報の円の概要 356
13.3 位置インデックス 356
13.3.1 さまざまな種類の位置インデックス 356
13.3.2 位置インデックスの位置 360
13.3.3 位置情報レキシコン 360
13.4 API の使用 360
13.4.1 位置クエリを実行する基本的な手順 360
13.4.2 位置情報の cts:query コンストラクタ 361
13.4.3 領域の位置情報値コンストラクタ 361
13.4.4 位置情報の形式変換関数 361
13.4.5 位置情報の操作 362
13.5 シンプルな位置検索の例 362

14.0 エンティティエンリッチメントによるドキュメントのマークアップ 365
14.1 エンティティエンリッチメントの概要 365
14.2 エンティティエンリッチメントパイプライン 365
14.2.1 サードパーティ技術を使用するサンプルパイプライン 366
14.2.2 カスタムエンティティエンリッチメントパイプライン 366

15.0 アラートアプリケーションの作成 367
15.1 MarkLogic サーバーのアラートアプリケーションの概要 367
15.2 cts:reverse-query コンストラクタ 368
15.3 cts:query コンストラクタの XML シリアライズ 369
15.4 アラートアプリケーションのセキュリティ上の考慮事項 369
15.4.1 アラートユーザー、アラート管理者、アクセス制御 369
15.4.2 事前定義されたアラートアプリケーションのロール 370
15.5 リバースクエリのインデックス 371
15.6 アラート API 371
15.6.1 アラート API の概念 371
15.6.2 アラート API の使用 373
15.6.3 CPF とアラートアプリケーションの使用 377
15.7 サンプルアラートアプリケーション 378

16.0 fn:count と xdmp:estimate の使い方の比較 379
16.1 fn:count は正確、xdmp:estimate は高速 379
16.2 xdmp:estimate ビルトイン関数 379
16.3 検索サイズの推定のための cts:remainder の使い方 380
16.4 xdmp:estimate を使用する場合 381
16.4.1 推定値が十分に有益である場合 382
16.4.2 XPath が適切な条件を満たす場合 383
16.4.3 経験的テストによって正確性が示される場合 384

17.0 ステミング検索の概要と使い方 385
17.1 MarkLogic サーバーでのステミング 385
17.2 ステミングをオンにする 386
17.3 ステミング検索とワード検索の比較 387
17.4 語（ワード）がクエリと一致するかどうかを確認するために cts:highlight または cts:contains を使用する 388
17.5 ワイルドカード検索との関係 388

18.0 トークン化とステミングのためのカスタム辞書 389
18.1 MarkLogic サーバーのカスタム辞書 389
18.2 辞書とエントリーのスキーマ 390
18.3 カスタム辞書の関数 391
18.3.1 ライセンスされたすべての言語の取得 392
18.3.2 カスタム辞書の取得 392
18.3.3 カスタム辞書の追加 / 書き込み 392
18.3.4 カスタム辞書の削除 393
18.4 使用例 393

19.0 バイナリドキュメントからのメタデータとテキストの抽出 396
19.1 メタデータとテキスト抽出の概要 396
19.2 使用例 396
19.2.1 Microsoft Word 397
19.2.2 ファイルアーカイブ 398
19.2.3 PowerPoint 400
19.3 サポートされているバイナリ形式 401
19.3.1 アーカイブ 402
19.3.2 データベース 402
19.3.3 電子メールとメッセージング 402
19.3.4 マルチメディア 402
19.3.5 その他 403
19.3.6 プレゼンテーション 403
19.3.7 ラスター画像 403
19.3.8 表計算 403
19.3.9 テキストとマークアップ 404
19.3.10ベクトル画像 404

19.3.11ワードプロセッサおよびオフィス全般 404

20.0 ワイルドカード検索の概要と使い方 405
20.1 MarkLogic サーバーでのワイルドカード 405
20.1.1 ワイルドカード文字 405
20.1.2 ワイルドカード検索の規則 405
20.2 ワイルドカード検索をオンにする 406
20.2.1 クエリでのワイルドカードの指定 407
20.2.2 推奨されるワイルドカードインデックス設定 408
20.2.3 さまざまなワイルドカードインデックスについて 408
20.3 他の検索機能との関係 410
20.3.1 ワイルドカード機能とステミング 411
20.3.2 ワイルドカード機能と句読点による区別 411

21.0 コレクション 416
21.1 collection() 関数 416
21.2 コレクションとディレクトリの比較 417
21.3 コレクションの定義 418
21.3.1 保護されていないコレクションの暗黙的な定義 419
21.3.2 保護されているコレクションの明示的な定義 420
21.4 コレクションのメンバーシップ 420
21.5 コレクションとセキュリティ 421
21.5.1 保護されていないコレクション 422
21.5.2 保護されているコレクション 422
21.6 パフォーマンス特性 423
21.6.1 ドキュメントが所属するコレクションの数 423
21.6.2 コレクションへの既存のドキュメントの追加と削除 424

22.0 シソーラス関数の使用 425
22.1 シソーラスモジュール 425
22.2 関数リファレンス 425
22.3 シソーラススキーマ 426
22.4 大文字 / 小文字の区別 426
22.5 シソーラスドキュメントの管理 426
22.5.1 シソーラスドキュメントの読み込み 427
22.5.2 シソーラスドキュメントの挿入時のタームの小文字化 427
22.5.3 WordNet シソーラスの XML バージョンの読み込み 428
22.5.4 シソーラスドキュメントの更新 428
22.5.5 シソーラスドキュメントのセキュリティ上の考慮事項 429
22.5.6 シソーラス管理関数を使用するクエリの例 429
22.6 シソーラスによる検索の拡張 432

23.0 スペリング修正関数の使用 433
23.1 スペリング修正の概要 433

23.2 関数リファレンス 433
23.2.1 スペリングビルトイン関数 434
23.2.2 スペリング辞書管理モジュール関数 434
23.3 辞書ドキュメント 435
23.4 大文字 / 小文字の区別 435
23.5 辞書ドキュメントの管理 436
23.5.1 辞書ドキュメントの読み込み 436
23.5.2 サンプル XML 辞書の読み込み 436
23.5.3 辞書ドキュメントの更新 437
23.5.4 辞書ドキュメントのセキュリティ上の考慮事項 438
23.6 単語のスペルが正しいかどうかのテスト 439
23.7 単語のスペルが正しくない場合のスペルの候補の取得 439

24.0 distinctive terms と cts:similar-query 440
24.1 cts:similar-query の概要 440
24.2 一連のノードに含まれている distinctive terms の検索 440
24.3 cts:distinctive-terms の出力の概要 441
24.4 デザインパターンの例：タグクラウドの作成 443

25.0 分類子の学習 444
25.1 学習と分類の仕組みの概要 444
25.1.1 学習と分類 444
25.1.2 XML SVM 分類子 445
25.1.3 クラスのハイパープレーンとしきい値 446
25.1.4 分類子の学習用コンテンツ 449
25.2 分類子 API 449
25.2.1 XQuery ビルトイン関数 450
25.2.2 データは任意の場所に格納することも、作成することも可能 450
25.2.3 API は多様なチューニングが可能 450
25.2.4 分類子の supports 形式と weights 形式の比較 451
25.2.5 カーネル（マッピング関数） 451
25.2.6 精度と再現度のバランスを取るしきい値を見つける 452
25.3 分類子による XML の活用 452
25.4 学習用セットの作成 452
25.4.1 学習用セットの重要性 452
25.4.2 学習用セットのラベルの定義 453
25.5 クラスごとのしきい値を判断する方法 454
25.6 例：分類子の学習と実行 455
25.6.1 シェークスピアの戯曲：学習用セット 456
25.6.2 Comedy、Tragedy、History：クラス 456
25.6.3 学習用コンテンツセットを分割する 456
25.6.4 学習用コンテンツの最初の半分に対してラベルを作成する 457
25.6.5 学習用コンテンツの最初の半分に対して cts:train を実行する 457
25.6.6 コンテンツセットの残り半分に対して cts:classify を実行する 458
25.6.7 cts:thresholds を使用して、残り半分のしきい値を計算する 459

25.6.8 結果を評価して、変更を加えたうえで、このプロセスを反復する 460
25.6.9 他のコンテンツに対して分類子を実行する 460

26.0 cts:cluster を使用した結果のクラスタ化 461
26.1 cts:cluster とは 461
26.2 cts:cluster のオプション 462
26.2.1 クラスタ化（cts:cluster）のオプション 462
26.2.2 インデックス付け（db:）オプション 463
26.3 cts:cluster の出力とは 464
26.4 クラスタの HTML レポートを作成する例 465

27.0 MarkLogic サーバーの言語サポート 469
27.1 MarkLogic サーバーでの言語サポートの概要 469
27.2 トークン化とステミング 470
27.2.1 言語固有のトークン化 470
27.2.2 さまざまな言語でのステミング検索 472
27.3 ドキュメントの読み込み時および更新時の言語 473
27.3.1 トークン化とステミング 473
27.3.2 xml:lang 属性 473
27.3.3 ドキュメントの読み込みおよび更新の際の言語に関する注意 475
27.4 言語別のドキュメントのクエリ実行 475
27.4.1 トークン化、ステミング、および xml:lang 属性 476
27.4.2 言語を認識した検索 476
27.4.3 ステミングなし検索 476
27.4.4 不明な言語 478
27.5 サポートされている言語 478
27.6 一般言語のサポート 479

28.0 カスタムトークン化 480
28.1 カスタムのトークナイザーオーバーライドに関する概要 480
28.2 文字の分類のトークン化への影響 481
28.3 xdmp:describe を使用してトークン化を調べる 482
28.4 トークナイザーオーバーライドを使用した場合のパフォーマンスへの影響 483
28.5 カスタムのトークナイザーオーバーライドの定義 483
28.6 カスタムのトークナイザーオーバーライドの例 484
28.6.1 例：フィールドをトークナイザーオーバーライドとともに設定する 484
28.6.2 例：ワイルドカードがオンにされている検索の精度の向上 485
28.6.3 例：データの正規化 487
28.6.4 例：単語内の検索 488
28.6.5 例：symbol 分類の使い方 490

29.0 エンコーディングとコレーション 492
29.1 文字エンコーディング 492
29.2 コレーション 493
29.2.1 コレーションの概要 493
29.2.2 一般的な 2 つのコレーションの URI 494
29.2.3 コレーションの URI のシンタックス 494
29.2.4 3.1 のレンジインデックスとレキシコンとの後方互換性 499
29.2.5 UCA ルートコレーション 499
29.2.6 コレーションのデフォルトの決定方法 500
29.2.7 コレーションの指定 501
29.3 言語ごとのコレーションと文字セット 502

30.0 データ視覚化ウィジェット 508
30.1 視覚化ウィジェットの概要 509
30.2 視覚化ウィジェットの操作 512
30.2.1 Oscars サンプルアプリケーションの構築 512
30.2.2 ウィジェットの全般的な操作 514
30.2.3 折れ線グラフウィジェットの操作 515
30.2.4 棒グラフウィジェットと横棒グラフウィジェットの操作 516
30.2.5 円グラフウィジェットの操作 518
30.2.6 地図ウィジェットの操作 519
30.3 ウィジェットアーキテクチャの概要 522
30.3.1 ウィジェット 523
30.3.2 コントローラ 524
30.3.3 識別 524
30.3.4 ウィジェット内の相互作用 525
30.3.5 シャドウクエリ 525
30.4 HTML ページへの視覚化ウィジェットの追加 526
30.4.1 <head> 要素内の一般的な JavaScript と CSS 526
30.4.2 <body> 要素内の一般的なコード 527
30.4.3 表示を初期化する 528
30.5 例：アプリケーションへのウィジェットの追加 529
30.6 視覚化ウィジェットの制限 535
30.7 プロキシをセットアップする 536
30.8 ウィジェット API リファレンス 536
30.8.1 ML.controller メソッド 537
30.8.2 ML.createWidget（Null Widget）メソッド 539
30.8.3 一般的なウィジェットのメソッド 540
30.8.4 ML.chartWidget メソッド 541
30.8.5 ML.mapWidget メソッド 542
30.8.6 ウィジェットのイベント 545
30.8.7 ウィジェットのクエリ構造 546
30.8.8 ウィジェットのクエリ更新構造 546
30.8.9 検索結果の形式 547
30.8.10ファセット構造 547
30.8.11ウィジェットの内部イベント 548

1.0 MarkLogic サーバーでの検索アプリケーションの開発

この章では、MarkLogic サーバーでの検索アプリケーション開発の概要について説明します。この章は、次のセクションで構成されます。

- MarkLogic サーバーの検索機能の概要
- 検索情報の入手先

1.1 MarkLogic サーバーの検索機能の概要

MarkLogic サーバーには、豊富な全文検索機能が用意されています。すべての検索機能が XQuery で利用できる拡張機能として実装され、ほとんどの検索機能が REST および Java インターフェイスを通じても利用できます。このセクションでは、MarkLogic サーバーの主要な検索機能のうちいくつかの概要について簡単に説明します。このセクションは、次のパートから構成されます。

- ハイパフォーマンスな全文検索
- XQuery、Java、および REST API
- 複数のクエリタイプのサポート
- XQuery での XPath 検索のフルサポート
- レキシコンとレンジインデックスに基づく API
- ステミング（語幹処理）、ワイルドカード、スペリングほか多数の機能
- アラート API とビルトイン

1.1.1 ハイパフォーマンスな全文検索

MarkLogic サーバーは、極めて大規模なデータベース（数百テラバイト以上）まで拡張できるように設計されています。すべての検索機能は、データベースのサイズに関係なくデータベースを直接操作します。ドキュメント読み込みの際に、任意の検索を高速化する全文インデックスが作成されます。検索では、インデックスが自動的に使用されます。`xdmp:estimate` などの XQuery 関数や `unfiltered` 検索オプションなどの機能を使用すると、MarkLogic インデックスから直接結果を返すことができます。

1.1.2 XQuery、Java、および REST API

MarkLogic サーバーは、API のレイヤーを使って検索機能を提供します。次の図に、MarkLogic の search API のレイヤー構造を示します。これらの API には拡張性があり、多数のアプリケーションで利用できます。

Java API	REST API と同じ機能を公開する Java クラスのセット。
REST API	search API および CTS API のほとんどの機能への RESTful な HTTP インターフェイス。
XQuery Search API (`search:*`)	CTS より上位のレイヤーで抽象化を提供する XQuery ライブラリ。複雑な検索アプリケーションの作成をシンプルにします。
XQuery CTS API (`cts:*`)	ビルトインの基礎的な検索操作およびビルトイン XQuery 関数からの生の検索結果へのアクセス。

注: レイヤーによっては、隣接するレイヤーの一部の機能が利用できません。

MarkLogic サーバーのテキスト検索基盤の核となるのは、`cts` 名前空間内のビルトイン XQuery 関数で、全文検索を実行する CTS API です。この基盤より上位にある API は、XQuery、Java、または HTTP リクエスト対応の任意のプログラミング言語で検索アプリケーションを迅速に開発するための、より高次の抽象化を実現しています。例えば、XQuery の Search API は、`cts:search`、`cts:word-query`、`cts:element-value-query` など、CTS の機能を使用して構築されています。

ほとんどのアプリケーションでは、search、REST、または Java API で十分です。CTS API は、アラートアプリケーション用のリバースクエリやコンテンツの分類など、高度なアプリケーション機能に使用してください。こういったより高次の API は、次のような利点を提供します。

- これらをサポートしている制約およびインデックスからのクエリの抽象化
- 検索結果のスニペット作成、強調表示、パフォーマンス分析のビルトインサポート
- 拡張性のあるシンプルな文字列クエリ文法
- 使いやすいクエリ作成用シンタックス
- パフォーマンスを最適化するビルトインのベストプラクティス

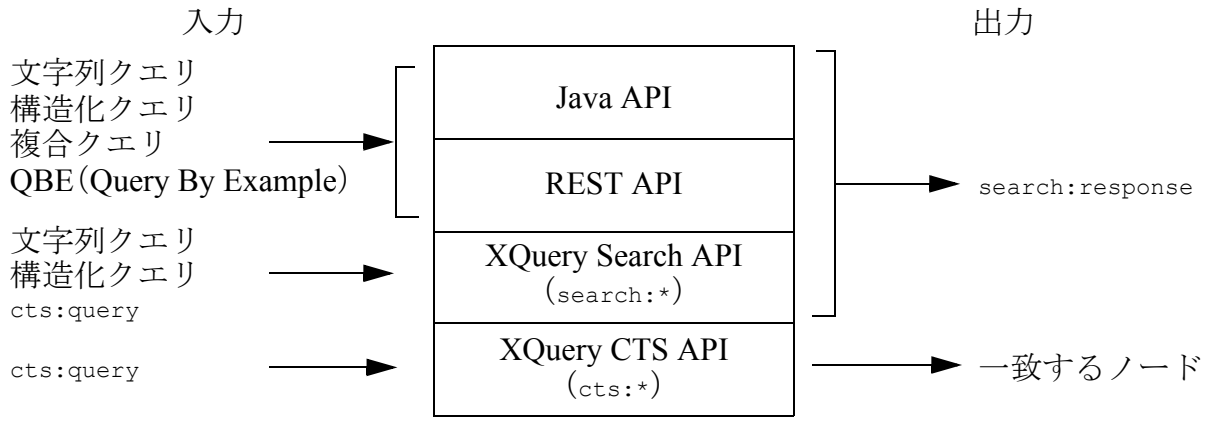
1 つのアプリケーション内で、これら複数の API を使用できます。例えば、Java アプリケーションに XQuery 拡張関数を含めることで、サーバー上での検索結果のカスタム変換を実行できます。同様に、XQuery アプリケーションで `search:*` 関数と `cts:*` 関数の両方を呼び出すことができます。

1.1.3 複数のクエリタイプのサポート

「XQuery、Java、および REST API」（16 ページ）で説明されている各 API は、シンプルな文字列クエリ（`cat OR dog`）から XML または JSON の複雑なクエリまで、コンテンツおよびメタデータの検索用に 1 つ以上の入力クエリタイプをサポートしています。検索結果は生（raw）の結果またはレポート形式で返されます。サポートされているクエリタイプと結果の形式は、API によって異なります。

例えば、CTS API の主要な検索関数である `cts:search` では、きめ細かな検索ができるコンポーザブルクエリである `cts:query` 形式の入力が利用できます。`cts:search` 関数は、一致するノードのシーケンスとして生の結果を返します。Search API、REST API、および Java API では、文字列クエリや構造化クエリなど、より抽象的なタイプのクエリが利用でき、`search:response` XML 要素としてレポート形式の結果を返します。このカスタマイズ可能なレポートには、一致したタームが強調表示されたスニペットや、クエリに関する指標などの詳細を含めることができます。REST API と Java API では、JSON として結果のレポートを返すこともできます。

次の図に、コンテンツおよびメタデータの検索のために各 API が提供するクエリタイプと結果の形式をまとめます。



次の表に、各クエリタイプの簡単な説明を示します。下へ行くほど、クエリ作成が複雑になります。

クエリタイプ	サポートしている API	説明
文字列クエリ	• search • REST • Java	ターム、語句、および AND、OR、NEAR のような演算子から構成されるシンプルな文法を使用して、テキスト文字列としてクエリを作成します。文字列クエリは、エンドユーザーが検索テキストボックスに入力して簡単に作成できます。詳細については、「文字列クエリを使用した検索」（60 ページ）を参照してください。
QBE（Query By Example）	• REST • Java	対象となるドキュメントの構造に類似するシンタックスを使用して、XML または JSON でクエリを作成します。理論的には、QBE は開発者が「類似しているドキュメント」を素早く検索できるようにするためのものです。詳細については、「Query By Example を使用した検索」（181 ページ）を参照してください。
構造化クエリ	• search • REST • Java	抽象構文ツリー（AST）を使用して、JSON または XML でクエリを作成します。ただし、search API ベースの抽象化およびオプションも利用します。元々文字列クエリとして表現されたクエリの調整または追加に便利です。詳細については、「構造化クエリを使用した検索」（74 ページ）を参照してください。
複合クエリ	• REST • Java	XML や JSON の構造を使用して検索します。その際、オプションとして文字列クエリや構造化クエリも使用します。これにより、クエリオプションを事前定義せずに検索できます。これを使用しない場合、REST および Java API では事前定義が必要になります。詳細については、『REST Application Developer's Guide』の「Specifying Dynamic Query Options with Combined Query」または『Java Application Developer's Guide』の「Apply Dynamic Query Options to Document Searches」を参照してください。

クエリタイプ	サポートしている API	説明
`cts:query`	• search • CTS	`cts:and-query` や `cts:not-query` のような低レベルの `cts:query` 要素から、XML でクエリを作成します。この表現は構造化クエリのようなツリー構造ですが、操作するにはずっと複雑になります。詳細については、「cts:query 式の構成」（239 ページ）を参照してください。

1.1.4 XQuery での XPath 検索のフルサポート

MarkLogic サーバーでは XQuery 言語（XPath 2.0 を含む）が使用できます。XPath 式は、データベース全体を検索します。例えば、次のような XPath 式について考えてみましょう。

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

この式は、データベース全体を検索し、式に一致する `my-child` ノードを返します。XPath 式は、データベース内のインデックスをフル活用し、高速になるように設計されています。

1.1.5 レキシコンとレンジインデックスに基づく API

MarkLogic サーバーには、要素、要素属性、XPath 式、JSON キーのような XML 構造にインデックス付けするレンジインデックスがあります。位置情報の値を対象とするレンジインデックスもあります。これらの各レンジインデックスには、レキシコン API が関連付けられています。レキシコン API を使用すると、インデックスから直接値を返すことができます。レキシコンは、ファセットを構築する場合や、要素または属性値の個数を高速に求める場合に非常に役立ちます。search、Java、および REST API では、レキシコン機能をよく使用します。レキシコンの詳細については、「レキシコンを利用した閲覧」（314 ページ）を参照してください。

1.1.6 ステミング（語幹処理）、ワイルドカード、スペリングほか多数の機能

MarkLogic サーバーの検索は、多様な全文検索機能をサポートしています。これらの機能には、ステミング（語幹処理）、ワイルドカード検索、発音符号による区別ありまたは区別なし（ダイアクリティックセンシティブ）の検索、大文字 / 小文字の区別ありまたは区別なし（ケースセンシティブ）の検索、スペリング修正機能、シソーラス機能、位置検索、高度な言語およびコレーションのサポートほか多数の機能があります。これらの機能はすべて、拡張性と柔軟性がある方法で相互に作成および連携できるように設計されています。

1.1.7 アラート API とビルトイン

事前定義されたクエリに一致する新しいコンテンツが入力されたことをユーザーに通知するアプリケーションを作成できます。このようなアプリケーションを作成するのに役立つ API があります。また、大規模で拡張性があるアラートアプリケーションを作成するのに役立つビルトイン `cts:query` コンストラクタ（`cts:reverse-query`）およびインデックス付け機能もあります。アラートアプリケーションの詳細については、「アラートアプリケーションの作成」（367 ページ）を参照してください。

1.2 検索情報の入手先

『MarkLogic XQuery and XSLT Function Reference』には、XQuery 関数のシグネチャと説明、および多数のコード例が記載されています。この『Search Developer's Guide』には、次のような、MarkLogic サーバーの検索機能に関する説明および技術的な詳細が記載されています。

- 「search API：概要と使い方」（22 ページ）
- 「cts:query 式の構成」（239 ページ）
- 「関連度スコア：概要とカスタマイズ」（289 ページ）
- 「レキシコンを利用した閲覧」（314 ページ）
- 「cts:query 式でのレンジクエリの使用」（328 ページ）
- 「検索タームのマッチ箇所の強調表示」（338 ページ）
- 「位置検索アプリケーション」（347 ページ）
- 「エンティティエンリッチメントによるドキュメントのマークアップ」（365 ページ）
- 「アラートアプリケーションの作成」（367 ページ）
- 「fn:count と xdmp:estimate の使い方の比較」（379 ページ）
- 「ステミング検索の概要と使い方」（385 ページ）
- 「ワイルドカード検索の概要と使い方」（405 ページ）
- 「コレクション」（416 ページ）
- 「シソーラス関数の使用」（425 ページ）
- 「スペリング修正関数の使用」（433 ページ）
- 「MarkLogic サーバーの言語サポート」（469 ページ）
- 「エンコーディングとコレーション」（492 ページ）

MarkLogic サーバーでのアプリケーション開発に関する他の情報については、『Application Developer's Guide』を参照してください。MarkLogic サーバーでの XQuery については、『XQuery and XSLT Reference Guide』を参照してください。

2.0 search API：概要と使い方

この章では、search API について説明します。search API は、ファセット、検索結果、スニペットを含んでいる検索アプリケーションを簡単に作成できるように設計された XQuery API です。この章は、次のセクションから構成されます。

- search API の概要
- クエリオプションによる検索の制御
- search:suggest による検索ターム入力支援
- カスタム制約の作成
- 検索文法
- search:values を使用してレキシコン値を返す
- search API での JSON のサポート
- search API のその他の例

この章では、search API の使用の背景、デザインパターン、および例を示します。関数のシグネチャと説明については、『MarkLogic XQuery and XSLT Function Reference』の「XQuery ライブラリモジュール」の検索に関するドキュメントを参照してください。

2.1 search API の概要

search API は、検索機能、検索パーシング機能、検索文法、ファセット化、スニペット作成、検索ターム入力支援、およびその他の検索アプリケーション機能を組み合わせて 1 つの API にした XQuery ライブラリです。「複数のクエリタイプのサポート」（17 ページ）で説明しているように、XQuery、REST、および Java を使って、さまざまなクエリタイプに対して search API を操作できます。

search API を使用することで、基礎となる `cts:search` および `cts:query` API の詳細を理解しなくても、検索アプリケーションを簡単に作成できます。search API は、大規模な実稼働アプリケーション向けに設計されています。

このセクションでは、search API の概要を示し、機能のいくつかについて説明します。このセクションは、次のトピックから構成されています。

- XQuery ライブラリモジュール
- シンプルな search:search の例とレスポンスの出力
- 自動クエリテキストパーシング機能および文法
- 制約付き検索とファセットナビゲーション

- ビルトインのスニペット作成
- 検索ターム入力支援
- オプションおよび拡張による検索のカスタマイズ
- スピードと精度

2.1.1 XQuery ライブラリモジュール

search API は、XQuery ライブラリモジュールとして実装されています。search API は、XQuery から直接使用できます。REST API および Java API を使っても、search API のほとんどの機能にアクセスできます。詳細については、『REST Application Developer's Guide』または『Java Application Developer's Guide』を参照してください。

XQuery から search API を使用するには、次のプロローグステートメントで XQuery モジュールに search API ライブラリモジュールをインポートします。

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

search API は、名前空間プレフィックス `search:` を使用します。この名前空間プレフィックスは、サーバーで事前定義されていません。search API には、検索実行、検索結果表示、スニペット、およびクエリ入力支援を提供するため、`search:search`、`search:snippet`、および `search:suggest` というコア関数があります。これらの操作をより細かく設定したり、便利なツールを提供する関数も他にあります。

search API の関数のシグネチャおよび個々の関数の詳細については、『MarkLogic XQuery and XSLT Function Reference』の search API の項を参照してください。

2.1.2 シンプルな search:search の例とレスポンスの出力

`search:search` 関数は、入力された検索タームをパースして適切な `cts:query` にし、データベース内の一致するノードのスニペットと URL を含むレスポンスを返します。次のような非常にシンプルなクエリで、search API を使ってみましょう。

```
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>
```

出力は search:response 要素で、検索結果ページの構築に必要なすべての内容を含んでいます。これには、検索に一致するドキュメントの総数の推定値、各結果の URI と XPath、検索結果のページネーション、結果コンテンツのスニペット、送信した元のクエリテキスト、レスポンス時間についてのメトリックが含まれます。各 search:result が返すデータは、result-decorator クエリオプションでカスタマイズできます。

手持ちのコンテンツに対して search API を試すには、上の例のようなシンプルな検索を自分のコンテンツデータベースに対して実行し、検索結果を調べます。

search:search 関数は高度にカスタマイズできますが、デフォルトでも多くのアプリケーションで有効な実用的な設定になっています。search:search の結果を利用して、必要に応じてシンプルまたは複雑な、便利な結果ページを簡単に作成できます。

2.1.3 自動クエリテキストパーシング機能および文法

通常の検索アプリケーションでは、ブラウザの検索ボックスにクエリテキストを入力します。このテキストが、*文字列クエリ*です。search API は、効率的で強力な検索を行うために、自動的に文字列クエリをパースして cts:query にします。XQuery、Java、および REST では、次のようなインターフェイスで文字列クエリを使用できます。

- XQuery：search:search および search:parse 関数
- Java：com.marklogic.client.query.QueryManager クラス
- REST:/search サービス

文字列クエリのデフォルトの文法は、Google の文法に似ています。デフォルトの文法は、単語（ターム）と二重引用符で囲んだ語句、論理演算子と関係演算子（AND、OR、LT、GT）、丸括弧（( )）による優先順位付け、マイナス記号（-）による否定、コロン（:）によるユーザー設定の制約をサポートしています。

デフォルトの文法の概要は、次のとおりです。詳細については、「デフォルトの文字列クエリの文法」（61 ページ）を参照してください。

- 単語（ターム）は、単独で利用できます。

```
cat
```

- AND 演算子と OR 演算子。AND の方が、優先されます。
- 括弧は、デフォルトの優先順位よりも優先されます。

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

- 複数の単語（ターム）は、AND で組み合わせられます。

```
cat dog
```

- 語句（フレーズ）は、二重引用符で囲みます。

```
"cat and dog"
```

- 単語（ターム）は、前にマイナスを付けることで結果から除外されます。

```
cat -dog
```

- コロン演算子は、設定済みの制約または演算子の検索を示します（詳細については、「制約オプション」（257 ページ）および「演算子オプション」（268 ページ）を参照してください）。

```
tag:value
```

- 制約および演算子の検索は、語句を対象にできます。

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

- クエリテキストとしては、これらのタイプの検索を任意の順序ならびに任意の数、利用できます。
- デフォルトの検索順序としては、明示的な順序（例えば、括弧付き）が優先され、その後、暗黙的な順序に進みます。したがって、複数ターム（複数の単語）のクエリで明示的な AND 演算子を使用しているものと、暗黙的な AND を使用しているものは、優先順位が違うため、パース方法が異なります。例えば、A OR B AND C は A OR (B AND C) としてパースされ、A OR B C は (A OR B) and C としてパースされます。

文字列クエリのパーシングでは、オプションノードで指定した制約および演算子を検索実行時に適用します。さらに、オプションノードでデフォルトの検索パーシング文法を変更、拡張、および修正できます。デフォルトの文法は非常に強固でフル機能なので、ほとんどのアプリケーションでは、検索文法を修正する必要はありません。デフォルトの文法の修正の詳細については、「文字列クエリの文法の修正と拡張」（66 ページ）を参照してください。search API のオプションノードの詳細については、「クエリオプションによる検索の制御」（30 ページ）を参照してください。

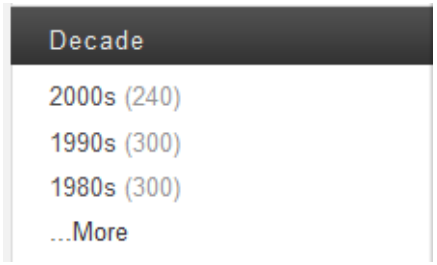
2.1.4 制約付き検索とファセットナビゲーション

search API を使用すると、検索対象をコンテンツの一部に簡単に制約できます。例えば、タイトルに `hello` が含まれるドキュメントだけを返す検索や、特定の 10 年間だけを対象にした検索を作成できます。文字列クエリのデフォルトの文法を使用すると、シンプルなクエリテキスト文字列で、こういった検索を簡単に表現できます。例えば、クエリオプションで制約することができます。次の文字列クエリは特定の 10 年間に検索を制約しています。

```
decade:2000s
```

このような検索は、ファセット作成に役立ちます。ファセットを使用すると、ユーザーは検索条件を絞り込んでドリルダウンできます。通常ファセットは、一致する結果の件数も表示します。ファセットに表示される件数は、search、REST、および Java API で返すことができます。

次に、エンドユーザーアプリケーションでのファセットの例を示します。



ユーザーは、任意のリンクをクリックして、検索の結果を 10 年単位で絞り込むことができます。例えば、一番上のリンクをクリックすると生成されるクエリには、`decade:2000s` という文字列が含まれ、検索をその 10 年間に制約します。

ファセットには、各制約値の件数も含まれます。リンク右の数字は、その 10 年間に限った検索結果の数です。

search API はレスポンスとして、上の例のようなファセット作成に必要なすべての情報を含む XML を返します。REST および Java API は、この情報を XML または JSON として返すことができます。

検索が返すファセットには、ユーザーインターフェイスの生成に必要な件数と値が含まれます。例えば、上のファセットを作成するために、次の XML が使用されています。これは、search API が返したものです。

```
<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>
```

この検索では、レスポンス内の件数と値は、さらに他のアクティブなクエリによってもフィルタリングされ、その検索の件数が表示されます。search、REST、および Java API で構築できる制約とファセットには、多くの種類があります。制約の詳細については、「制約オプション」（257 ページ）を参照してください。

2.1.5 ビルトインのスニペット作成

検索結果ページでは一般的に、一致するドキュメントの一部が表示され、さらに一致部分が強調表示されます。場合によっては、検索一致のコンテキストを説明するテキストも表示されます。これらの検索結果の断片は、*スニペット*と呼ばれます。例えば、MarkLogic サーバー（「MarkLogic Server」）を検索すると、次のようなスニペットが生成されます。

```
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 ...
```

search、REST、および Java API では、`search:response` の出力にスニペットが含まれます。これにより、一致部分をドキュメントのコンテキストの中で表示する検索結果ページを簡単に作成できます。ただし、多くの場合、特定のコンテンツセットにおいてどのようなスニペットが最適なのかは、アプリケーションごとに異なります。このため、search API で、スニペットをカスタマイズできます。その際、ビルトインのスニペット作成アルゴリズムを使用することも、独自のスニペットコードを追加することもできます。検索時のスニペット作成をカスタマイズする方法の詳細については、「結果のスニペットの修正」（272 ページ）を参照してください。

2.1.6 検索ターム入力支援

通常、検索アプリケーションでは、検索ボックスへの入力時に検索タームの候補が表示されます。これらの候補は、データベース内のタームに基づいており、一般的に、ユーザーインターフェイスの対話性を高め、アプリケーションに適した検索タームをすばやく提案するために使用されます。search API の `search:suggest` 関数は、検索入力支援ユーザーインターフェイスにタームを供給するためのものです。検索語入力支援の詳細については、「search:suggest による検索ターム入力支援」（30 ページ）を参照してください。

2.1.7 オプションおよび拡張による検索のカスタマイズ

search、REST、および Java API で、検索を簡単にカスタマイズできます。検索用クエリオプションで直接、広範囲にカスタマイズできます。数多くのオプションがあり、検索のほぼすべての要素を制御できます。

ビルトインのオプションでは必要なことを実行できない場合は、XQuery 拡張メカニズムを利用できます。このメカニズムには、独自の XQuery コードを呼び出すフックが含まれます。フックを使用すると、実装されている関数の場所と名前を指定して、search API のこの関数の実装を置き換えることができます。search API は関数値を使用してカスタム関数をパラメータとして渡すことで、デフォルトの search API の機能を置き換えます。関数値の詳細については、『Application Developer's Guide』の「Function Values」を参照してください。

拡張関数の基本的な指定方法は、`search:options` ノードのさまざまな要素に対して属性 `apply`、`ns`、および `at` を使用するものです。これらはそれぞれ、実装済み関数のローカル名、名前空間、およびコードが含まれる関数ライブラリモジュールの場所に対応します。例えば、次のコードについて考えてみましょう。

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

この例では、transform-results オプションで、search API のデフォルトのスニペット作成関数の代わりに、アプリケーションサーバールートの下にあるライブラリモジュール my-module 内の my-snippet 関数を使用するように指定しています。transform-results の使用方法の詳細については、「結果のスニペットの修正」（272 ページ）を参照してください。

apply 属性を含む検索オプションでは、この拡張パターン（transform-results、複数の grammar オプション、custom 制約など）を使用して、そのオプション機能を独自の実装で置き換えられます。

2.1.8 スピードと精度

search API、およびその上に構築されている REST/Java API は、高速になるように設計されています。検索アプリケーションを作成するときは、スピードと精度のトレードオフを考慮しなくてはなりません。search API のさまざまなオプションで値は、フィルタリングあり / なしの検索、発音符号による区別、大文字 / 小文字の区別などを制御します。これらのオプションは、MarkLogic サーバーでの検索の推定精度に影響します。これらのクエリオプションのデフォルト値は、ほとんどのアプリケーションで使えるように設計されています。ただしアプリケーションはみな異なるので、MarkLogic では、個々のアプリケーションごとに有効な値を制御できるツールを提供しています。

レンジ制約では、レキシコンを使用して、一意の正確な値と件数を高速に取得できます。ただし操作によっては、一部のケースで正確な件数が生成されないことがあります。例えば、cts:query をレキシコン API に渡した場合（場合によっては search API によって実行されます）、フィルタリングされた検索値ではなく cts:query のインデックス解決に基づいて、レキシコンの呼び出しがフィルタリングされますが、インデックス解決がすべてのクエリで正確であることは保証されません。検索のインデックス解決の詳細については、『Query Performance and Tuning Guide』の「Fast Pagination and Unfiltered Searches」を参照してください。

断片化や検索対象（search API オプションの searchable-expression）のような他の要因も検索のインデックス解決が正しいかどうかに影響する可能性があり、同様にレキシコンへのさまざまなオプションも影響する可能性があります。これらのさまざまなオプションのデフォルト値は、多くの検索アプリケーションで使えるようにトレードオフが適用されています。例えば、search:response 出力の total 属性の値は、cts:remainder の結果です。これは、常に高速ですが、すべての検索で正確であることは保証されません。詳細については、「fn:count と xdmp:estimate の使い方の比較」（379 ページ）を参照してください。

2.2 クエリオプションによる検索の制御

XQuery、REST、および Java API による検索操作のほとんどには、クエリのオプションがあります。クエリオプションで、検索の動作と結果形式を指定できます。事前定義されたデフォルトのクエリオプションがあります。デフォルトは、カスタムのクエリオプションを指定して変更できます。例えば、XQuery 関数 `search:search` はパラメータとして `search:options` ノードを取ります。クエリオプションは、XQuery、REST、または Java を使用して XML として表現できます。REST および Java API では、クエリオプションを JSON でも表現できます。

詳細については、「クエリオプションを使用した検索のカスタマイズ」（255 ページ）を参照してください。

2.3 search:suggest による検索ターム入力支援

`search:suggest` 関数は、ワイルドカード文字列に一致する候補を返す関数で、クエリ入力支援アプリケーションで使用されます。search:suggest を使用するアプリケーションの例については、Oscars サンプルアプリケーションを参照してください。このアプリケーションは、『Application Builder Developer's Guide』の「Building the Oscars Sample Application」の説明に従って Application Builder で作成できます。

アプリケーションでの `search:suggest` 関数の典型的な使用方法は、JavaScript イベントでテキストボックス内の変更を listen し、変更があったときに、`search:suggest` 呼び出しを MarkLogic サーバーに非同期で送信するというものです。これにより、文字が入力されるたびに新しい候補がユーザーインターフェイスに表示されます。このセクションの残りの部分では、次のような search:suggest 関数の詳細について説明します。

- default-suggestion-source オプション
- suggestion-source オプションを使用した候補の選択
- search:suggest への複数のクエリテキスト入力の使用
- カーソル位置に基づいた候補の作成
- search:suggest の例

REST および Java API でのこの関数の使用方法については、次を参照してください。

- 『REST Application Developer's Guide』の「Generating Search Term Completion Suggestions」
- 『Java Application Developer's Guide』の「Generating Search Term Completion Suggestions」

2.3.1 default-suggestion-source オプション

search:suggest を使用する場合は、default-suggestion-source を指定するのがベストです。search API は、default-suggestion-source を使用して、検索タームの候補を探します。default-suggestion-source が指定されていない場合、search:suggest のすべての呼び出しにおいて制約と演算子の候補のみを返し、候補がない場合は空のシーケンスを返します。search:suggest 関数は、クエリテキスト文字列と一致する制約と演算子の候補を提案します。また、レンジインデックスベースの制約の場合は、一致する制約値の候補を提案します。default-suggestion-source オプションのシンタックスの詳細については、『MarkLogic XQuery and XSLT Function Reference』の search:search オプションの項を参照してください。

特に大規模データベースで最高のパフォーマンスを実現するには、default-suggestion-source を range または collection と一緒に使用し、word レキシコンとは一緒に使用しないようにします。

次の default-suggestion-source の例では、my-attribute という属性の文字列レンジインデックスを候補語のソースとして使用します。通常、レンジ候補ソースが最も優れており、特に大規模データではそうです。このレンジインデックスは存在する必要があります。存在しない場合、検索実行時に例外がスローされます。

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

次の例では、フィールドレキシコンを使用して検索タームの候補を探すように指定しています。フィールドは、特にそのフィールドがデータベース全体の比較的小さなサブセットである場合、候補のソースとして適しています。指定したフィールドのフィールドワードレキシコンは存在する必要があります。存在しない場合、検索実行時に例外がスローされます。

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

2.3.2 suggestion-source オプションを使用した候補の選択

アプリケーションによっては、ある制約に基づく候補だけを使用したい場合があります。例えば、一意の値が数百万個ある name に制約を課し、そのうち特定の 500 のみを候補として表示したい場合などです。この場合、suggestion-source オプションを指定することで、search:suggest がその制約においてクエリテキストが返す候補よりも優先させることができます。

suggestion-source 要素の name 属性に、上書きする制約を指定します。例えば次のオプションでは、name 制約における候補として、name 要素の代わりに short-list-name 要素の値を使用するよう指定しています。

```
<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>
```

検索およびファセットに使用する名前付きの制約があり、入力候補表示用に若干（あるいは完全に）異なるソースを使用し、かつ検索語をパースし直す必要がない場合は、suggestion-source オプションを使用します。

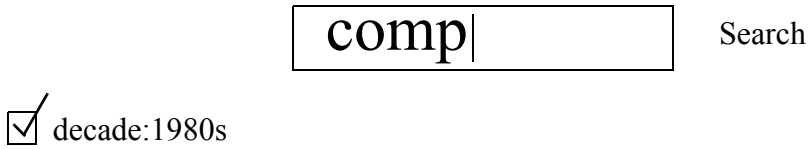
ある制約において候補を表示させないようにするには、その制約に空の suggestion-source を追加します。

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

2.3.3 search:suggest への複数のクエリテキスト入力の使用

search:suggest には、1 つまたは複数のクエリテキストパラメータを指定できます。search:search で複数のクエリテキストのシーケンスを指定する場合は、最初の項目（または $focus パラメータに対応する項目）で、候補ソースとマッチするテキストを指定します。このシーケンス内のその他の各項目は、cts:query としてパースされ、そのクエリはテキストマッチングクエリのテキストにおける検索候補の制約に使用されます。これは、複数のクエリテキストを cts:and-query と組み合わせる他の search API 関数とは違うので注意してください。

次のようなユーザーインターフェイスについて考えてみましょう。



上の検索テキストボックスにユーザーがテキストを入力します。下のチェックボックスでは、10 年単位の指定ができます。decade:1980s のようなクエリテキストは、ユーザーインターフェイスコントロール（例えばファセット）によって表示させることができます。ここで、このユーザーインターフェイスからの search:suggest 呼び出しを構築できます。この呼び出しでは、指定された候補ソースにおける comp にマッチするタームへの制約として decade:1980s というテキストを使用します。次に、この例から生成されうる search:suggest 呼び出しを示します。

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

これで、search:parse("decade:1980s") に一致するフラグメントのうち comp* に一致する候補が返されます。例えば、competent、component、および computer というような単語が含まれるシーケンスが返されます。

2.3.4 カーソル位置に基づいた候補の作成

search:suggest 関数は、カーソル位置に基づいた検索候補を作成します。カーソル位置は、$cursor-position パラメータで指定します。これは、カーソル位置が変更された場合、ユーザーが現在テキストを入力している場所に応じてタームを提案したほうがいいからです。

2.3.5 search:suggest の例

次に、いくつかの search:suggest クエリの例と出力のサンプルを示します。

次の例では、filesize という制約があると仮定します。

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

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

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

次の例では、バケット化された range 制約で search:suggest がどのように機能するかを示します。

```
(: 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 カスタム制約の作成

デフォルトでは、search API は多くのタイプの制約（ただしすべてではありません）をサポートしています。search API で事前定義されていない制約を作成する場合、search API を拡張して独自の制約タイプを使用できます。このタイプの制約は、custom 制約と呼ばれます。この場合、カスタムパーシングの実装とカスタムファセット生成のための XQuery 関数を記述する必要があります。次のように、オプションの XML で関数の実装を指定します。

```
<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>
```

実装する必要がある 3 つの関数は、parse、start-facet、および finish-facet です。apply 属性は関数のローカル名を、ns 属性は名前空間を、at 属性は関数を含むモジュールの場所を指定します。このセクションでは、カスタム制約の作成方法について説明し、カスタムの位置情報制約を作成するためのサンプルコードを記載します。このセクションは、次のパートから構成されます。

- parse 関数の実装
- start-facet 関数の実装
- finish-facet 関数の実装
- 例：シンプルなカスタム制約の作成
- 例：構造化クエリ用のカスタム制約の作成
- 例：カスタム制約位置情報ファセットの作成

2.4.1 parse 関数の実装

parse 関数の目的は、カスタム制約をパースし、クエリテキストから正しい cts:query を生成することです。

このセクションでは、次のトピックについて扱います。

- パーサーのインターフェイスの選択
- 文字列クエリの parse 関数の実装
- 構造化クエリの parse 関数の実装
- 複数形式の parse 関数の実装

2.4.1.1 パーサーのインターフェイスの選択

制約パーシング関数のシグネチャは、クエリ入力のタイプ（文字列クエリまたは構造化クエリ）およびクエリの作成に使用する API によって異なります。

cts:search または search:search の呼び出しなどにより、XQuery から開始されたクエリで制約が使用されうる場合は、次のいずれかを選択してください。

- 入力が常に文字列クエリである場合は、「文字列クエリの parse 関数の実装」（36 ページ）を参照してください。
- 入力が常に構造化クエリである場合は、「構造化クエリの parse 関数の実装」（37 ページ）を参照してください。
- 入力が文字列クエリまたは構造化クエリのどちらでもありうる場合は、「複数形式の parse 関数の実装」（37 ページ）を参照してください。

制約が、REST、Java、または Node.js クライアント API を通じて開始されたクエリでのみ使用される（XQuery を使用することはない）場合は、構造化クエリパースインターフェイスで、文字列クエリと構造化クエリの両方に対応できます。クエリは、必要に応じて内部で変換されます。上述のような XQuery における選択方法は、REST および Java API でも同様です。

2.4.1.2 文字列クエリの parse 関数の実装

文字列クエリでカスタム制約をパースする場合、実装するカスタム関数は、次のシグネチャと互換性がある必要があります。

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

関数には任意の名前空間とローカル名を使用できますが、パラメータの数と順序ならびに戻り値の型に互換性がある必要があります。

`$constraint-qtext` パラメータは制約名であり、またここで、この制約に関連するクエリテキストに結合子を付けます。例えば、制約名が `geo` で、結合子がデフォルトの場合、`$constraint-qtext` の値は、`geo:` になります。`$constraint-qtext` 値は、`qtextconst` 属性で使用されます。この属性は、`search:unparse` で注釈付きの `cts:query` からクエリテキストを作成し直すために必要です。

`$right` パラメータには、`cts:query` としてパースされる制約の値が含まれています。言い換えると、それは、クエリテキストで `$constraint-qtext` に渡されるものの右にあるテキストであり、そのテキストが search API によって `cts:query` としてパースされ、`cts:query` の XML 表現として parse 関数に返されます。`$right` の値は、parse 関数によってカスタム `cts:query` クエリの生成に使用されます。`cts:query` コンストラクタの機能の詳細については、「cts:query 式の構成」（239 ページ）を参照してください。

実装する `parse` 関数は、`$right` パラメータから `cts:query` を取得し、適合して見えるようにパースしてから、`cts:query` XML 要素を返します。例えば、`$right` の値が次のようになっているとします。

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

必要な cts:query クエリを構築するように cts:text 要素を処理するコードを使用します。例えば、cts:text 要素の @ 文字でトークン化し、各値を使用してクエリの一部を作成できます。cts:query 作成の一環として、生成する cts:query に cts:annotation 要素および注釈属性を追加することもできます。これらの注釈を使用すると、search API で、cts:query を逆パーシングして元の形式に戻すことができます。適切な注釈が追加されていない場合、search:unparse が元のクエリテキストを返さないことがあります。これと同様な関数のサンプルについては、「例：カスタム制約位置情報ファセットの作成」（43 ページ）を参照してください。

2.4.1.3 構造化クエリの parse 関数の実装

構造化クエリでカスタム制約を使用するには、カスタム parse 関数は、次と互換性があるシグネチャを持っている必要があります。

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

関数には任意の名前空間とローカル名を使用できますが、パラメータの数と順序ならびに戻り値の型に互換性がある必要があります。例の全体については、「例：構造化クエリ用のカスタム制約の作成」（41 ページ）を参照してください。

$query-elem パラメータは、制約を参照する custom-constraint-query 構造化クエリです。詳細については、「custom-constraint-query」（166 ページ）を参照してください。

2.4.1.4 複数形式の parse 関数の実装

文字列クエリと構造化クエリのどちらの入力も処理できる parse 関数を作成できます。そのためには、両方に対応するように parse 関数インターフェイスを一般化し、XQuery の instance of 演算子でクエリタイプを特定します。

次の parse 関数のスケルトンは、入力クエリを item() として一般化し、2 番目のパラメータ（cts:query または search:options のどちらか）を element() に一般化します。その後、instance of で入力クエリの実際のタイプを検出します。

```
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 :)
};
```

入力クエリタイプを特定したら、2 番目のパラメータを正しいタイプに強制的に設定し、「文字列クエリの parse 関数の実装」（36 ページ）および「構造化クエリの parse 関数の実装」（37 ページ）にあるように、適切な文字列クエリや構造化クエリの parse 関数と同様にクエリをパースします。

2.4.2 start-facet 関数の実装

start-facet 関数は、ファセットで使用する値と件数を返すレキシコン API の呼び出しを唯一の目的としています。レキシコンの詳細については、「レキシコンを利用した閲覧」（314 ページ）を参照してください。実装するカスタム関数は、次のシグネチャと互換性がある必要があります。

```
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()*
```

関数には任意の名前空間とローカル名を使用できますが、パラメータの数と順序ならびに戻り値の型にも互換性がある必要があります。

各パラメータは、search API によって関数に渡されます。$query パラメータには、ユーザーの parse 関数が実装するあらゆるカスタムクエリが含まれ、search API が生成するその他のクエリ（additional-query など、元の検索に渡される他のオプションによって異なります）と組み合わされます。他のすべてのパラメータは、search API の呼び出しに渡される search:options XML ノードで指定されます。カスタムアクションに応じて、それらを使用するかどうかを選択できます。

start-facet 関数でレキシコンの呼び出しを実装するときは、$facet-options パラメータに "concurrent" オプションを追加し、組み合わされたシーケンスをレキシコン API の $options パラメータの入力とする必要があります。"concurrent" オプションは同時実行を利用するもので、特に多くのファセットを使用するアプリケーションでスピードが大幅に向上します。関数のサンプルについては、「例：カスタム制約位置情報ファセットの作成」（43 ページ）を参照してください。

注： start-facet 関数はオプションですが、何らかの MarkLogic サーバーレキシコン関数を使用するカスタムファセットを作成する場合は使った方がいいでしょう。start-facet 関数を使用しない場合、finish-facet 関数で、ファセット構築の全作業（ファセット用の値の構築を含む）を実行する必要があります。レキシコン関数の詳細については、『MarkLogic XQuery and XSLT Function Reference』および「レキシコンを利用した閲覧」（314 ページ）を参照してください。

2.4.3 finish-facet 関数の実装

finish-facet 関数は、start-facet 関数（使用している場合）から入力を取得し、facet 要素を構築します。この関数は、次のシグネチャと互換性がある必要があります。

```
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)
```

関数には任意の名前空間とローカル名を使用できますが、パラメータの数と順序ならびに戻り値の型にも互換性がある必要があります。

パラメータは、search API によって関数に渡されます。$query パラメータには、ユーザーの parse 関数が実装するあらゆるカスタムクエリが含まれ、search API が生成するその他のクエリ（additional-query など、元の検索に渡される他のオプションによって異なります）と組み合わされます。残りのすべてのパラメータは、search API の呼び出しに渡される search:options XML で指定されます。カスタムアクションに応じて、それらを使用するかどうかを選択できます。関数のサンプルについては、「例：カスタム制約位置情報ファセットの作成」（43 ページ）を参照してください。

start-facet 関数を使用しなかった場合、$start パラメータに対して空のシーケンスが渡されます。start-facet 関数を使用していない場合、finish-facet 関数が、ファセットで使用される値と件数の構築、およびファセット XML の作成を行います。

2.4.4 例：シンプルなカスタム制約の作成

次に、文字列クエリで使用する非常にシンプルなカスタム制約を実装するライブラリモジュールを示します。この制約は、制約で指定する値に対する `cts:directory-query` を追加します。この制約にはファセットがないので、`start-facet` 関数および `finish-facet` 関数は必要ありません。このコードは最小限のパースを実行します。実際のコードは、これよりも丁寧に `$right` クエリをパースします。

```
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()}
} ;
```

このモジュールをアプリケーションサーバーのルートにある my-module.xqy というファイルに置いた場合、次のオプションノードでこの制約を実行できます。

```
<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>
```

次のクエリテキストは、この検索を /mydocs/book-dir/ ディレクトリに制約します。

```
part:book
```

2.4.5 例：構造化クエリ用のカスタム制約の作成

次に、構造化クエリ用の非常にシンプルなカスタム制約を実装するライブラリモジュールを示します。この制約は、制約で指定する値に対する cts:directory-query を追加します。この制約にはファセットがないので、start-facet 関数および finish-facet 関数は必要ありません。

```
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()}
} ;
```

このモジュールをアプリケーションサーバーのルートにある `my-module.xqy` というファイルに置いた場合は、次のオプションノードでこの制約を実行できます。

```
<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>
```

次の構造化クエリは、この検索を `/mydocs/book-dir/` ディレクトリに制約します。

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

`return-query` クエリオプションを使用して、カスタムクエリによって生成された `directory-query` を表示できます。例えば、次をオプションノードに追加するとします。

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

この場合、検索のレスポンスには次のようなクエリが含まれます。

```
<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 例：カスタム制約位置情報ファセットの作成

次に、カスタム制約を使用する位置情報ファセットを実装するライブラリモジュールを示します。このモジュールは、@ 文字で制約値をトークン化して、位置情報レキシコンファセット関数への入力を生成します。この例は、デザイン手法の紹介を意図として単純化されており、実稼働向けではありません。このため、ユーザー入力をより確実にするためのエラーチェックは一切実行しません。

注： この例のコードは使用可能ですが、カスタム制約を作成する際のデザイン手法の例にすぎません。位置情報制約を使用する場合は、「制約オプション」（257 ページ）の説明に従って、ビルトインの位置情報制約タイプ（geo-attr-pair、geo-elem-pair、geo-elem）を使用してください。

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

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

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

     <quake>
```

```
      <area>0</area>
      <perimeter>0</perimeter>
      <quakesx020>2</quakesx020>
      <quakesx0201>26024</quakesx0201>
      <catalog_sr>PDE</catalog_sr>
      <year>1994</year>
      <month>6</month>
      <day>11</day>
      <origin_tim>164453.48</origin_tim>
      <lat>61.61</lat>
      <lon>168.28</lon>
      <depth>9</depth>
      <magnitude>4.3</magnitude>
      <mag_scale>mb</mag_scale>
      <mag_source/>
      <dt>1994-06-11T16:44:53.48Z</dt>
    </quake>
:)

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

```
};

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

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

上のカスタムコードを参照するカスタム制約を実行するには、アプリケーションサーバーのルートにある `geoexample.xqy` というファイルに上のモジュールを置き、次のコードを実行します。

```
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 検索文法

XQuery Search API、REST、Java API ではビルトインの文法を使用して、シンプルなクエリテキストから検索クエリを生成します。このクエリテキストは通常、エンドユーザーが単純な HTML フォームに入力したものです。デフォルトの文法には、複雑なクエリを生成するための堅牢な機能が用意されています。次に、デフォルトの文法を使用するいくつかのクエリの例を示します。

- `(cat OR dog) NEAR vet`

  `vet` というタームから 10 ターム以内（`cts:near-query` のデフォルトの距離）に `cat` または `dog` が少なくとも 1 つある

- `dog NEAR/30 vet`

  `vet` という単語から 30 ターム以内にある `dog`

- `cat -dog`

`dog` という単語がない場所での、`cat` という単語

`grammar` クエリオプションを使用して、文字列クエリの文法をカスタマイズできます。

詳細については、「文字列クエリを使用した検索」（60 ページ）を参照してください。

2.6 search:values を使用してレキシコン値を返す

レキシコンは単語または値のリストであり、その対象はデータベース全体（単語のみ）あるいは名前付きの要素、属性、またはフィールド（単語または値）です。`search:values` search API 関数は、レキシコンから値を返します。オプションとして、構造化クエリによる値の制約、値が一致するサブセットの選択、レキシコンの値に基づく集計、複数レキシコン内の値の共起の検出ができます。

レキシコンの概要については、「レキシコンを利用した閲覧」（314 ページ）を参照してください。このセクションでは、search API 固有の次のトピックについて扱います。

- 入力レキシコンの指定
- 結果の制約およびフィルタリング
- 例：結果を制約するためのクエリの使用
- 例：開始値、制限、ページ長を使用したフィルタリング
- 例：値の共起の検出
- 追加のインターフェイス

2.6.1 入力レキシコンの指定

基本的な search:values 呼び出しは、ほとんどの場合、次のような形式になります。

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

$spec-name は、2 番目のパラメータとして渡される search:options で定義された values または tuples 指定の名前です。1 つのレキシコン内の値を操作するには、values 指定を使用します。複数のレキシコン内の値の共起を操作するには、tuples 指定を使用します。

要素、属性、またはフィールド内の値または単語に対するクエリを実行する前に、Admin 画面または Admin API を使用して、対応するレンジインデックスまたはワードレキシコンを定義します。URI またはコレクションのレキシコンのクエリを実行する前には、Admin 画面または Admin API を使用してそのレキシコンを有効にする必要があります。詳細については、「レキシコンの作成」（315 ページ）を参照してください。

次の例は、<first-name/> 要素を対象とする要素レンジインデックスが存在することを前提として、この要素のすべての値を返します。<first-name/> の要素レンジインデックスも存在する必要があります。

```
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>
```

値およびタプル指定のその他の例については、search:values の API リファレンスを参照してください。

search:values 関数では、結果を制約およびフィルタリングするための追加のパラメータを使用できます。詳細については、「結果の制約およびフィルタリング」（49 ページ）を参照してください。また検索オプション内で集計を定義することによって、事前定義またはユーザー定義の集計関数を値またはタプルに適用できます。詳細については、「集計関数の使用」（332 ページ）を参照してください。

2.6.2 結果の制約およびフィルタリング

search:values 関数には、次のようなインターフェイスがあります。$spec-name と $options のみが必須です。

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

search:values によって返される結果をフィルタリングするには、次の表で説明するように、$query、$limit、$start、$page-start、および $page-length パラメータを使用します。

パラメータ	説明
$query	結果を、指定したクエリに一致するドキュメント内の値に制限します。デフォルト：なし。すべてのドキュメントからの値を返します。
$limit	レキシコンから取得する値の最大個数。デフォルト：制限なし。レキシコン内のすべての値、または $query によって選択されたサブセット内のすべての値を返します。
$start	最初に返す値。この値がレキシコン内にない場合、値は次の論理値から返されます。デフォルト：レキシコン内の最初の値、または $query によって選択されたサブセット内の最初の値。
$page-start $page-length	アプリケーションに返す結果のサブセットを定義します。デフォルト：$query、$limit、および $start によって選択されたすべての値を返します。

$query、$limit、および $start パラメータは、レキシコンから選択される値を制限します。$page-start パラメータと $page-length パラメータは選択された値のサブセットを取得するパラメータで、連続する呼び出しで選択された値の「ページング」に使用できます。

$page-start および $page-length を使用して、$limit や $start が選択したサブセット以外の値を取得することはできません。例えば、$page-start + $page-length が $limit を超えている場合、($limit - $page-start + 1) 個の値のみが返されます。

ほとんどのフィルタリングパラメータは相互に独立して使用できます。つまり、クエリなしで制限を指定したり、制限指定なしで開始値を指定したりできます。ただし、`$page-start` を指定する場合は、`$page-length` も指定する必要があります。

2.6.3 例：結果を制約するためのクエリの使用

動物に関する一連のドキュメントについて考えてみましょう。各ドキュメントには、動物の名前と種類が含まれます。例えば、各ドキュメントが次の形式になっているとします。

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

`/animal/name` に要素またはフィールドのレンジインデックスが定義されている場合、次のクエリは結果としてデータベース内のすべての動物名を返します。

```
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>
```

次の例は、クエリ “mammal OR marsupial”（哺乳類 OR 有袋目）に一致するドキュメント内の値に結果を制限し、`duck`、`emu`、および他の “bird” の値を結果から除外します。この例では、`search:parse` を呼び出すことにより、文字列クエリから生成した構造化クエリを使用しますが、任意の構造化クエリも使用できます。

```
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>
```

$limit などの他のフィルタリングパラメータを含む場合、これはクエリの後で適用されます。例えば、4 という制限を追加した場合、上の結果から値セット [aardvark badger camel fox] が返されます。

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

2.6.4 例：開始値、制限、ページ長を使用したフィルタリング

レキシコンにアルファベットの各小文字の文字列値が含まれていて、次のクエリが a,b,c...,z という値の結果を返すと仮定します。

```
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)
```

次のクエリは、上のクエリに対して、個数制限 10、開始値「c」、開始ページ 4、およびページ長 3 を指定します。

```
search:values("alphabet", $options, 10, "c", 4, 3)
```

$limit および $start の値の結果は、レキシコンから取得された「c」から始まる 10 個の値のサブセットになります。下の例では、角括弧（[ ]）を使用して、選択されたサブセットを区切っています。

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

次に、$page-start と $page-length で、search:values によって返される最終的な値の “ページ” を定義します。$limit と $start によって定義されるサブセット内の 4 番目の値は「f」なので、最終的な結果のサブセットには f..h の値が含まれています。下の例では、波括弧（{ }）を使用して、選択された値のページを区切ります。

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

$page-start と $page-length は、$limit が定義する値のサブセットの最後の値を越えた結果を生成することはありません。このため上の例では、$start または $limit を変更しないと、「I」を以降の値を返すことはできません。

下の表に、$start、$limit、$page-start、および $page-length パラメータのさまざまな組み合わせを適用したときに返される値、および search:values でどのようにして最終結果が得られるのかを示します。前述のように、角括弧（[ ]）は、$limit や $start によって選択された値を区切り、波括弧（{ }）は、$page-start と $page-length によって選択された値を区切ります。

フィルタリングパラメータ	戻される値	結果の生成方法
$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

フィルタリングパラメータ	戻される値	結果の生成方法
$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

クエリパラメータがある場合、クエリ適用後の結果に上記のフィルタリングが適用されます。

2.6.5 例：値の共起の検出

次に、`hello AND goodbye` の URI レキシコン、要素、クエリの制約から共起（タプル）を返し、レンジインデックスのデータだけを取り出す方法を示します。

```
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 追加のインターフェイス

次のインターフェイスを使用してレキシコンのクエリを実行することもできます。

- `cts:values` XQuery 関数。詳細については、「レキシコンを利用した閲覧」（314 ページ）を参照してください。
- REST API メソッド `GET:/v1/values/{name}` および `POST:/v1/values/{name}`。詳細については、『REST Application Developer's Guide』の「Querying the Values in a Lexicon or Range Index」および「Finding Value Co-Occurrences in Lexicons」を参照してください。
- Java API の `ValuesDefinition` インターフェイス。詳細については、Javadoc および『Java Application Developer's Guide』の「Search On Tuples (Tuples Query / Values Query)」を参照してください。

2.7 search API での JSON のサポート

search API のオプションノードを使用すると、JSON ドキュメントをデータベースに読み込むときに JSON キーを指定できます。検索している値は JSON キーに関連付けられます。次のオプションノードは、`json-key` の指定例です。

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

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

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

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

<!-- default term with word -->
    <term apply="term">
```

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

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

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

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

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

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

2.8 search API のその他の例

このセクションでは、search API を使用する次のような例を示します。

- バケットの例
- 計算されるバケットの例
- ソート順序の例

2.8.1 バケットの例

Oscars サンプルアプリケーションからの次の例は、10 年間単位のバケットを定義する検索の作成方法を示しています。これらのバケットはファセットの生成に使用され、検索文法で制約として使用されます。バケットはレンジ制約の一種です。詳細については、「制約オプション」（257 ページ）を参照してください。

この例では、Wikipedia の `nominee/@year` 属性に対して `xs:gYear` タイプのレンジインデックスを使用する制約を定義します。

```
xquery version "1.0-ml";

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

let $options :=
<search:options>
  <search:constraint name="decade">
    <search:range type="xs:gYear" facet="true">
      <search:bucket ge="2000" name="2000s">2000s</search:bucket>
      <search:bucket lt="2000" ge="1990"
        name="1990s">1990s</search:bucket>
      <search:bucket lt="1990" ge="1980"
        name="1980s">1980s</search:bucket>
      <search:bucket lt="1980" ge="1970"
         name="1970s">1970s</search:bucket>
      <search:bucket lt="1970" ge="1960"
         name="1960s">1960s</search:bucket>
      <search:bucket lt="1960" ge="1950"
         name="1950s">1950s</search:bucket>
      <search:bucket lt="1950" ge="1940"
         name="1940s">1940s</search:bucket>
      <search:bucket lt="1940" ge="1930"
         name="1930s">1930s</search:bucket>
      <search:bucket lt="1930" ge="1920"
         name="1920s">1920s</search:bucket>
```

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

次に、このクエリからのレスポンスの一部を示します。

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

2.8.2 計算されるバケットの例

computed-bucket レンジ制約は、xs:date および xs:dateTime レンジインデックスを対象にしています。この制約は、現在時刻での計算に基づいて実行時に計算されるバケットの境界を指定します。computed-bucket 要素の anchor 属性には、次の値が指定できます。

<computed-bucket anchor="value">	説明
anchor="now"	現在の時刻。
anchor="start-of-day"	現在の日の開始時刻。
anchor="start-of-month"	現在の月の開始時刻。
anchor="start-of-year"	現在の年の開始時刻。

これらの値は、computed-bucket 要素の ge-anchor 属性および le-anchor 属性でも使用できます。

次の検索では計算されたバケットを指定し、今日更新されたすべてのドキュメントを見つけます（この例では、maintain last-modified プロパティがデータデーベースで設定されていると仮定しています)。

```
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>)
```

anchor 属性に start-of-day の値が指定されているので、ge および lt 属性で指定されている期間の値は、現在の日の開始時に適用されます。start-of-day 値は、1 日の始まりとして深夜 12 時を使用するので、“ 過去 24 時間 ” と同じではありません。相対的なバケット（computed-bucket）と絶対的なバケット（bucket）の違いは、「日、月、年に対する相対的な時間」と「現在時刻に対する相対的な時間」の違いです。絶対的なバケットを使用する例については、「バケットの例」（56 ページ）を参照してください。

2.8.3 ソート順序の例

次の検索では、カスタムソート順序を指定します。

```
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)
```

この検索では、年を条件としたソートを指定しています。オプション指定では、year または relevance を指定できます。指定しなかった場合、スコアでソートされます（この例では、これは relevance と同じです）。

3.0 文字列クエリを使用した検索

この章では、search API でシンプルな文字列クエリを使用して検索を実行する方法について説明します。この章は、次のセクションから構成されます。

- 文字列クエリの概要
- デフォルトの文字列クエリの文法
- 文字列クエリの文法の修正と拡張

この章では、文字列クエリの使用の背景、デザインパターン、使用例を示します。関数のシグネチャと説明については、『MarkLogic XQuery and XSLT Function Reference』の「XQuery Library Modules」の検索に関するドキュメントを参照してください。

3.1 文字列クエリの概要

*文字列クエリ*は、ターム、語句、および演算子から構成されるプレーンテキストの検索文字列であり、エンドユーザーがアプリケーションの検索ボックスに入力することによって簡単に構成できます。例えば、「cat AND dog」は、「cat」というタームと「dog」というタームの両方が含まれているドキュメントを検索するための文字列クエリです。

文字列クエリのシンタックスは、設定可能な文法によって決まります。強力なデフォルトの文法が事前定義されています。文法は `grammar` 検索オプションを使用して修正または拡張できます。詳細については、「デフォルトの文字列クエリの文法」（61 ページ）および「文字列クエリの文法の修正と拡張」（66 ページ）を参照してください。

デフォルトの文法には、複雑なクエリを生成するための堅牢な機能が用意されています。次に、デフォルトの文法を使用するいくつかのクエリの例を示します。

- `(cat OR dog) NEAR vet`

  `vet` というタームから 10 ターム以内（`cts:near-query` のデフォルトの距離）に `cat` または `dog` が少なくとも 1 つある

- `dog NEAR/30 vet`

  `vet` という単語から 30 ターム以内にある `dog`

- `cat -dog`

`dog` という単語がない場所での、`cat` という単語

文字列クエリを使用すると、次の MarkLogic サーバー API でコンテンツおよびメタデータを検索できます。

- XQuery Search API。詳細については、「`search:search`」および「`search:parse`」を参照してください。

- Java API。詳細については、『Java Application Developer's Guide』の「Searching」を参照してください。
- REST API。詳細については、『REST Application Developer's Guide』の「Using and Configuring Query Features」を参照してください。

3.2 デフォルトの文字列クエリの文法

search API には、"cat AND dog" のような文字列クエリを解釈するための、ビルトインのデフォルトの文法があります。デフォルトの文法を使用すると、シンプルな検索文字列を基にして、データベースに対して複雑なクエリを実行するアプリケーションを作成できます。デフォルトの文法を修正したり、カスタム文法を定義したりすることもできます。詳細については、「文字列クエリの文法の修正と拡張」（66 ページ）を参照してください。

- クエリのコンポーネントと演算子
- 演算子の優先順位
- 制約での関係演算子の使用
- 文字列クエリの例

3.2.1 クエリのコンポーネントと演算子

デフォルトの検索文法で文字列クエリを作成するには、以下のコンポーネントおよび演算子を使用します。

クエリ	例	説明
任意のターム	`dog` `dog cat`	`cts:and-query` の場合と同様、1 つあるいは複数のタームをマッチングします。隣にあるタームおよび語句は、暗黙的に `AND` で結合されます。例えば、`dog cat` は `dog AND cat` と同じです。

`" "`	`"dog tail"`
	`"dog tail" "cat whisker"`
	`dog "cat whisker"`
		二重引用符で囲まれたタームは、語句として扱われます。隣にあるタームおよび語句は、暗黙的に `AND` で結合されます。例えば、`dog "cat whisker"` は、`dog` というタームと `cat whisker` という語句の両方を含んでいるドキュメントとマッチされます。

クエリ	例	説明
( )	`(cat OR dog) zebra`	丸括弧はグループ化を示します。この例は、cat または dog というタームを少なくとも 1 つ含んでいて、かつ zebra というタームも含んでいるドキュメントとマッチされます。

-query	`-dog`
	`-(dog OR cat)`
	`cat -dog`
		cts:not-query の場合と同様、NOT 演算です。例えば、cat -dog は、cat というタームを含んでいて、dog というタームを含んでいないドキュメントとマッチされます。

query1 AND *query2*	`dog AND cat` `(cat OR dog) AND zebra`	cts:and-query の場合と同様、2 つのクエリ式とマッチされます。例えば、dog AND cat は、dog というタームと cat というタームの両方を含んでいるドキュメントとマッチされます。AND は、タームおよび語句を組み合わせるためのデフォルトの方法なので、この例は dog cat と等価になります。
query1 OR *query2*	`dog OR cat`	cts:or-query の場合と同様、2 つのクエリのいずれかとマッチされます。この例は、cat というタームまたは dog というタームを少なくとも 1 つを含んでいるドキュメントとマッチされます。
query1 NOT_IN *query2*	`dog NOT_IN "dog house"`	cts:not-in-query の場合と同様、後者のマッチに合致しない前者のクエリとマッチされます。この例は、dog house という語句の一部ではない dog にマッチされます。

クエリ	例	説明
query1 NEAR *query2*	dog NEAR cat (cat food) NEAR mouse	cts:near-query の場合と同様、NEAR 演算子の両側にあるクエリとマッチし、それらが互いに 10 ターム以内にあるドキュメントを検索します。例えば、dog NEAR cat は、cat の 10 ターム以内に dog を含んでいるドキュメントとマッチされます。
query1 NEAR/*N* *query2*	dog NEAR/2 cat	cts:near-query の場合と同様、NEAR 演算子の両側にあるクエリとマッチし、それらが互いに *N* ターム以内にあるドキュメントを検索します。この例は、cat というタームから 2 ターム以内に dog というタームがあるドキュメントとマッチされます。
constraint:*value*	color:red decade:1980s birthday:1999-12-31	cts:element-range-query などのレンジクエリの場合と同様、指定された値を含む名前付き制約とマッチするドキュメントを検索します。詳細については、「制約での関係演算子の使用」（65 ページ）を参照してください。
operator:state	sort:relevance sort:date	検索オプション内の operator XML 要素または JSON キーによって定義される、ソート順序などのランタイム設定演算子を適用します。詳細については、「演算子オプション」（268 ページ）を参照してください。
constraint LT *value*	color LT red birthday LT 1999-12-31	*value* 未満の値を含む名前付きレンジ制約とマッチするドキュメントを検索します。詳細については、「制約での関係演算子の使用」（65 ページ）を参照してください。

クエリ	例	説明
constraint LE *value*	`color LE red` `birthday LE 1999-12-31`	*value* 以下の値を含む名前付きレンジ制約とマッチするドキュメントを検索します。詳細については、「制約での関係演算子の使用」（65 ページ）を参照してください。
constraint GT *value*	`color GT red` `birthday GT 1999-12-31`	*value* よりも大きい値を含む名前付きレンジ制約とマッチするドキュメントを検索します。詳細については、「制約での関係演算子の使用」（65 ページ）を参照してください。
constraint GE *value*	`color GE red` `birthday GE 1999-12-31`	*value* 以上の値を含む名前付きレンジ制約とマッチするドキュメントを検索します。詳細については、「制約での関係演算子の使用」（65 ページ）を参照してください。
constraint NE *value*	`color NE red` `birthday NE 1999-12-31`	*value* と等しくない値を含む名前付きレンジ制約とマッチするドキュメントを検索します。詳細については、「制約での関係演算子の使用」（65 ページ）を参照してください。
query1 BOOST *query2*	`george BOOST washington`	*query1* とマッチするドキュメントを検索します。その中で *query2* ともマッチするドキュメントの関連度スコアをブーストさせます。この例は、「george」というタームと一致するものをすべて返し、その中で同一ドキュメント内で「washington」というタームも含んでいるものにより高い関連度スコアを付与します。詳細については、「`cts:boost-query`」を参照してください。

3.2.2 演算子の優先順位

次の表に、デフォルトの文法での演算子の優先順位を示します。上の方が優先順位が高くなっています。表の各行は、優先順位レベルを表します。複数の演算子が同じ優先順位を持つ場合、評価は左から右への順序で行われます。表内で上位にある演算子を使用するクエリ内の式は、下位にある演算子を使用する式よりも前に評価されます。

演算子
:、LT、LE、GT、GE、NE
-
NOT_IN
BOOST
( )、NEAR、NEAR/*N*
AND
OR

例えば、AND は、OR よりも高い優先順位を持つので、クエリ

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

では、次のように記述されているように評価します。

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

3.2.3 制約での関係演算子の使用

関係クエリ演算子（LT、LE、GT、GE、および NE では、左側に制約名、右側に値が利用できます。つまり、これらの演算子を使用するクエリは、次のような形式を取ります。

制約 op *値*

これらの関係演算子は、演算子によって定義された関係性（等しい、小さい、大きいなど）と一致する値を含む名前付き制約を満たすフラグメントとマッチされます。例えば、クエリオプションで color という名前の、要素の単語制約を定義した場合、color:red は、red という値を含む color 制約を満たす要素を含んでいるドキュメントとマッチされます。詳細および他の例については、「制約オプション」（257 ページ）を参照してください。

制約名は、検索を制御するクエリオプションによって定義された <constraint/> という XML 要素または「constraint」という JSON オブジェクトの名前である必要があります。制約は、単語制約、値制約、レンジ制約、位置制約のいずれかにすることができます。制約には、関連付けられているレンジインデックスがある必要があります。

制約がバケット化されていない場合、演算子の右側の値は、制約の型に変換できる必要があります。例えば、制約の後ろにあるレンジインデックスの型が xs:date である場合、マッチされる値は xs:date 形式である必要があります。

制約がバケット化されている場合、値は、制約によって定義されたバケットの名前になっている必要があります。例えば、「バケットレンジ制約の例」（266 ページ）で定義された decade というバケット制約を使用して検索する場合、右側の値は、1920s や 2000s のようなバケット名、つまり、decade:1920s のようにする必要があります。

3.2.4 文字列クエリの例

デフォルトの文法には、複雑なクエリを生成するための堅牢な機能が用意されています。次に、デフォルトの文法を使用するいくつかのクエリの例を示します。

- `(cat OR dog) NEAR vet`

  vet という単語から 10 ターム以内（cts:near-query のデフォルトの距離）に cat または dog が少なくとも 1 つある

- `dog NEAR/30 vet`

  vet という単語から 30 ターム以内にある dog

- `cat -dog`

dog という単語がない場所での、cat という単語

3.3 文字列クエリの文法の修正と拡張

文字列クエリを構築するために使用される文法は、検索で使用されるクエリオプションでカスタムの grammarXML 要素または JSON オブジェクトを指定することによってカスタマイズできます。文法は、次のコンポーネントによって定義されます。

- starter
- joiner
- quotation
- implicit

grammar は、少なくとも starter、joiner、または implicit 要素を 1 つ含んでいる必要があります。grammar 要素がクエリオプション内に存在しても空になっている場合、検索が term-option の設定に従ってパースされます。

次に、「デフォルトの文字列クエリの文法」（61 ページ）で説明されているシンタックスとセマンティックを実装するデフォルトの文字列クエリ grammar を示します。デフォルトの文法は、デフォルトのクエリオプションを取得することによって表示できます。詳細については、「デフォルトのクエリオプションの取得」（256 ページ）を参照してください。

```
<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>
    <joiner strength="30" apply="infix" element="cts:near-query"
tokenize="word">NEAR</joiner>
    <joiner strength="30" apply="near2" consume="2"
element="cts:near-query">NEAR/</joiner>
    <joiner strength="32" apply="boost" element="cts:boost-query"
tokenize="word">BOOST</joiner>
    <joiner strength="35" apply="not-in" element="cts:not-in-query"
tokenize="word">NOT_IN</joiner>
    <joiner strength="50" apply="constraint">:</joiner>
    <joiner strength="50" apply="constraint" compare="LT"
tokenize="word">LT</joiner>
    <joiner strength="50" apply="constraint" compare="LE"
tokenize="word">LE</joiner>
    <joiner strength="50" apply="constraint" compare="GT"
tokenize="word">GT</joiner>
    <joiner strength="50" apply="constraint" compare="GE"
tokenize="word">GE</joiner>
    <joiner strength="50" apply="constraint" compare="NE"
tokenize="word">NE</joiner>
  </grammar>
```

次の表に、検索文法で使用される概念の説明を示します。

概念	説明
implicit	*implicit* 文法要素は、2 つの検索タームを結合するためにデフォルトで使用する cts:query を指定します。デフォルトでは、search API は cts:and-query を使用しますが、implicit 文法オプションで任意の cts:query に変更できます。
starter	*starter* は、タームの前に記述される、そのタームに対する特別なパーシングを示す文字列です。例えば、否定を表すマイナス記号（-）があります。さらに、delimiter 属性とともに使用された場合、starter は、グループ化するためにタームを区切る先頭と末尾の文字列を指定します。また、文法で文字列をパーシングするときのタームの優先順位の順序を設定することもできます。

joiner	*joiner* は、2 つのタームを結合する文字列です。例えば、AND および OR は、デフォルトの文法を使用している次のクエリ内で結合子として機能しています。
	`cat AND dog`
	`cat OR dog`
	デフォルトの文法では、制約または演算子をその値と区切る文字列にも結合子を使用します（「制約オプション」（257 ページ）および「演算子オプション」（268 ページ）を参照）。「word」属性に設定された joiner/@tokenize が存在する場合、タームと結合子はスペースで区切る必要があります。それ以外の場合は、パーサーにより、クエリテキスト内で結合子文字列が探されます。

quotation	quotation 文字列では、語句の先頭と末尾を示すために使用する文字列を指定します。例えば、デフォルトの文法では、次は（AND で組み合わされた複数のタームではなく）語句としてパースされます。 `"this is a phrase"`
strength	*strength* 属性は、最初に処理するトークンに関する情報をパーサーに提供します。strength が高いトークンまたはグループは、strength が低いトークンまたはグループの前に処理されます。

starter 要素は、文法の一部分をパースする方法を定義します。apply 属性は、starter および delimiter が適用される関数を指定します。

joiner 要素は、さまざまな演算子、制約、および他の演算をパースする方法を定義し、結合子の動作を定義する関数を指定します。例えば、トークンを cts:or-query で結合する前述の OR 結合子を、パイプ文字（|）を使用するように変更する場合は、前述の joiner 要素の代わりに次のものを使用します。

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

@tokenize を word に設定すると、トークンが認識されるためにはその直前または直後にスペースが必要であることが指定されます。この属性がなく、OR が結合子である場合、CORN を検索すると、C OR N（cts:or-query(("C"), ("N"))）の検索結果が返されます。制約で結合子（例えばコロン文字 :）を使用している場合、通常そのように処理されたくないので、tokenize 属性を省略することで、decade:1990s のような制約をパースする検索ができます。

apply="infix" 結合子の element 属性で要素を指定することにより、クエリのシーケンスを取るコンポーザブルな cts:query 要素（cts:or-query、cts:and-query、または cts:near-query）を指定する結合子文字列を追加できます。例えば、次の search:joiner 要素は、cts:near-query の結合子を指定します。これにより、結合子文字列 CLOSETO を使用して前後のタームが cts:near-query で組み合わされます（デフォルトの距離である 10 が使用されます）。

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

上記の結合子の指定を使用すると、クエリテキスト bicycle CLOSETO shop は、bicycle と shop が互いに 10 ターム以内にある一致を返します。

デフォルトでは、検索文法は非常に強力であり、Google の文法に似た文法を実装しています。これをカスタマイズしてさらに強力にし、特定のニーズに合わせることができます。カスタムパーシングを追加するには、関数を実装し、apply、ns、at デザインパターン（「オプションおよび拡張による検索のカスタマイズ」（28 ページ）を参照）を使用し、実装した関数を参照するように search:grammar options ノードを構築する必要があります。

3.3.1 starter

starter（開始子）は、単項のプレフィックス演算子またはグループ化記号のペアを定義します。例えば、デフォルトの文法では、マイナス記号（-）を否定の開始子として定義し、丸括弧（()）をグループ化の開始子として定義しています。

grammar クエリオプションには、0 個以上の開始子要素を入れることができますが、少なくとも 1 つの starter または joiner を含んでいる必要があります。

文法で単項の開始子演算子を定義するには、次の手順を実行します。

1. apply、at、および ns を使用して、XQuery のパーシング関数を特定します（「オプションおよび拡張による検索のカスタマイズ」（28 ページ）を参照）。

2. 演算子トークンを XML の <starter/> テキストノードまたは JSON の label サブオブジェクトに入れます。

3. この演算子が持つ、同じ文法内の他の演算子との相対的な優先順位を、strength で設定します。

4. element を、パーシング関数によって返される cts:query 要素の QName に設定します。例えば、デフォルトの文法によって定義されている否定演算子では、cts:not-query 要素が生成されます。

5. オプションとして、スペース区切りの検索オプションのリストをパーシング関数に渡すように options を設定します。

例えば、デフォルトの文法は、次のように単項の “-” 演算子を定義しています。

XML	JSON

<starter strength="40" apply="prefix"
element="cts:not-query">-</starter>
	"starter": [
	{
	"strength": 40,
	"apply": "prefix",
	"element": "cts:not-query",
	"label": "-"
	}
	]

文法でグループ化記号を定義するには、次の手順を実行します。

1. apply、at、および ns を使用して、XQuery のパーシング関数を特定します（「オプションおよび拡張による検索のカスタマイズ」（28 ページ）を参照）。

2. グループ化開始トークンを XML の <starter/> テキストノードまたは JSON の label サブオブジェクトに入れます。

3. delimiter をグループ化終了トークンに設定します。

4. この演算子が持つ、同じ文法内の他の演算子との相対的な優先順位を、strength で設定します。

5. element を、パーシング関数によって返される cts:query 要素の QName に設定します。例えば、デフォルトの文法によって定義されている否定演算子では、cts:not-query 要素が生成されます。

6. オプションとして、スペース区切りの検索オプションのリストをパーシング関数に渡すように options を設定します。

例えば、デフォルトの文法は、次のように "( )" をグループ化トークンとして定義しています。

XML	JSON
<starter strength="30" apply="grouping" delimiter=")">(</starter>	"starter": [{ "strength": 30, "apply": "grouping", "delimiter": ")", "label": "(" }]

3.3.2 joiner

Joiner（結合子）は、2 つの文字列クエリ式を「結合」する二項の演算子を定義します。デフォルトの文法での結合子の例として、AND、OR、LT、コロン（:）などがあります。

grammar クエリオプションには、0 個以上の結合子を入れることができますが、少なくとも 1 つの starter または joiner を含んでいる必要があります。

結合子を定義するには、次の手順を実行します。

1. apply、at、および ns を使用して、XQuery のパーシング関数を特定します（「オプションおよび拡張による検索のカスタマイズ」（28 ページ）を参照）。

2. 演算子トークンまたは記号を XML の <joiner/> テキストノードまたは JSON の label サブオブジェクトに入れます。

3. この演算子が持つ、同じ文法内の他の演算子との相対的な優先順位を、strength で設定します。

4. element を、パーシング関数によって返される cts:query 要素の QName に設定します。例えば、デフォルトの文法によって定義されている AND 演算子では、cts:and-query 要素が生成されます。

5. オプションとして、スペース区切りの検索オプションのリストをパーシング関数に渡すように options を設定します。

プレフィックス演算子を定義するには、演算子トークンを XML の <starter/> テキストノードまたは JSON の label サブオブジェクトに入れます。例えば、デフォルトの文法は、次のように単項の "-" 演算子を定義しています。

XML	JSON

<starter strength="40" apply="prefix"
element="cts:not-query">-</starter>
	"starter": [
	{
	"strength": 40,
	"apply": "prefix",
	"element": "cts:not-query",
	"label": "-"
	}
	]

3.3.3 quotation

quotation 文法要素は、語句を分離するために使用する記号を定義します。デフォルトの文法では、引用符（"）を使用します。

```
<quotation>"</quotation>
```

文法には、1 つ以下の quotation 定義を入れることができます。

XML では、<quotation/> 要素のテキストノード内に記号を配置します。JSON では、quotation キーに関連付けられている文字列値内に記号を配置します。例えば、語句に対して二重引用符の代わりにパーセント（%）を使用するには、文法に次を含めます。

XML	JSON
<quotation>%</quotation>	"quotation": "%"

3.3.4 implicit

implicit 文法要素は、明示的な結合子演算子によって区切られていない隣接するタームの処理方法を指定します。例えば、“cat dog” のような文字列クエリを解釈する方法を指定します。grammar クエリオプションには、1 つ以下の暗黙的なルールを入れることができます。

暗黙的な演算を定義するには、次の手順を実行します。

1. 「cts:query とは」（240 ページ）で定義されている cts:query 階層からクエリタイプを選択します。

2. XML のクエリオプションを構築する場合は、<implicit/> 要素に、適切なタイプの子要素を追加します。これは cts:query コンストラクタを使用して構築できます。例えば、“cts:and-query((), ())” を評価することにより、空の cts:and-query を構築できます。

3. JSON のクエリオプションを構築する場合は、implicit キーに関連付けられている値を、手順 1 で選択した cts:query 要素タイプのシリアライズされた表現に設定します。

例えば、デフォルトの文法には、暗黙的な演算として cts:and-query を指定する implicit ルールが含まれます。そのため、「cat dog」は「cat AND dog」と等価になります。

XML	JSON

<implicit>
  <cts:and-query
  xmlns:cts="http://marklogic.com/cts"/>
</implicit>
	"implicit":
	  "<cts:and-query
	xmlns='http://marklogic.com/cts
	'"

4.0 構造化クエリを使用した検索

この章では、XML または JSON で表現された構造化クエリを使用して検索を実行する方法について説明します。`search:parse` または `search:search` によってデフォルトで生成される注釈付きの `cts:query` は、クエリを大幅に修正する必要がない場合に適しています。独自のクエリを生成する場合、または search API の文法ルールとは異なるルールを使用してクエリをパースする必要がある場合、代わりのクエリスタイルとして構造化クエリがあります。

この章は、次の内容から構成されます。

- 構造化クエリの概要
- 構造化クエリの概念
- 構造化クエリの構築
- シンタックスの概要
- 構造化クエリの例
- シンタックスリファレンス

4.1 構造化クエリの概要

*構造化クエリ*は、検索式の抽象構文ツリー表現であり、XML または JSON で表現されます。例えば、次は、文字列クエリ “cat AND dog” と等価な、XML の構造化クエリです。

```
<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>
```

クエリの途中を、何らかの方法で拡張または操作する必要がある場合はいつでも、構造化クエリの使用を検討してください。次のような場合、構造化クエリが適しています。

- 文字列クエリではうまくいかないクエリ。例えば、複雑な多角形によって場所を指定する制約は、マシンで生成されるように設計されています。
- 複雑なルールの集まり、または文字列クエリの文法で適切に表現できないルールの集まりを含む検索文字列。

- 事前にパースされた構造化クエリを、ユーザー生成または動的に構築された文字列クエリと組み合わせる場合。
- 社内独自のクエリ形式など、MarkLogic 以外のクエリ表現を MarkLogic サーバーで使用可能な形態に変換する場合。
- アプリケーション固有の検証を必要とする文字列クエリ。

構造化クエリは、XQuery 関数 `search:parse` を使用するか、構造化クエリを返す独自のコードを記述することで生成できます。構造化クエリを使用して、次の MarkLogic サーバー API でコンテンツおよびメタデータを検索できます。

- XQuery Search API。詳細については、「`search:search`」および「`search:resolve`」を参照してください。
- Java API。詳細については、『Java Application Developer's Guide』の「Searching」を参照してください。
- REST API。詳細については、『REST Application Developer's Guide』の「Using and Configuring Query Features」を参照してください。

4.2 構造化クエリの概念

このセクションで扱われている概念は、構造化クエリのさまざまな構成要素の目的ならびに対象範囲を理解するために役立ちます。特定のサブクエリの詳細については、「シンタックスリファレンス」（87 ページ）を参照してください。

次の内容を扱います。

- 主なクエリカテゴリ
- タームクエリとワードクエリの違い
- 包含とは
- 完全一致と不完全一致
- 構造化クエリのサブクエリのタクソノミー

4.2.1 主なクエリカテゴリ

クエリは、検索条件をカプセル化します。次の種類のサブクエリは、マッチさせるコンテンツを記述するための検索用の基本的な構成要素です。

クエリタイプ	説明
ターム	ドキュメントまたはコンテナ内の任意の場所に出現する単語または語句とマッチさせます。タームクエリは、テキスト値のサブセットとマッチさせます。デフォルトでは、タームクエリは完全一致を使用しません。
ワード	特定の JSON プロパティ、XML 要素または属性、あるいはフィールドのような特定の場所にある単語（ワード）または語句とマッチさせます。ワードクエリは、テキスト値のサブセットとマッチさせます。デフォルトでは、ワードクエリは完全一致を使用しません。
値	JSON プロパティや XML 要素のような特定の場所にある、文字列や数字のようなリテラル値全体とマッチさせます。デフォルトでは、値クエリは完全一致を使用します。
レンジ	入力された値に適用される関係式を満たす値とマッチさせます。“5 未満 ” や “true と等しくない ” のような条件を表すことができます。レンジクエリは、レンジインデックスが利用されているか、フィルタリングされた検索操作で使用される必要があります。

追加の種類のサブクエリを使用すると、基本的なコンテンツクエリを互いに組み合わせたり、追加の条件および制約と組み合わせたりできます。追加のクエリタイプは、次の一般的なカテゴリに分類されます。

- 論理構成子：条件間の論理的な関係性を表します。“*x* AND (*y* OR *z*)” のような複合論理式を構築できます。
- ドキュメントセレクタ：コレクション、ディレクトリ、または URI に基づいてドキュメントを選択します。例えば、“ コレクション *y* 内のドキュメント内で *x* がマッチした場合 ” のような条件を表すことができます。
- 場所修飾子：マッチが出現した場所に基づいて条件をさらに制限します。例えば、“JSON プロパティ *z* に含まれている *x* のみ ”、“*y* から *n* 語以内でマッチする *x* のみ ”、“ ドキュメントのプロパティ内でマッチする *x* のみ ” などです。

4.2.2 タームクエリとワードクエリの違い

タームクエリとワードクエリの主な違いは、包含の処理方法にあります。タームクエリはそのコンテキストコンテナ内のあらゆる場所でマッチするものを検索しますが、ワードクエリは直接の子とのみマッチされます。例えば、JSON または XML ドキュメントに次の構造があると仮定します。

JSON	XML

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

「a」内の「value」を検索するタームクエリでは、次の 2 つ、すなわち「b」内のものと、「d」内のものがマッチします。一方、「a」内の「value」を検索するワードクエリでは、「value」は直接の子ではないため、マッチがありません。

ワードクエリで「value」の出現箇所を見つけるには、範囲を「b」または「d」に制約する必要があります。例えば、次のサブクエリはそれぞれ、JSON、XML ドキュメント内の「b」内の「value」とマッチされます。

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 包含とは

特定のコンテナをあるコンテキストにおいてマッチさせる場合、さまざまなサブクエリで制約を課すことができます。「コンテナ」とは、JSON プロパティ、XML 要素、または XML 要素の属性のことです。

コンテナの名前を対象とするワードクエリや値クエリは、そのコンテナ内での出現とのみマッチされます。しかしこの際、コンテナは、ドキュメントや他のコンテナ内の任意の階層にあるものが対象となります。つまり、直接の子でなければならないというわけではありません。

例えば、次のワードクエリは、「a」という名前の JSON プロパティまたは XML 要素の値内に出現する場合にのみ「value」とマッチされます（これらの例では、JSON 版では `json-property` を使用し、XML 版では `element` を使用していますが、クエリ形式に関係なくこれらの指定子を使用できます）。

JSON	XML

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

ただし、特に制約がない場合、「b」という名前のコンテナはあらゆる場所に発生する可能性があります。例えば、次の 2 つのドキュメントでは、「value」を含む JSON プロパティ（または XML 要素）が 2 つあるため、それぞれ、2 つマッチします。

JSON	XML

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

container-query でクエリをラップすることで、マッチの対象をさらに制限できます。例えば、次のクエリは、「c」内で「b」が出現している場合にのみ、「b」内の「value」とマッチされます。

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>`

マッチの対象は、コレクション、データベースのディレクトリ、プロパティのフラグメントなど、他の方法でも制限できます。詳細については、「場所修飾子」（81 ページ）および「ドキュメントセレクタ」（81 ページ）を参照してください。

4.2.4 完全一致と不完全一致

値クエリは、完全一致（exact match）を使用します。デフォルトでは、ワードクエリとタームクエリではそうではありません。

例えば、value-query は、デフォルトでは次のようにマッチされます。

- 検索条件の値は、大文字/小文字の区別、発音符号による区別、句読点による区別、スペースの区別がオンになった（つまりこれらを違うものとして区別する）状態でマッチされます。
- ステミング（語幹処理）とワイルドカード機能は使用されません。
- クエリ内の値が、ドキュメント内の値のサブセットである場合はマッチしません。

したがって、`value-query` で「thomas」を検索した場合、「Thomas」や「thomas edison」のような値とはマッチしません。

これとは対照的に、word-query は完全一致を使用せず、次のようにマッチされます。

- 検索条件の値は、大文字/小文字の区別、発音符号による区別、句読点による区別、スペースの区別がオフになった（つまりこれらを同一のものとして扱う）状態でマッチされます。
- 語幹（つまりステミングされたもの）とマッチされます。
- 該当データベースにおいてワイルドカード機能がオンの場合、ワイルドカードマッチが実行されます。

したがって、`word-query` を使用して「thomas」を検索した場合、「Thomas」や「thomas edison」のような値とマッチします。

これらの動作の一部は、クエリオプションおよびデータベース設定で上書きできます。例えば、該当データベースにおいてワイルドカード検索がオフの場合、ワードクエリで「thom*」を検索しても「Thomas」とマッチしません。同様に、特定の `word-query` または `value-query` にだけローカルなタームオプションを設定することも、あるいは `term-options` クエリオプションでより広い範囲を対象に設定することもできます。

4.2.5 構造化クエリのサブクエリのタクソノミー

構造化クエリでは、「主なクエリカテゴリ」（76 ページ）で説明されているすべてのクエリタイプを明示的に公開しています。このセクションは、この分類に基づいて、必要なサブクエリの種類を見つけるためのクイックリファレンスです。

ほとんどの種類のサブクエリは、互いに組み合わせて複雑なクエリを構築するために使用できます。詳細については、「シンタックスリファレンス」（87 ページ）を参照してください。

- 基本的なコンテンツクエリ
- 論理式構成子
- 場所修飾子
- ドキュメントセレクタ

4.2.5.1 基本的なコンテンツクエリ

基本的なコンテンツクエリは、「JSON のプロパティ A に値 B が入っている」または「『dog』という語句が含まれる任意のドキュメント」のように、コンテンツに関する検索条件を表します。これらのクエリは、サブクエリを含んでいないため、複雑な複合クエリの構造内で「リーフ」として機能します。

- 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 論理式構成子

論理構成子は、1 つあるいは複数のサブクエリを結合して 1 つの論理式にするクエリです。例えば、「query1 と query2 の両方にマッチするドキュメント」または「query1、query2、query3 のいずれかとマッチするドキュメント」などです。

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

4.2.5.3 場所修飾子

場所修飾子は、コンテンツ内のみ、メタデータ内のみ、指定された JSON プロパティまたは XML 要素に含まれている場合のみなど、サブクエリがマッチした場所に基づいて結果を制限します。例えば、「メタデータ内でこのサブクエリにマッチするもの」または「JSON プロパティ P でこのサブクエリにマッチするもの」などです。

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

4.2.5.4 ドキュメントセレクタ

ドキュメントセレクタは、ドキュメントのグループをマッチする対象としてコンテンツ（内容）本体ではなく、データベースの属性（コレクションのメンバーシップ、ディレクトリ、URI など）を使用するクエリです。例えば、「コレクション A および B 内のすべてのドキュメント」または「ディレクトリ D 内のすべてのドキュメント」などです。

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

4.3 構造化クエリの構築

構造化クエリは、次のように複数の方法で構築できます。

- 「シンタックスリファレンス」（87 ページ）で説明されているシンタックスを使用して手動で構築する。
- XQuery で、XQuery 関数 `search:parse` を呼び出し、「`search:query`」を 3 番目のパラメータとして指定して、XML 表現を表示することで構築する。
- Java で、クラス `com.marklogic.client.query.StructuredQueryBuilder`、`com.marklogic.client.pojo.PojoQueryBuilder`、または同等のインターフェイスを使用して構築する。

XQuery Search API では、XML の構造化クエリのみが利用できます。REST API および Java API では、XML および JSON 表現が利用できます。

XML の場合は、Query Console で `search:parse` 手法を使用して、文字列クエリまたはシリアライズされた `cts:query` がどのように構造化クエリにマッピングされるかを調べ、必要に応じてその構造化クエリを修正できます。例えば以下のようになります。

```
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")
```

上記のクエリを Query Console で実行し、結果を XML として表示した場合、デフォルトの検索オプションを有効にしてパースしたときの「cat AND dog」の XML 表現が得られます。

```
<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 シンタックスの概要

このセクションでは、構造化クエリの構造について簡単に説明します。詳細については、「シンタックスリファレンス」（87 ページ）を参照してください。

構造化クエリは、`search:query` の XML 要素であり、`cts:query` 構成子、`cts:query` スコープ設定コンストラクタ、search API コンポーネント用の抽象化（制約や演算子など）を表現する子要素を持ちます。XQuery の Search API を使用する場合は、XML 表現を使用する必要があります。REST および Java では、XML 表現または JSON 表現を選択できます。

XQuery の `cts:query` コンストラクタと同じように、構造化クエリはコンポーザブルであり、複雑な検索クエリを作ることができます。XML 版の構造化クエリには、`<search:query>` ラッパーノードが必要です。例えば以下のようになります。

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

同様に、JSON 版の構造化クエリには、`query` ラッパーオブジェクトがあります。このラッパー内で、サブクエリは `queries` 配列で囲まれます。複数のサブクエリが出現する可能性がある場合は常に、queries ラッパーが使用されます。例えば以下のようになります。

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

and-query、or-query などを含む複雑なクエリを構成し、検索対象のタームを含むタームクエリを任意の数入れることができます。クエリを要素、属性、JSON プロパティ、で制約したり、あるいはフィールドに対してレンジクエリを使用できます。MarkLogic サーバーで `cts:query` 式がどのように機能するかの背景知識については、「cts:query 式の構成」（239 ページ）を参照してください。

REST API と Java API は、構造化クエリの XML 表現と JSON 表現をサポートしています。REST API と Java API では、MarkLogic の JSON 変換機能を内部的に使用して JSON から XML に、またその逆の変換ができます。この変換の詳細については、『Application Developer's Guide』の「Working With JSON」を参照してください。

4.5 構造化クエリの例

このセクションは、次のような構造化検索の例（XML および JSON）から構成されます。

- 例：シンプルな構造化検索
- 例：テキストによる制約参照がある構造化検索
- 例：制約参照を使用した構造化検索

その他の例については、「シンタックスリファレンス」（87 ページ）を参照してください。

4.5.1 例：シンプルな構造化検索

次に、「imagine a complex search」という語句の文字列クエリと等価な構造化クエリの XML ノードを示します。

```
<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>
```

次に、同じクエリの JSON 表現を示します（REST API および Java API で使用できます）。

```
{
  "query": {
    "queries": [{
      "term-query": {
        "text": [ "imagine a complex search" ]
      }
    }]
}}
```

Xquery の場合、次のコードに示すように、search:parse で文字列クエリから XML 構造化クエリを生成し、search:resolve を使ってその構造化クエリで検索できます。

```
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
```

REST API と Java API には、直接、構造化クエリで検索できるインターフェイスがあります。詳細については、『REST Application Developer's Guide』の「Searching With Structured Queries」および『Java Application Developer's Guide』の「Search Documents Using Structured Query Definition」を参照してください。

4.5.2 例：テキストによる制約参照がある構造化検索

次の構造化検索クエリは、これまでより少し複雑です。このクエリには、タームを組み合わせるための and-query があり、また検索の options ノードで定義される制約を参照します。ここでは、デフォルトのクエリ文法（例えば、`decade:1940s` は、値が `1940s` である `decade` 制約を表します）を使用してパースされると想定しています。

```
<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>
```

次に、これを JSON で示します。

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

4.5.3 例：制約参照を使用した構造化検索

次の例は、クエリオプションで定義される制約の明示的な参照を含むクエリを示しています。

クエリオプションに、decade というバケット制約の定義が含まれ、バケットの 1 つの名前が 1940s であると仮定します。

```
<!-- すべてのオプションに関しては「バケットの例」(56 ページ) を参照してください。-->
<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>
```

上記のオプションで評価された場合、文字列クエリ「hepburn AND decade:1940s」は、レンジ制約（decade:1940s）を表します。これは decade 制約の 1940s バケットの条件を満たすものにマッチを制限します。次に、これと等価な構造化クエリを XML で示します。

```
<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>
```

次に、これを JSON で示します。

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

4.6 シンタックスリファレンス

このセクションでは、構造化クエリに関するシンタックス情報を示します。構造化クエリ内の最上位レベルの要素それぞれに、サブセクションがあります。各セクションには、詳細なシンタックス、子要素の説明、および例が含まれます。まず最上位レベルの `query` ラッパーから説明します。

- query
- 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
- 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
- lsqt-query
- period-compare-query
- period-range-query
- operator-state

4.6.1 query

`query` は、構造化クエリ定義を囲む、最上位レベルのラッパーです。サブクエリとして子を 1 つあるいは複数入れることができます。詳細については、各子に関するサブセクションを参照してください。

XML	JSON

すべての要素は名前空間
http://marklogic.com/appservices/search
内にあります。
 	
<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

1 つあるいは複数の検索タームまたは語句とマッチさせるクエリです。デフォルトでは、term-query は cts:word-query と等価です。ただし、term クエリオプションを使用してクエリのターム処理をカスタマイズしている場合、これらは違うものとなることがあります。例えば、検索でフィールドのワード制約を指定する term オプションを含む場合、term-query は cts:field-word-query として処理されることがあります。詳細については、『REST Application Developer's Guide』の「term-definition」（JSON）または「<term>」（XML）を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.2.1 シンタックスの概要

XML	JSON

すべての要素は名前空間 http://marklogic.com/appservices/search 内にあります。
 	
`<term-query>`
`  <text>`*term*`-or-phrase</text>`
`  <weight>`*value*`</weight>`
`</term-query>`
	`"term-query": {`
	`  "text": [ "`*term-or-phrase*`" ]`
	`  "weight": "`*value*`"`
	`}`

4.6.2.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
text	はい	検索するタームまたは語句。このクエリには、text の子を複数入れることができます。
weight	いいえ	このクエリの重み。デフォルト：1.0。重みが大きいほど、検索結果の関連度が高くなります。重みは、-16 から 64 の間にする必要があります。64 よりも大きい値は、64 と同じものとして扱われます。絶対値が 0.0625 未満のもの（つまり -0.0625 と 0.0625 の間）は 0 として扱われます。このため、スコアに影響しません。

4.6.2.3 例

次の例では、「dog」または「cat」というタームを含んでいるドキュメントを検索します。

XML	JSON

すべての要素は名前空間 http://marklogic.com/appservices/search 内にあります。
 	
`<query>`
`  <term-query>`
`  <text>dog</text>`
`  <text>cat</text>`
`  </term-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "term-query": {`
	`  "text": [ "dog", "cat" ]`
	`  }`
	`  }]`
	`}}`

4.6.3 and-query

1 つあるいは複数のサブクエリによるマッチの積集合を求めます。詳細については、「cts:and-query」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.3.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<and-query>`
`  anyQueryType`
`  <ordered>bool</ordered>`
`</and-query>`
	`"and-query": {`
	`  "queries": [`
	`  anyQueryType,`
	`  "ordered": boolean`
	`  ]`
	`}`

4.6.3.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
anyQueryType	いいえ	1 つあるいは複数のサブクエリ。
ordered	いいえ	サブクエリの順序でサブクエリをマッチさせる必要があるかどうか。例えばサブクエリが「cat」と「dog」の場合、順序付きクエリでは、「cat」と「dog」の両方が出現し、かつこのフラグメント内で「cat」そして「dog」の順で出現するフラグメントのみとマッチされます。デフォルト：false。

4.6.3.3 例

次の例では、ターム「dog」と「cat」の両方を含んでいて、「dog」そして「cat」の順で出現するドキュメントを検索します。

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<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

1 つあるいは複数のサブクエリによるマッチの和集合を求めます。詳細については、「`cts:or-query`」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.4.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<or-query>`
`  anyQueryType`
`</or-query>`
	`"or-query": {`
	`  "queries": [ anyQueryType ]`
	`}`

4.6.4.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
anyQueryType	いいえ	1 つあるいは複数のサブクエリ。

4.6.4.3 例

次の例は、「dog bone」という語句（フレーズ）または「cat」というタームのいずれかを含んでいるドキュメントとマッチされます。

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<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

2 つのサブクエリによるマッチの差集合を求めます。つまり、肯定クエリ（positive-query）とマッチするが、否定クエリ（negative-query）とマッチしない結果を返します。詳細については、「`cts:and-not-query`」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.5.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<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 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`positive-query`	はい	フィルタリングで含められる結果を指定するクエリ。すべての結果はこのクエリとマッチします。
`negative-query`	はい	フィルタリングで除外される結果を指定するクエリ。どの結果もこのクエリとマッチしません。

4.6.5.3 例

次の例は、同じフラグメント内に「cat」が出現しない「dog」の出現とマッチされます。

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<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

サブクエリとマッチする結果をフィルタリングで除外するクエリ。詳細については、「`cts:not-query`」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.6.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
http://marklogic.com/appservices/search
内にあります。
 	
<not-query>
anyQueryType
</not-query>
	"not-query":{
	anyQueryType
	}

4.6.6.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
anyQueryType	はい	否定クエリ。これは、フィルタリングで除外する検索結果を指定します。

4.6.6.3 例

次のクエリは、「dog」というタームを含まないドキュメントとのみマッチされます。

XML	JSON

すべての要素は名前空間
http://marklogic.com/appservices/search
内にあります。
 	
<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

指定された距離の範囲内でマッチする結果のみを返すクエリ。詳細については、「cts:not-in-query」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.7.1 シンタックスの概要

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<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 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
positive-query	はい	肯定クエリ。これは、フィルタリングで含める検索結果を指定します。
negative-query	はい	否定クエリ。これは、フィルタリングで除外する検索結果を指定します。

4.6.7.3 例

下の例は、「man bites dog」の一部ではない「dog」を少なくとも 1 つ含んでいるフラグメントとマッチされます。

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<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 near-query

指定された距離の範囲内でマッチする結果のみを返すクエリ。詳細については、「`cts:near-query`」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.8.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<near-query>`
`  anyQueryType`
`  <distance>double</distance>`
`  <distance-weight>`
`  double`
`  </distance-weight>`
`  <ordered>boolean</ordered>`
`</near-query>`
	`"near-query": {`
	`  "queries": [`
	`  anyQueryType,`
	`  "distance": "number",`
	`  "distance-weight": "number",`
	`  "ordered" : boolean`
	`  ]`
	`}`

4.6.8.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`anyQueryType`	はい	指定された近接（距離の範囲）内でマッチされる 1 つあるいは複数のクエリ。
`distance`	いいえ	2 つのマッチングクエリの間の距離（単語数)。デフォルト：10
`distance-weight`	いいえ	このクエリの距離に割り当てられた重み。重みが大きいほど、関連度の計算の際に距離の重要性が高くなります(相対的にタームマッチの重要性が低くなります)。デフォルト：1.0。
`ordered`	いいえ	サブクエリの順序でサブクエリをマッチさせる必要があるかどうか。例えばサブクエリが「cat」と「dog」の場合、順序付きクエリでは、「cat」と「dog」の両方が指定された距離内で出現し、かつフラグメント内で「cat」そして「dog」の順で出現するフラグメントのみとマッチされます。デフォルト：false。

4.6.8.3 例

次の例は、「cat」から 2 ターム以内に出現する「dog」とマッチされます。

XML	JSON
すべての要素は名前空間 http://marklogic.com/appservices/search 内にあります。 <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" }] } }] }}

4.6.9 boost-query

まずクエリにマッチするものをすべて探します。その中でブーストクエリともマッチする結果に対して、その検索関連度スコアをブースト（大きく）します。詳細については、「cts:boost-query」および「セカンダリクエリによる関連度スコアのブースト」（297 ページ）を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.9.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<boost-query>`
`  <matching-query>`
`  *anyQueryType*`
`  </matching-query>`
`  <boosting-query>`
`  anyQueryType`
`  </boosting-query>`
`</and-query>`
	`"boost-query":{`
	`  "matching-query":{ *anyQueryType* },`
	`  "boosting-query" :{ *anyQueryType* }`
	`}`

4.6.9.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`matching-query`	いいえ	マッチ用のクエリ。このクエリとマッチするすべての検索結果が返されます（検索オプションにより、モジューロの制限が課されます)。この要素は複数含めることができます。複数の出現は AND で結合されます。出現がない場合、ブーストクエリは暗黙的にすべてのドキュメントとマッチされます。
`boosting-query`	いいえ	関連度スコアのブースト用のクエリ。`matching-query` と `boosting-query` の両方とマッチした結果に対して、`boosting-query` の重みに応じて関連度スコアを修正します。`matching-query` とマッチしない場合、`boosting-query` は評価されません。この要素は複数含めることができます。複数の出現は AND で結合されます。出現がない場合、ブーストクエリは暗黙的にすべてのドキュメントとマッチされます。

4.6.9.3 例

次の例では、「dog」というタームを含むドキュメントを検索します。「cat」というタームも含むドキュメントは、関連度スコアを重み 10.0 に応じてブーストした search:result を返します。このスコアは「cat」を含んでいないドキュメントよりも高くなります。

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<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"`
	`  }}`
	`  }`
	`  }]`
	`}}`

4.6.10 properties-fragment-query

サブクエリがドキュメントプロパティとマッチしたドキュメントすべてに対して、マッチさせるクエリ。詳細については、「`cts:properties-fragment-query`」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.10.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<properties-fragment-query>`
`  anyQueryType`
`</properties-fragment-query>`
	`"properties-fragment-query":{`
	`  anyQueryType`
	`}`

4.6.10.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
anyQueryType	はい	ドキュメントのプロパティに対して実行するサブクエリ。

4.6.10.3 例

次の例は、2012 年 12 月 31 日以降に修正されたすべてのドキュメントとマッチさせるものです。ここでは「last-modified」プロパティに要素のレンジインデックスが定義してあり、クエリに「modified」制約を定義するオプションが含まれると想定しています。

形式	クエリ

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>

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"
	  }
	  }
	  }]
	}}

4.6.11 directory-query

指定された URI を持つディレクトリ内のドキュメントとマッチさせるクエリ。詳細については、「cts:directory-query」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.11.1 シンタックスの概要

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<directory-query>`
`  <uri>`*`directory-uri`*`</uri>`
`  <infinite>`*`boolean`*`</infinite>`
`</directory-query>`
	`"directory-query": {`
	`  "uri": [ `*`directory-uris`*` ],`
	`  "infinite": `*`boolean`*
	`}`

4.6.11.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`uri`	はい	1 つあるいは複数のディレクトリの URI。ディレクトリの URI は、必ず末尾がスラッシュ（/）です。
`infinite`	いいえ	すべての子ディレクトリを再帰処理するかどうか。デフォルト：true。

4.6.11.3 例

次の例は、データベースのディレクトリ /documents/ または /images/ 内のドキュメントとマッチさせます。その下のサブディレクトリ内とはマッチさせません。

XML	JSON

すべての要素は名前空間 http://marklogic.com/appservices/search 内にあります。
 	
`<query>`
`  <directory-query>`
`  <uri>/documents/</uri>`
`  <uri>/images/</uri>`
`  <infinite>false</infinite>`
`  </directory-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "directory-query": {`
	`  "uri": [`
	`  "/documents/",`
	`  "/images/"`
	`  ],`
	`  "infinite": false`
	`  }`
	`  }]`
	`}}`

4.6.12 collection-query

指定された URI を持つコレクション内のドキュメントとマッチさせるクエリ。詳細については、「cts:collection-query」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.12.1 シンタックスの概要

XML	JSON

すべての要素は名前空間 http://marklogic.com/appservices/search 内にあります。
 	
`<collection-query>`
`  <uri>`*collection-uri*`</uri>`
`</collection-query>`
	`"collection-query": {`
	`  uri": [ `*collection-uris*` ]`
	`}`

4.6.12.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`uri`	はい	1 つあるいは複数のコレクションの URI。

4.6.12.3 例

次の例は、`reports` コレクションまたは `analysis` コレクション内のドキュメントとマッチされます。

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<query>`
`  <collection-query>`
`  <uri>reports</uri>`
`  <uri>analysis</uri>`
`  </collection-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "collection-query": {`
	`  "uri": ["reports","analysis"]`
	`  }`
	`  }]`
	`}}`

4.6.13 container-query

XML 要素または JSON プロパティのコンテンツが指定されたサブクエリとマッチしているドキュメントと、マッチさせるクエリ。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.13.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
http://marklogic.com/appservices/search
内にあります。
 	
<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
	}

4.6.13.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
element	はい	XML 要素の記述子。name と名前空間（ns）によって特定されます。name と ns の両方が必須です。element を含める場合は、json-property を含められません。
json-property	はい	JSON プロパティ名。json-property を含める場合は、element を含められません。
fragment-scope	いいえ	マッチの対象を、指定されたフラグメントスコープに制約します。使用可能な値：documents（デフォルト）または properties。詳細については、「fragment-scope」クエリオプションを参照してください。
anyQueryType	はい	マッチングコンテナ（XML 要素または JSON プロパティ）のコンテンツに対して実行するサブクエリ。

クエリに含められるのは、element と json-property のうち、1 つだけです。

4.6.13.3 例

次の XML の例は、<pets/> 要素の子孫が「dog」というタームとマッチするすべてのドキュメントとマッチされます。JSON の例は、「pets」という名前のプロパティの子孫が「dog」というタームとマッチするすべてのドキュメントとマッチされます。

XML	JSON

すべての要素は名前空間
http://marklogic.com/appservices/search
内にあります。
 	
<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"]
	  }
	  }
	  }]
	}}

4.6.14 document-query

指定された URI を持つドキュメントとマッチさせるクエリ。詳細については、「cts:document-query」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.14.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
http://marklogic.com/appservices/search
内にあります。
 	
<document-query>
  <uri>*document-uri*</uri>
</document-query>
	"document-query": {
	  "uri": [*document-uris*]
	}

4.6.14.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`uri`	はい	1 つあるいは複数のドキュメントの URI。

4.6.14.3 例

次の例は、URI が `/documents/reports.xml` であるドキュメントまたは URI が `/documents/analysis.xml` であるドキュメントとマッチされます。

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<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"`
	`  ]`
	`  }`
	`  }]`
	`}}`

4.6.15 document-fragment-query

サブクエリが任意のドキュメントフラグメントとマッチするすべてのドキュメントとマッチさせるクエリ。詳細については、「`cts:document-fragment-query`」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.15.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<document-fragment-query>`
`  anyQueryType`
`</document-fragment-query>`
	`"document-fragment-query":{`
	`  anyQueryType`
	`}`

4.6.15.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
anyQueryType	はい	あらゆるドキュメントフラグメントに対してマッチさせるクエリ。

4.6.15.3 例

次の例は、`dog` というタームを含んでいるフラグメントを含むドキュメントとマッチされます。

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<query>`
`  <document-fragment-query>`
`  <term-query>`
`  <text>dog</text>`
`  </term-query>`
`  </document-fragment-query>`
`</query>`
	`{`
	`  "query": {`
	`  "queries": [{`
	`  "document-fragment-query": {`
	`  "term-query": {`
	`  "text": [ "dog" ]`
	`  }`
	`  }`
	`  }]`
	`}}`

4.6.16 locks-fragment-query

サブクエリが `document-locks` フラグメントとマッチするすべてのドキュメントとマッチさせるクエリ。詳細については、「`cts:locks-fragment-query`」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.16.1 シンタックスの概要

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<locks-fragment-query>`
`  anyQueryType`
`</locks-fragment-query>`
	`"locks-fragment-query":{`
	`  anyQueryType`
	`}`

4.6.16.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
anyQueryType	はい	あらゆるドキュメントフラグメントに対してマッチさせるクエリ。

4.6.16.3 例

次の例は、`lock:lock-type` 要素内に `write` というタームを含む、ドキュメントロックフラグメントがあるドキュメントとマッチされます。この例では、`lock:lock-type` に要素レンジインデックスが存在すること、および `lock-type` という名前のその要素に対する制約を含むクエリオプションが存在することを前提としています。

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<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" ]`
	`  }`
	`  }}`
	`  }]`
	`}}`

4.6.17 range-query

レンジ制約を適用し、その結果を指定された値と比較するクエリ。詳細については、「制約オプション」（257 ページ）および XQuery 関数「`cts:element-range-query`」、「`cts:element-attribute-range-query`」、「`cts:field-range-query`」、および「`cts:path-range-query`」を参照してください。

`range-query` は、形式が `constraint:value` または `constraint LE value` である文字列クエリ式と等価です。制約は、クエリ内で定義され、レンジインデックスが利用されている必要があります。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.17.1 シンタックスの概要

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<range-query type=index-type collation=uri>`
`  <element name=elem-name ns=namespace />`
`  <attribute name=attr-name ns=namespace />`
`  <json-property>name</json-property>`
`  <field name=field-name collation=uri />`
`  <path-index>path-expr</path-index>`
`  <fragment-scope>scope</fragment-scope>`
`  <value>value</value>`
`  <range-operator>operator</operator>`
`  <range-option>option</range-option>`
`</range-query>`
	`"range-query": {`
	`  "type": index-type,`
	`  "collation": index-collation-uri,`
	`  "element": {`
	`  "name": elem-name,`
	`  "ns": namespace`
	`  },`
	`  "attribute": {`
	`  "name": attr-name,`
	`  "ns": namespace`
	`  },`
	`  "json-property": prop-name,`
	`  "field": {`
	`  "name": field-name,`
	`  "collation": uri`
	`  },`
	`  "path-index": {`
	`  "text": path-expr,`
	`  "namespaces": {`
	`  prefix: namespace-uri`
	`  }`
	`  },`
	`  "fragment-scope": scope,`
	`  "value": value-as-string,`
	`  "range-operator": operator,`
	`  "range-option": option`
	`}`

4.6.17.2 コンポーネントの説明

element、json-property、field、path-index を指定します。あるいは、element と attribute を指定して、クエリに適用するレンジ制約を定義する必要があります。これらのコンポーネントは互いに排他的です。つまり、element と attribute の組み合わせを除いて、クエリには 1 つのみが含まれる必要があります。

要素または JSON プロパティ名	必須？	説明
element	いいえ	XML 要素の記述子。要素 name と名前空間（ns）によって特定されます。name と ns の両方が必須です。element を含める場合は、json-property、field、および path-index を含められません。
attribute	いいえ	XML 属性の記述子。属性は name と名前空間（ns）によって特定します。name と ns の両方が必須です。
json-property	いいえ	JSON プロパティ名。json-property を含める場合は、element、field、および path-index を含められません。
field	いいえ	フィールド記述子。フィールド name（必須）とオプションの collation によって特定されます。データベース設定に、このフィールドの定義が含まれる必要があります。field を含める場合は、element、json-property、および path-index を含められません。
path-index	いいえ	パスレンジ式。パス式に名前空間のプレフィックスが含まれる場合は、path-index に名前空間のバインドを定義する必要があります。path-index を含める場合は、element、json-property、および field を含められません。 データベース設定に、マッチングパスレンジインデックスが含まれる必要があります。パス式と名前空間の URI は、インデックスの設定と一致する必要があります。名前空間のプレフィックスは、一致する必要はありません。 パス式は、パスレンジインデックスを定義するために使用できる XPath のサブセットに制限されます。詳細については、『Administrator's Guide』の「Understanding Path Range Indexes」を参照してください。

要素または JSON プロパティ名	必須？	説明
fragment-scope	いいえ	マッチの対象を、指定されたフラグメントスコープに制約します。使用可能な値：documents（デフォルト）または properties。詳細については、「fragment-scope」クエリオプションを参照してください。
value	いいえ	constraint-name によって識別される制約と一致する XML 要素、XML 要素の属性、JSON プロパティ、またはフィールドに対してマッチさせる値。この要素を 0 個以上含むことができます。
range-operator	いいえ	LT、LE、GT、GE、EQ、NE のいずれか。デフォルト：EQ。マッチさせる制約と value の間で満たす必要がある関係性。
range-option	いいえ	1 つあるいは複数のレンジクエリオプション。使用可能な値は、レンジクエリのタイプ（要素、パス、フィールドなど）によって異なります。詳細については、『Search Developer's Guide』の「スコア計算にレンジクエリや位置クエリを含める」を参照してください。
type	いいえ	レンジインデックスのタイプ。同じ項目に対して、異なるデータ型の複数のインデックスがある場合は必須です。
collation	いいえ	インデックスのタイプが文字列である場合に使用する、コレーションの URI。

4.6.17.3 例

次の例は、<body-color/> に要素レンジインデックスがあることを前提として、値が black である <body-color/> 要素を含んでいるドキュメントとマッチされます。

XML	JSON

すべての要素は名前空間 http://marklogic.com/appservices/search 内にあります。
 	
`<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" ]`
	`  }`
	`  }]`
	`}}`

4.6.18 value-query

XML 要素、XML 属性、JSON プロパティ、またはフィールドのテキストコンテンツの値がクエリ内の text 語句と完全に一致する場合にフラグメントとマッチさせるクエリ。マッチの方法は、text 値、データベース設定、および有効なオプションによって異なります。詳細については、「cts:element-value-query」、「cts:element-attribute-value-query」、および「cts:field-value-query」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.18.1 シンタックスの概要

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<value-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>`
`  <number>value</number>`
`  <boolean>true-or-false</boolean>`
`  <null/>`
`  <term-option>option/term-option>`
`  <weight>value</weight>`
`</value-query>`
	`"value-query": {`
	`  "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 ],`
	`  "number": [ value ],`
	`  "boolean": [ true-or-false ],`
	`  "null": { },`
	`  "term-option": [ option ],`
	`  "weight": number`
	`}`

4.6.18.2 コンポーネントの説明

`element`、`json-property`、`field` を指定します。あるいは、`element` と `attribute` を指定して、クエリに適用するレンジ制約を定義する必要があります。これらのコンポーネントは互いに排他的です。つまり、`element` と `attribute` の組み合わせを除いて、クエリには 1 つのみが含まれる必要があります。

要素または JSON プロパティ名	必須？	説明
`element`	いいえ	XML 要素の記述子。要素 `name` と名前空間（`ns`）によって特定されます。`name` と `ns` の両方が必須です。`element` を含める場合は、`json-property` を含められません。
`attribute`	いいえ	XML 属性の記述子。属性は `name` と名前空間（`ns`）によって特定します。`name` と `ns` の両方が必須です。

要素または JSON プロパティ名	必須？	説明
`json-property`	いいえ	JSON プロパティ名。`json-property` を含める場合は、`element` を含められません。
`field`	いいえ	フィールド記述子。フィールド `name`（必須）とオプションの `collation` によって特定されます。データベース設定に、このフィールドの定義が含まれる必要があります。
`fragment-scope`	いいえ	マッチの対象を、指定されたフラグメントスコープに制約します。使用可能な値：`documents`（デフォルト）または `properties`。詳細については、「`fragment-scope`」クエリオプションを参照してください。
`text`	いいえ	XML 要素、XML 要素の属性、JSON プロパティ、またはフィールド値内とマッチさせる必要があるテキスト。複数の値を指定できます。`text`、`number`、`boolean`、`null` コンポーネントのいずれもない場合、制約とマッチするすべての値が返されます。
`number`	いいえ	JSON プロパティの値またはフィールド内とマッチさせる必要がある数値。複数の値を指定できます。`text`、`number`、`boolean`、`null` コンポーネントのいずれもない場合、制約とマッチするすべての値が返されます。
`boolean`	いいえ	JSON プロパティの値またはフィールド内とマッチさせる必要があるブール型値。複数の値を指定できます。`text`、`number`、`boolean`、`null` コンポーネントのいずれもない場合、制約とマッチするすべての値が返されます。
`null`	いいえ	JSON プロパティまたはフィールド内の null 値とマッチされます。複数の値を指定できます。`text`、`number`、`boolean`、`null` コンポーネントのいずれもない場合、制約とマッチするすべての値が返されます。

要素または JSON プロパティ名	必須？	説明
term-option	いいえ	クエリに適用するタームオプション。複数のタームオプションを指定できます。オプションが値を持つ場合、term-option の値は option=value になります。例えば次のようになります。 <term-option>min-occurs=1</term-option> 詳細については、クエリの制約タイプに対応する cts クエリを参照してください。cts:element-value-query、cts:element-attribute-value-query、または cts:field-value-query を参照してください。
weight	いいえ	このクエリの重み。デフォルト：1.0。重みが大きいほど、検索結果の関連度が高くなります。重みは、-16 から 64 の間にする必要があります。64 よりも大きい値は、64 と同じものとして扱われます。絶対値が 0.0625 未満のもの（つまり -0.0625 と 0.0625 の間）は 0 として扱われます。このため、スコアに影響しません。

4.6.18.3 例

次の例は、「myFieldName」という名前で定義されたフィールドが「Jane Doe」という値を持つドキュメントとマッチされます。この例では、データベース設定でフィールド「myFieldName」が定義され、データベースに対してフィールド検索がオンになっていることを前提としています。

XML	JSON

すべての要素は名前空間 http://marklogic.com/appservices/search 内にあります。
 	
<query>
  <value-query>
  <field name="myFieldName" />
  <text>Jane Doe</text>
  </value-query>
</query>
	{
	  "query": {
	  "queries": [{
	  "value-query": {
	  "field": {
	  "name": "myFieldName",
	  },
	  "text": ["Jane Doe"],
	  }
	  }]
	}}

4.6.19 word-query

クエリで定義された制約によって識別される XML 要素、XML 属性、JSON プロパティ、またはフィールド内に、指定されたタームまたは語句を含んでいるフラグメントとマッチさせるクエリ。これは、`constraint:value` という形式を取る、制約が要素、要素属性、JSON プロパティ、またはフィールド制約になっている場合の文字列クエリと同様です。詳細については、`cts:element-word-query`、`cts:field-word-query` など、制約タイプに対応する cts word Query（ワードクエリ）関数を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.19.1 シンタックスの概要

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<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`
	`}`

4.6.19.2 コンポーネントの説明

element、json-property、field を指定します。あるいは、element と attribute を指定して、クエリに適用するレンジ制約を定義する必要があります。これらのコンポーネントは互いに排他的です。つまり、element と attribute の組み合わせを除いて、クエリには 1 つのみが含まれる必要があります。

要素または JSON プロパティ名	必須？	説明
element	いいえ	XML 要素の記述子。要素 name と名前空間（ns）によって特定されます。name と ns の両方が必須です。element を含める場合は、json-property を含められません。
attribute	いいえ	XML 属性の記述子。属性は name と名前空間（ns）によって特定します。name と ns の両方が必須です。
json-property	いいえ	JSON プロパティ名。json-property を含める場合は、element を含められません。
field	いいえ	フィールド記述子。フィールド name（必須）とオプションの collation によって特定されます。データベース設定に、このフィールドの定義が含まれる必要があります。
fragment-scope	いいえ	マッチの対象を、指定されたフラグメントスコープに制約します。使用可能な値：documents（デフォルト）または properties。詳細については、「fragment-scope」クエリオプションを参照してください。
text	いいえ	このクエリによって定義された制約とマッチされるドキュメント内に存在すべきタームまたは語句。複数の値を指定できます。いずれかの値とマッチした場合、ドキュメントがクエリとマッチします。

要素または JSON プロパティ名	必須？	説明
term-option	いいえ	クエリに適用するタームオプション。複数のタームオプションを指定できます。オプションが値を持つ場合、term-option の値は option=value になります。例えば次のようになります。 <term-option>min-occurs=1</term-option> 詳細については、次のようなクエリの制約タイプに対応する cts クエリを参照してください。 cts:element-word-query、cts:element-attribute-word-query、cts:field-word-query など。
weight	いいえ	このクエリの重み。デフォルト：1.0。重みが大きいほど、検索結果の関連度が高くなります。重みは、-16 から 64 の間にする必要があります。64 よりも大きい値は、64 と同じものとして扱われます。絶対値が 0.0625 未満のもの（つまり -0.0625 と 0.0625 の間）は 0 として扱われます。このため、スコアに影響しません。

4.6.19.3 例

次の例は、black という単語を含んでいる <body-color/> 要素を含んでいるドキュメントとマッチされます。

XML	JSON

すべての要素は名前空間 http://marklogic.com/appservices/search 内にあります。
 	
<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"],
	  }
	  }]
	}}

4.6.20 geo-elem-query

クエリで定義された位置情報要素制約とマッチするドキュメントを返すクエリ。詳細については、「cts:element-geospatial-query」、「cts:element-child-geospatial-query」、または「位置検索アプリケーション」（347 ページ）を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.20.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
http://marklogic.com/appservices/search
内にあります。
 	
`<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>`
`</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 ]`
	`}`

4.6.20.2 コンポーネントの説明

位置クエリは、point、box、circle、および polygon XML 子要素または JSON サブオブジェクトによって記述された 1 つあるいは複数の地点または地域を含んでいます。位置データを含んでいる要素ペアは、parent、lat（緯度）および lon（経度）によって記述されます。詳細については、「位置検索アプリケーション」（347 ページ）を参照してください。

要素または JSON プロパティ名	必須？	説明
parent	いいえ	オプション。位置データを含む要素の親要素。要素の name と名前空間（ns）によって特定されます。name と ns の両方が必須です。
element	はい	位置データを含む要素。name と名前空間（ns）によって特定されます。name と ns の両方が必須です。
fragment-scope	いいえ	マッチの対象を、指定されたフラグメントスコープに制約します。使用可能な値：documents（デフォルト）または properties。詳細については、「fragment-scope」クエリオプションを参照してください。
geo-option	いいえ	クエリに適用する位置オプション。複数のオプションを指定できます。オプションが値を持つ場合、geo-option の値は *option=value* という形式を取ります。例えば次のようになります：<geo-option>units=miles</geo-option>。詳細については、「cts:element-geospatial-query」または「cts:element-child-geospatial-query」を参照してください。

要素または JSON プロパティ名	必須？	説明
point	いいえ	0 個以上の地理的な地点。それぞれ、latitude（緯度）と longitude（経度）によって定義されます。クエリには、0 個以上の地点を入れることができます。
box	いいえ	0 個以上の四角形の地域。それぞれ、次の 4 つの地点によって定義されます：north、south、east、および west。
circle	いいえ	0 個以上の円。それぞれ、radius（半径）と point（中心）によって定義されます。
polygon	いいえ	0 個以上の多角形。それぞれは一連の point（点）として表現されます。

4.6.20.3 例

次の例は、2 つの多角形のいずれかに含まれている地点とマッチされます。クエリで parent と element によって定義されている制約は、クエリで定義されている地域とマッチさせる地点を構築する方法を定義しています。

形式	クエリ
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>

形式	クエリ

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>

4.6.21 geo-elem-pair-query

クエリで定義された位置情報 XML 要素制約とマッチするドキュメントを返すクエリ。詳細については、「`cts:element-pair-geospatial-query`」および「位置検索アプリケーション」（347 ページ）を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.21.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
http://marklogic.com/appservices/search
内にあります。
 	
`<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>`
`</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 ]`
	`}`

4.6.21.2 コンポーネントの説明

位置クエリは、point、box、circle、および polygon XML 子要素または JSON サブオブジェクトによって記述された 1 つあるいは複数の地点または地域を含んでいます。位置データを含んでいる要素ペアは、parent、lat（緯度）および lon（経度）によって記述されます。

要素または JSON プロパティ名	必須？	説明
parent	いいえ	位置データを含んでいる要素。要素の name と名前空間（ns）によって特定されます。name と ns の両方が必須です。
lat	はい	緯度データを含んでいる XML 要素。name と名前空間（ns）によって特定されます。name と ns の両方が必須です。
lon	はい	経度データを含んでいる XML 要素。name と名前空間（ns）によって特定されます。name と ns の両方が必須です。
fragment-scope	いいえ	マッチの対象を、指定されたフラグメントスコープに制約します。使用可能な値：documents（デフォルト）または properties。詳細については、「fragment-scope」クエリオプションを参照してください。
geo-option	いいえ	クエリに適用する位置オプション。複数のオプションを指定できます。オプションが値を持つ場合、geo-option の値は *option=value* という形式を取ります。例えば次のようになります：<geo-option>units=miles</geo-option>。詳細については、「cts:element-pair-geospatial-query」を参照してください。

要素または JSON プロパティ名	必須？	説明
point	いいえ	0 個以上の地理的な地点。それぞれ、latitude（緯度）と longitude（経度）によって定義されます。クエリには、0 個以上の地点を入れることができます。
box	いいえ	0 個以上の四角形の地域。それぞれ、次の 4 つの地点によって定義されます：north、south、east、および west。
circle	いいえ	0 個以上の円。それぞれ、radius（半径）と point（中心）によって定義されます。
polygon	いいえ	0 個以上の多角形。それぞれは一連の point（点）として表現されます。

4.6.21.3 例

次の例は、2 つの多角形のいずれかに含まれている地点とマッチされます。クエリで parent、lat（緯度）および lon（経度）によって定義されている制約は、クエリで定義されている地域とマッチさせる地点を構築する方法を定義しています。

形式	クエリ
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>

形式	クエリ

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 }
	]
	}
	]
	}
	}]
	}}

4.6.22 geo-attr-pair-query

クエリで定義された位置情報要素制約とマッチするドキュメントを返すクエリ。詳細については、「`cts:element-attribute-pair-geospatial-query`」または「位置検索アプリケーション」（347 ページ）を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.22.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
http://marklogic.com/appservices/search
内にあります。
 	
`<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>`
`</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 ]`
	`}`

4.6.22.2 コンポーネントの説明

位置クエリは、point、box、circle、および polygon XML 子要素または JSON サブオブジェクトによって記述された 1 つあるいは複数の地点または地域を含んでいます。位置データを含んでいる要素ペアは、parent、lat（緯度）および lon（経度）によって記述されます。詳細については、「位置検索アプリケーション」（347 ページ）を参照してください。

要素または JSON プロパティ名	必須？	説明
parent	はい	位置データを保持する lat（緯度）および lon（経度）属性を含んでいる要素。要素の name と名前空間（ns）によって特定されます。name と ns の両方が必須です。
lat	はい	緯度データを含んでいる属性の名前。name と名前空間（ns）によって特定されます。name と ns の両方が必須です。
lon	はい	経度データを含んでいる属性の名前。name と名前空間（ns）によって特定されます。name と ns の両方が必須です。
fragment-scope	いいえ	マッチの対象を、指定されたフラグメントスコープに制約します。使用可能な値：documents（デフォルト）または properties。詳細については、「fragment-scope」クエリオプションを参照してください。
geo-option	いいえ	クエリに適用する位置オプション。複数のオプションを指定できます。オプションが値を持つ場合、geo-option の値は *option=value* という形式を取ります。例えば次のようになります：<geo-option>units=miles</geo-option>。詳細については、「cts:element-attribute-pair-geospatial-query」を参照してください。

要素または JSON プロパティ名	必須？	説明
point	いいえ	0 個以上の地理的な地点。それぞれ、latitude（緯度）と longitude（経度）によって定義されます。クエリには、0 個以上の地点を入れることができます。
box	いいえ	0 個以上の四角形の地域。それぞれ、次の 4 つの地点によって定義されます：north、south、east、および west。
circle	いいえ	0 個以上の円。それぞれ、radius（半径）と point（中心）によって定義されます。
polygon	いいえ	0 個以上の多角形。それぞれは一連の point（点）として表現されます。

4.6.22.3 例

次の例は、2 つの多角形のいずれかに含まれている地点とマッチされます。クエリで parent、lat（緯度）および lon（経度）によって定義されている制約は、クエリで定義されている地域とマッチさせる地点を構築する方法を定義しています。

形式	クエリ
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>

形式	クエリ

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-path-query

指定された XPath 式とマッチする XML 要素、XML 属性、または JSON プロパティ内で出現した場合にのみ、クエリで定義された位置制約とマッチするドキュメントを返すクエリ。詳細については、「`cts:path-geospatial-query`」または「位置検索アプリケーション」（347 ページ）を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.23.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
http://marklogic.com/appservices/search
内にあります。
 	
`<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>`
`</geo-path-query>`
	`"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*` ]`
	`}`

4.6.23.2 コンポーネントの説明

位置クエリは、`point`、`box`、`circle`、および`polygon` XML 子要素または JSON サブオブジェクトによって記述された 1 つあるいは複数の地点または地域を含んでいます。位置データを含んでいる要素または属性は、`path-index` によって記述されます。詳細については、「位置検索アプリケーション」（347 ページ）を参照してください。

要素または JSON プロパティ名	必須？	説明
path-index	はい	指定の地域内に含まれている地点を表現するコンテンツを持つ要素または属性とマッチさせるパスレンジ式。パス式に名前空間のプレフィックスが含まれる場合は、path-index に名前空間のバインドを定義する必要があります。 データベース設定に、マッチングパスレンジインデックスが含まれる必要があります。パス式と名前空間の URI は、インデックスの設定と一致する必要があります。名前空間のプレフィックスは、一致する必要はありません。 パス式は、パスレンジインデックスを定義するために使用できる XPath のサブセットに制限されます。詳細については、『Administrator's Guide』の「Understanding Path Range Indexes」を参照してください。
fragment-scope	いいえ	マッチの対象を、指定されたフラグメントスコープに制約します。使用可能な値：documents（デフォルト）または properties。詳細については、「fragment-scope」クエリオプションを参照してください。
geo-option	いいえ	クエリに適用する位置オプション。複数のオプションを指定できます。オプションが値を持つ場合、geo-option の値は *option=value* という形式を取ります。例えば次のようになります：<geo-option>units=miles</geo-option>。詳細については、「cts:element-attribute-pair-geospatial-query」を参照してください。

要素または JSON プロパティ名	必須？	説明
point	いいえ	0 個以上の地理的な地点。それぞれ、latitude（緯度）と longitude（経度）によって定義されます。クエリには、0 個以上の地点を入れることができます。
box	いいえ	0 個以上の四角形の地域。それぞれ、次の 4 つの地点によって定義されます：north、south、east、および west。
circle	いいえ	0 個以上の円。それぞれ、radius（半径）と point（中心）によって定義されます。
polygon	いいえ	0 個以上の多角形。それぞれは一連の point（点）として表現されます。

4.6.23.3 例

次の例は、2 つの多角形のいずれかに含まれている地点とマッチされます。クエリで path-index によって定義されている制約は、クエリで定義されている地域とマッチさせる地点を構築する方法を定義しています。

形式	クエリ
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>

形式	クエリ

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 }
	] }
	]
	}
	}}

4.6.24 geo-json-property-query

指定された JSON プロパティ内で出現した場合にのみ、クエリで定義された位置制約とマッチするドキュメントを返すクエリ。詳細については、「cts:json-property-geospatial-query」、「cts:json-property-child-geospatial-query」、または「位置検索アプリケーション」（347 ページ）を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.24.1 シンタックスの概要

XML	JSON
すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。 `<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>` `</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*`]` `}`

4.6.24.2 コンポーネントの説明

位置クエリは、point、box、circle、および polygon XML 子要素または JSON サブオブジェクトによって記述された 1 つあるいは複数の地点または地域を含んでいます。位置データを含んでいる要素ペアは、parent、lat（緯度）および lon（経度）によって記述されます。詳細については、「位置検索アプリケーション」（347 ページ）を参照してください。

要素または JSON プロパティ名	必須？	説明
parent-property	いいえ	オプション。位置データを含んでいるプロパティの親プロパティ。名前によって特定されます。
json-property	はい	位置データを含んでいるプロパティの名前。
fragment-scope	いいえ	マッチの対象を、指定されたフラグメントスコープに制約します。使用可能な値：documents（デフォルト）または properties。詳細については、「fragment-scope」クエリオプションを参照してください。
geo-option	いいえ	クエリに適用する位置オプション。複数のオプションを指定できます。オプションが値を持つ場合、geo-option の値は *option=value* という形式を取ります。例えば次のようになります：<geo-option>units=miles</geo-option>。詳細については、「cts:json-property-geospatial-query」または「cts:json-property-child-geospatial-query」を参照してください。
point	いいえ	0 個以上の地理的な地点。それぞれ、latitude（緯度）と longitude（経度）によって定義されます。クエリには、0 個以上の地点を入れることができます。
box	いいえ	0 個以上の四角形の地域。それぞれ、次の 4 つの地点によって定義されます：north、south、east、および west。
circle	いいえ	0 個以上の円。それぞれ、radius（半径）と point（中心）によって定義されます。
polygon	いいえ	0 個以上の多角形。それぞれは一連の point（点）として表現されます。

4.6.24.3 例

次の例は、2 つの多角形のいずれかに含まれている地点とマッチされます。
`parent-property` と `json-property` によってクエリで定義されている JSON プロパティは、クエリで定義されている地域とマッチさせる地点を構築する方法を定義しています。

形式	クエリ
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>

形式	クエリ

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>

4.6.25 geo-json-property-pair-query

指定された XPath 式とマッチする JSON プロパティ内で出現した場合にのみ、クエリで定義された位置制約とマッチするドキュメントを返すクエリ。詳細については、「`cts:json-property-pair-geospatial-query`」または「位置検索アプリケーション」（347 ページ）を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.25.1 シンタックスの概要

XML	JSON
すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。 `<geo-json-property-pair-query>` `  <parent-property>`*name*`</parent-property>` `  <lat-property>`*number*`</lat-property>` `  <lon-property>`*number*`</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>` `</geo-json-property-pair-query>`	`"geo-json-property-pair-query": {` `  "parent-property":` *name*`,` `  "lat-property":` *number*`,` `  "lon-property":` *number*`,` `  "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* `]` `}`

4.6.25.2 コンポーネントの説明

位置クエリは、`point`、`box`、`circle`、および `polygon` XML 子要素または JSON サブオブジェクトによって記述された 1 つあるいは複数の地点または地域を含んでいます。位置データを含んでいる要素ペアは、`parent`、`lat`（緯度）および `lon`（経度）によって記述されます。詳細については、「位置検索アプリケーション」（347 ページ）を参照してください。

要素または JSON プロパティ名	必須？	説明
parent-property	はい	lat-property と lon-property を含んでいる JSON プロパティの名前。
lat-property	はい	緯度データを含んでいる JSON プロパティの名前。
lon-property	はい	経度データを含んでいる JSON プロパティの名前。
fragment-scope	いいえ	マッチの対象を、指定されたフラグメントスコープに制約します。使用可能な値：documents（デフォルト）または properties。詳細については、「fragment-scope」クエリオプションを参照してください。
geo-option	いいえ	クエリに適用する位置オプション。複数のオプションを指定できます。オプションが値を持つ場合、geo-option の値は *option=value* という形式を取ります。例えば次のようになります：<geo-option>units=miles</geo-option>。詳細については、「cts:json-property-pair-geospatial-query」を参照してください。
point	いいえ	0 個以上の地理的な地点。それぞれ、latitude（緯度）と longitude（経度）によって定義されます。クエリには、0 個以上の地点を入れることができます。
box	いいえ	0 個以上の四角形の地域。それぞれ、次の 4 つの地点によって定義されます：north、south、east、および west。
circle	いいえ	0 個以上の円。それぞれ、radius（半径）と point（中心）によって定義されます。
polygon	いいえ	0 個以上の多角形。それぞれは一連の point（点）として表現されます。

4.6.25.3 例

次の例は、2 つの多角形のいずれかに含まれている地点とマッチされます。クエリで `parent`、`lat`（緯度）および `lon`（経度）によって定義されている制約は、クエリで定義されている地域とマッチさせる地点を構築する方法を定義しています。

形式	クエリ
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>

形式	クエリ

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>

4.6.26 range-constraint-query

事前定義されたレンジ制約を適用し、その結果を指定された値と比較するクエリ。詳細については、「制約オプション」（257 ページ）および XQuery 関数「`cts:element-range-query`」、「`cts:element-attribute-range-query`」、「`cts:field-range-query`」、および「`cts:path-range-query`」を参照してください。

`range-constraint-query` は、形式が `constraint:value` または `constraint LE value` である文字列クエリ式と等価です。名前付き制約は、レンジインデックスが利用されている必要があります。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.26.1 シンタックスの概要

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<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*` ]`
	`}`

4.6.26.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`constraint-name`	はい	グローバルなクエリオプションまたはクエリ固有のクエリオプションで定義された制約の名前。
`value`	はい	`constraint-name` によって識別される制約と一致する XML 要素、XML 要素の属性、JSON プロパティ、またはフィールドに対してマッチさせる値。この要素を 0 個以上含むことができます。
`range-operator`	いいえ	`LT`、`LE`、`GT`、`GE`、`EQ`、`NE` のいずれか。デフォルト：`EQ`。`constraint-name matches` と `value` の間で満たす必要がある、マッチの関係性。
`range-option`	いいえ	1 つあるいは複数のレンジクエリオプション。使用可能な値は、レンジクエリのタイプ（要素、パス、フィールドなど）によって異なります。詳細については、『Search Developer's Guide』の「スコア計算にレンジクエリや位置クエリを含める」を参照してください。

4.6.26.3 例

次の例は、<body-color/> に要素レンジインデックスがあることを前提として、値が black である <body-color/> 要素を含んでいるドキュメントとマッチされます。

形式	クエリ
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"]
	  }
	  }]
	}}

4.6.27 value-constraint-query

XML 要素、XML 属性、JSON プロパティ、またはフィールドのコンテンツの値がクエリ内の text、number、boolean、または null 値と完全に一致する場合にフラグメントとマッチさせるクエリ。マッチの方法は、値、データベース設定、および有効なオプションによって異なります。要素、属性、プロパティ、またはフィールドは、クエリオプションで定義された値制約によって特定されます。これは、constraint:value という形式を取る、文字列クエリのタームと同様です。詳細については、「cts:element-value-query、cts:element-attribute-value-query」、「cts:field-value-query」、「cts:json-property-value-query」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.27.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
http://marklogic.com/appservices/search
内にあります。
 	
<value-constraint-query>
<constraint-name>*name*</constraint-name>
<text>*name*</text>
<number>*value*</number>
<boolean>*true-or-false*</boolean>
<null/>
<weight>*value*</weight>
</value-constraint-query>
	"value-constraint-query":{
	"constraint-name":"*name*",
	"text":[*name*],
	"number":[*value*],
	"boolean":[*true-or-false*],
	"null":XXX what does this look like?XXX,
	"weight":*number*
	}

4.6.27.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`constraint-name`	はい	グローバルなクエリオプションまたはクエリ固有のクエリオプションで定義された値制約の名前。
`text`	いいえ	マッチさせる値。複数の値を指定できます。`text`、`number`、`boolean`、`null` コンポーネントのいずれもない場合、制約とマッチするすべての値が返されます。
`number`	いいえ	マッチさせる数値。JSON ドキュメントにのみ適用可能です。複数の値を指定できます。`text`、`number`、`boolean`、`null` コンポーネントのいずれもない場合、制約とマッチするすべての値が返されます。
`boolean`	いいえ	マッチさせるブール型値。JSON ドキュメントにのみ適用可能です。複数の値を指定できます。`text`、`number`、`boolean`、`null` コンポーネントのいずれもない場合、制約とマッチするすべての値が返されます。
`null`	いいえ	null 値とマッチされます。JSON ドキュメントにのみ適用可能です。複数の値を指定できます。`text`、`number`、`boolean`、`null` コンポーネントのいずれもない場合、制約とマッチするすべての値が返されます。
`weight`	いいえ	このクエリの重み。デフォルト：1.0。重みが大きいほど、検索結果の関連度が高くなります。重みは、-16 から 64 の間にする必要があります。64 よりも大きい値は、64 と同じものとして扱われます。絶対値が 0.0625 未満のもの（つまり -0.0625 と 0.0625 の間）は 0 として扱われます。このため、スコアに影響しません。

4.6.27.3 例

次の例は、“myFieldName” という名前で定義されたフィールドが “Jane Doe” という値を持つドキュメントとマッチされます。この例では、データベース設定でフィールド「myFieldName」が定義され、データベースに対してフィールド検索がオンになっていることを前提としています。

形式	クエリ
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"
	  }
	  }]
	}}

4.6.28 word-constraint-query

XML 要素、XML 属性、JSON プロパティ、または指定された制約によって識別されるフィールド内に、指定されたタームまたは語句を含んでいるフラグメントとマッチさせるクエリ。これは、`constraint:value` という形式を取る、制約がワード制約になっている場合の文字列クエリと同様です。詳細については、「`cts:word-query`」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.28.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<word-constraint-query>`
`  <constraint-name>`*name*`</constraint-name>`
`  <text>`*name*`</text>`
`  <weight>`*value*`</weight>`
`</word-constraint-query>`
	`"word-constraint-query":{`
	`  "constraint-name":"`*name*`",`
	`  "text":[ `*name*` ],`
	`  "weight":`*number*
	`}`

4.6.28.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`constraint-name`	はい	グローバルなクエリオプションまたはクエリ固有のクエリオプションで定義されたワード制約の名前。
`text`	いいえ	`constraint-name` によって定義された制約とマッチするドキュメント内に出現する必要があるタームまたは語句。複数の値を指定できます。いずれかの値とマッチした場合、ドキュメントがクエリとマッチします。
`weight`	いいえ	このクエリの重み。デフォルト：1.0。重みが大きいほど、検索結果の関連度が高くなります。重みは、-16 から 64 の間にする必要があります。64 よりも大きい値は、64 と同じものとして扱われます。絶対値が 0.0625 未満のもの（つまり -0.0625 と 0.0625 の間）は 0 として扱われます。このため、スコアに影響しません。

4.6.28.3 例

次の例は、black という単語を含んでいる <body-color/> 要素を含んでいるドキュメントとマッチされます。

形式	クエリ
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"
	  }
	  }]
	}}

4.6.29 collection-constraint-query

事前定義された制約を適用し、その結果を指定された値と比較するクエリ。詳細については、「cts:collection-query」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.29.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<collection-constraint-query>`
`  <constraint-name>`*name*`</constraint-name>`
`  <uri>`*collection-uri*`</value>`
`</collection-constraint-query>`
	`"collection-constraint-query": {`
	`  "constraint-name": "`*name*`",`
	`  "uri": [ `*collection-uri*` ]`
	`}`

4.6.29.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`constraint-name`	はい	グローバルなクエリオプションまたはクエリ固有のクエリオプションで定義されたコレクション制約の名前。
`uri`	いいえ	マッチさせる 1 つあるいは複数のコレクションの URI。

4.6.29.3 例

次の例は、コレクション reports またはコレクション analysis 内のドキュメントとマッチされます。

形式	クエリ
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"
	  }
	  }]
	}}

4.6.30 container-constraint-query

指定された制約を満たし、指定されたサブクエリとマッチする要素、属性、またはプロパティを内容として持つ XML 要素または JSON プロパティとマッチさせるクエリ。サブクエリによってマッチングコンテナとそのすべての子孫が考慮されます。詳細については、「cts:element-query」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.30.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
http://marklogic.com/appservices/search
内にあります。
 	
<container-constraint-query>
<constraint-name>*name*</constraint-name>
anyQueryType
</element-constraint-query>
	"container-constraint-query":{
	"constraint-name":"*name*",
	anyQueryType
	}

4.6.30.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
constraint-name	はい	グローバルなクエリオプションまたはクエリ固有のクエリオプションで定義された制約の名前。
anyQueryType	はい	constraint-name によって識別された制約とマッチするコンテナに対して実行するクエリ。

4.6.30.3 例

次の例は、`black` というタームを含んでいる `<body-color/>` 要素の出現箇所とマッチされます。

形式	クエリ

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

4.6.31 element-constraint-query

指定された要素制約を満たし、指定されたサブクエリとマッチするサブ要素または属性を持つ要素とマッチさせるクエリ。サブクエリによってマッチング要素とそのすべての子孫が考慮されます。詳細については、「cts:element-query」を参照してください。

> **注：** このクエリタイプの使用は推奨されません。代わりに、「container-constraint-query」を使用してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.31.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<element-constraint-query>`
`  <constraint-name>`*`name`*`</constraint-name>`
`  `*`anyQueryType`*
`</element-constraint-query>`
	`"element-constraint-query":{`
	`  "constraint-name":"`*`name`*`",`
	`  `*`anyQueryType`*
	`}`

4.6.31.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`constraint-name`	はい	グローバルなクエリオプションまたはクエリ固有のクエリオプションで定義されたコンテナ制約の名前。
`anyQueryType`	はい	`constraint-name` によって識別された制約とマッチする要素に対して実行するクエリ。

4.6.31.3 例

次の例は、`black` というタームを含んでいる `<body-color/>` 要素の出現箇所とマッチされます。

形式	クエリ

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

4.6.32 properties-constraint-query

指定されたプロパティ制約とマッチするプロパティを持つドキュメントとマッチさせるクエリ。ただし、マッチさせるプロパティも、指定されたクエリとマッチする必要があります。詳細については、「cts:properties-fragment-query」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.32.1 シンタックスの概要

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<properties-constraint-query>`
`  <constraint-name>`*name*`</constraint-name>`
`  `*anyQueryType*
`</properties-constraint-query>`
	`"properties-constraint-query":{`
	`  "constraint-name":"`*name*`",`
	`  `*anyQueryType*
	`}`

4.6.32.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`constraint-name`	はい	グローバルなクエリオプションまたはクエリ固有のクエリオプションで定義されたプロパティ制約の名前。
anyQueryType	はい	`constraint-name` によって識別された制約とマッチするプロパティに対して実行するクエリ。

4.6.32.3 例

次の例は、dog というタームを含んでいるプロパティフラグメントを持つドキュメントとマッチされます。

形式	クエリ
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"]
	  }
	  }
	  }]
	}}

4.6.33 custom-constraint-query

カスタムの XQuery 拡張関数によって構築されたクエリ。指定された条件を使用します。詳細については、「カスタム制約の作成」（34 ページ）を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.33.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<custom-constraint-query>`
`  <constraint-name>`*name*`</constraint-name>`
`  <text>term</text>`
`</custom-constraint-query>`
	`"custom-constraint-query": {`
	`  "constraint-name": "`*name*`",`
	`  "text": [ `*term*` ]`
	`}`

4.6.33.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`constraint-name`	はい	グローバルなクエリオプションまたはクエリ固有のクエリオプションで定義されたカスタム制約の名前。
`text`	いいえ	`constraint-name` によって識別された制約とマッチするフラグメントに対して実行するクエリ。

4.6.33.3 例

次の例は、“part:book” という文字列クエリと等価です。ただし、part は、クエリオプションで定義されているカスタム制約の名前です。

形式	クエリ
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"
	  }
	  }]
	}}

4.6.34 geospatial-constraint-query

指定された位置制約とマッチするドキュメントを返すクエリ。マッチさせるフラグメントも、位置クエリとマッチする必要があります。詳細については、「位置検索アプリケーション」（347 ページ）を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.34.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
http://marklogic.com/appservices/search
内にあります。
 	
`<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>` ターム `</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*` ]`
	`}`

4.6.34.2 コンポーネントの説明

位置制約クエリは、point、box、circle、および polygon XML 子要素または JSON サブオブジェクトによって記述された 1 つあるいは複数の地点または地域を含んでいます。

要素または JSON プロパティ名	必須？	説明
constraint-name	はい	グローバルなクエリオプションまたはクエリ固有のクエリオプションで定義されたカスタム制約の名前。
point	いいえ	0 個以上の地理的な地点。それぞれ、latitude（緯度）と longitude（経度）によって定義されます。クエリには、0 個以上の地点を入れることができます。
box	いいえ	0 個以上の四角形の地域。それぞれ、次の 4 つの地点によって定義されます：north、south、east、および west。
circle	いいえ	0 個以上の円。それぞれ、radius（半径）と point（中心）によって定義されます。
polygon	いいえ	0 個以上の多角形。それぞれは一連の point（点）として表現されます。

4.6.34.3 例

次の例は、2 つの多角形のいずれかに含まれている地点とマッチされます。クエリオプション内の `my-geo-elem-pair` 制約は、地点を構築する方法を定義します。

形式	クエリ
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>

形式	クエリ

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

4.6.35 lsqt-query

Last Stable Query Time（LSQT）よりも前、または LSQT の前の特定のタイムスタンプよりも前のドキュメントを返すクエリ。詳細については、「`cts:lsqt-query`」または『Temporal Developer's Guide』の「Searching Temporal Documents」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.35.1 シンタックスの概要

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<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`* `]`
	`}`

4.6.35.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`temporal-collection`	はい	テンポラルコレクションの名前。
`timestamp`	いいえ	システム開始時刻がこの値以前であるテンポラルドキュメントのみを返します。LSQT よりも大きいタイムスタンプは拒否されます。デフォルト：その名前が付けられたテンポラルコレクションの LSQT。
`weight`	いいえ	このクエリの重み。デフォルト：1.0。重みが大きいほど、検索結果の関連度が高くなります。重みは、-16 から 64 の間にする必要があります。64 よりも大きい値は、64 と同じものとして扱われます。絶対値が 0.0625 未満のもの（つまり -0.0625 と 0.0625 の間）は 0 として扱われます。このため、スコアに影響しません。
`temporal-option`	いいえ	クエリに適用するテンポラルオプション。複数のオプションを指定できます。オプションが値を持つ場合、`temporal-option` の値は `option=value` になります。例えば次のようになります：`<temporal-option>score-function=linear</temporal-option>`。使用可能なオプションについては、「`cts:lsqt-query`」を参照してください。

4.6.35.3 例

次の例は、LSQT よりも前のテンポラルコレクション「myTemporalCollection」内のドキュメントを返します。

形式	クエリ

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

4.6.36 period-compare-query

2 つのテンポラル（時間）軸に関して指定された関係性に該当するドキュメントとマッチさせるクエリ。詳細については、「`cts:period-compare-query`」または『Temporal Developer's Guide』の「Searching Temporal Documents」を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.36.1 シンタックスの概要

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<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`*` ]`
	`}`

4.6.36.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`axis1`	はい	1 番目のテンポラル軸の名前。
`temporal-operator`	はい	2 つの軸に適用される比較。演算子名のリストについては、「`cts:period-compare-query`」および『Temporal Developer's Guide』の「Period Comparison Operators」を参照してください。
`axis2`	はい	2 番目のテンポラル軸の名前。
`temporal-option`	いいえ	クエリに適用するテンポラルオプション。複数のオプションを指定できます。オプションが値を持つ場合、`temporal-option` の値は `option=value` になります。例えば次のようになります：`<temporal-option>score-function=linear</temporal-option>`。使用可能なオプションについては、「`cts:lsqt-query`」を参照してください。

4.6.36.3 例

次の例は、「system」軸によって定義される期間内に、「valid」という名前の軸によって定義される期間がある場合に、データベース内にあるドキュメントとマッチされます。

形式	クエリ
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"
	  }
	  }]
	}}

4.6.37 period-range-query

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.37.1 シンタックスの概要

XML	JSON

すべての要素は名前空間 `http://marklogic.com/appservices/search` 内にあります。
 	
`<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>`
`</period-range-query>`
	`"period-range-query": {`
	`  "axis": [ `*`name`*` ],`
	`  "temporal-operator": `*`string`*`,`
	`  "period": [`
	`  {`
	`  "period-start": `*`string`*`,`
	`  "period-end": `*`string`*
	`  }`
	`  ],`
	`  "temporal-option": [ `*`string`*` ]`
	`}`

4.6.37.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`axis`	はい	テンポラル軸の名前。複数の軸名を指定できます。
`temporal-operator`	はい	軸と期間に適用する比較操作。演算子名のリストについては、「`cts:period-range-query`」および『Temporal Developer's Guide』の「Period Comparison Operators」を参照してください。
`period`	はい	マッチさせる 1 つあるいは複数の期間。複数の期間が指定されている場合、いずれかの値がマッチする場合にクエリはマッチします。
`temporal-option`	いいえ	クエリに適用するテンポラルオプション。複数のオプションを指定できます。オプションが値を持つ場合、`temporal-option` の値は `option=value` になります。例えば次のようになります：`<temporal-option>score-function=linear</temporal-option>`。使用可能なオプションについては、「`cts:lsqt-query`」を参照してください。

4.6.37.3 例

次の例は、有効な終了時刻が 14:00 よりも前であるテンポラルドキュメントとマッチされます。

形式	クエリ
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>9999-12-31T23:59:59.99Z</period-end> </period> </period-range-query> </query>

JSON	{"query": {
	  "queries": [{
	  "period-range-query": {
	  "axis": ["valid"],
	  "temporal-operator": "aln_before",
	  "period": [
	  {
	  "period-start": "2014-04-03T14:00:00",
	  "period-end": "9999-12-31T23:59:59.99Z"
	  }
	  ]
	  }
	  }]
	}}

4.6.38 operator-state

この構造化クエリのコンポーネントは、文字列クエリの文法によって定義されているカスタムランタイム設定演算子の状態を設定します。詳細については、「演算子オプション」（268 ページ）を参照してください。

- シンタックスの概要
- コンポーネントの説明
- 例

4.6.38.1 シンタックスの概要

XML	JSON

すべての要素は名前空間
`http://marklogic.com/appservices/search`
内にあります。
 	
`<operator-state>`
`  <operator-name>`*name*`</operator-name>`
`  <state-name>`*state*`</state-name>`
`</operator-state>`
	`"operator-state": {`
	`  "operator-name": "`*name*`",`
	`  "state-name": "`*state*`"`
	`}`

4.6.38.2 コンポーネントの説明

要素または JSON プロパティ名	必須？	説明
`operator-state`	はい	`<operator/>` クエリオプションによって定義されているカスタムランタイム設定演算子の名前。
`state-name`	はい	この演算子によって認識される状態の名前。

4.6.38.3 例

次の例は、クエリオプションによって定義されているカスタムの `sort` 演算子の使用方法を示しています。この例の構造化クエリは、文字列クエリ「`sort:date`」と等価です。詳細については、「演算子オプション」（268 ページ）を参照してください。

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

形式	クエリ

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

5.0 Query By Example を使用した検索

この章では、Query By Example（QBE）を使用して検索を実行する方法について説明します。QBE は、マッチの対象となるドキュメントの構造に合わせてモデル化された構造を持つクエリです。REST API および Java API で、QBE を使用して XML および JSON ドキュメントを検索できます。Node.js クライアント API で、QBE を使用して JSON ドキュメントを検索できます。

この章は、次のセクションから構成されます。

- QBE の概要
- 例
- QBE のサブクエリタイプの概要
- 検索条件のクイックリファレンス
- QBE 構造リファレンス
- インデックス付けがクエリに与える影響
- QBE へのオプションの追加
- 検索結果のカスタマイズ
- ドキュメントタイプによる検索のスコープ設定
- QBE の複合クエリへの変換

サポートする API の詳細については、『Java Application Developer's Guide』および『REST Application Developer's Guide』を参照してください。

5.1 QBE の概要

Query By Example（QBE）のシンプルで直観的なシンタックスを使用すると、「類似のドキュメント」を探すクエリのプロトタイプが素早く作成できます。これは、QBE の検索条件が、データベース内のドキュメントの構造と類似しているためです。最もシンプルな形式では、QBE は、ドキュメント内の 1 つあるいは複数の XML 要素、XML 要素の属性、または JSON プロパティをモデリングします。

例えば、ドキュメントに author という XML 要素または JSON プロパティが含まれる場合、次の QBE を使用して、author 値が「Mark Twain」であるドキュメントを探すことができます。

形式	例
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" }
	}

QBE は常に、検索条件を定義する query コンポーネントを含んでいます。QBE には、オプションの response コンポーネントを含めることができます。このコンポーネントにより、検索結果、および検索動作を制御するフラグとオプションをカスタマイズできます。詳細については、「QBE 構造リファレンス」（211 ページ）を参照してください。

QBE では、次を含む、search API の多くの強力な機能を公開しています。

- ドキュメントの構造に基づいた検索条件
- 論理演算子。AND、OR、NOT、および NEAR 演算子を使用して、複雑な複合クエリを作成します。
- 比較演算子。比較演算子（未満、より大きい、等しくないなど）を使用して、XML 要素、XML 属性、および JSON プロパティの値をチェックするレンジクエリを作成します。
- 値または単語（ワード）によるクエリ。値そのものとマッチさせるか、含まれているコンテンツのサブセットとしてマッチさせるかを選択します。
- 検索結果のカスタマイズ。検索によって返される結果およびスニペットに含める内容を制御します。
- 検索動作を制御するためのオプション。オプションおよびフラグを使用して、大文字 / 小文字の区別、重み、出現数など、クエリの動作を制御します。

QBE を使用すると、データベースのインデックスを一切作成せずにクエリのプロトタイプを作成できますが、それによりパフォーマンスへの影響があります。詳細については、「インデックス付けがクエリに与える影響」（224 ページ）を参照してください。

この章では、QBE のシンタックスとセマンティックについて扱います。MarkLogic の次の API で、QBE を使用して XML および JSON ドキュメントを検索できます。

API	詳細情報
Node.js クライアント API (JSON のみ)	『Node.js Application Developer's Guide』の「Searching with Query By Example」
Java クライアント API	『Java Application Developer's Guide』の「Prototype a Query Using Query By Example」
REST クライアント API	『REST Application Developer's Guide』の「Using Query By Example to Prototype a Query」

より高度な検索機能を利用する必要がある場合は、API で QBE を複合クエリに変換し、これを基盤として利用します。詳細については、クライアント API のドキュメントを参照してください。

5.1.1 ドキュメントの構造に基づいた検索条件

QBE では、データベース内のドキュメントの一部分によく似た XML 要素、XML 要素の属性、または JSON プロパティ形式の検索条件を使用します。

例えば、データベースに次の形式のドキュメントが含まれているとします。

形式	ドキュメントの例

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" }`
	`  ]`
	`} }`

この場合、author、format（書籍の形態）をモデリングする条件を作成することで、特定の作者によるすべてのペーパーバックを検索する QBE を作成できます。次の QBE は、Mark Twain によるすべてのペーパーバックを探します。

形式	例
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"
	  }
	} }

デフォルトでは、条件内のリテラル値はドキュメントのコンテンツと完全に一致する必要があります。つまり、上記のクエリは、author 値が「Mark Twain」である場合はマッチされますが、作者が「M. Twain」や「mark twain」のドキュメントとはマッチされません。この動作は、ワードクエリおよびオプションを使用して変更できます。詳細については、「QBE のサブクエリタイプの概要」（195 ページ）および「QBE へのオプションの追加」（225 ページ）を参照してください。

値、ワード、およびレンジクエリを表す条件を作成できます。例えば、次のすべての条件を満たす QBE を作成できます。「条件の例」カラムに、各要件を表す、XML および JSON の条件を示します。

要件	クエリタイプ	条件の例

作者に「twain」が含まれる	単語	`<author><q:word>twain</q:word></author>`
		
		`"author": {"$word": "shakespeare"}`

ペーパーバック版がある	値	`<edition format="paperback"/>`
		
		`"edition": {"format": "paperback"}`

ペーパーバック版の価格が 9.00 ドル未満である	レンジ	`<edition>`
		`  <price><q:lt>9.00</q:lt></price>`
		`</edition>`
		
		`"edition": {"price": {"$lt": 9.00} }`

上記の条件を組み合わせて 1 つのクエリにすると、次の QBE が得られます。`query` の子要素が暗黙的に AND 結合されていることに注意してください。

形式	例

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>`

形式	例

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

上の例は、「作者は Mark Twain」のような直接包含を検索することを示しています。また、それらを含む XML 要素または JSON プロパティ内のあらゆる場所におけるマッチを検索できます。例えば、`book` に `author` の名前と `editor` の名前が含まれていて、`first-name` と `last-name` に分解されるとします。

形式	ドキュメントの例
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"
	  }
	} }

次のような条件を使用して、「書籍（book）に含まれる、名（first name）としての Mark の出現すべて」を検索できます。

形式	例
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" } } }

このような条件は、コンテナクエリの代表的なものです。詳細については、「コンテナクエリ」（201 ページ）を参照してください。

5.1.2 論理演算子

論理演算子を使用して、強力な複合クエリを作成できます。QBE の文法は、`and`、`or`、`not`、および `near` 構成子をサポートしています。次の例は、`author` という XML 要素または JSON プロパティ内に「twain」または「shakespeare」を含んでいるドキュメントとマッチされます。

形式	例
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" } }
	  ]
	  }
	}

クエリの直接の子であるサブクエリは、暗黙的な and クエリとなっています。

詳細については、「複合クエリ」（200 ページ）を参照してください。

5.1.3 比較演算子

QBE の文法では、XML 要素、XML 属性、および JSON プロパティの値に対するレンジクエリ用に次の比較演算子をサポートしています：lt、le、eq、ne、ge、gt。例えば、次のクエリは、価格が 10.00 以上かつ 20.00 以下であるすべてのドキュメントとマッチされます。

形式	例
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
	}}

上記のクエリに filtered フラグが含まれているのは、フィルタリングされた検索を使用するか、レンジクエリでレンジインデックスを利用するかのいずれかであるためです。詳細については、「インデックス付けがクエリに与える影響」（224 ページ）を参照してください。

詳細については、「レンジクエリ」（198 ページ）を参照してください。

5.1.4 値または単語（ワード）によるクエリ

リテラル値に対する条件は、値そのものとマッチされる暗黙的な値クエリになります。例えば、次の条件は、作者が「Mark Twain」であるものにのみマッチされます。「mark twain」および「M. Twain」とはマッチされません。

形式	例
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" }
	}

これが意図する動作ではない場合は、ワードクエリやオプションを使用してデフォルトの動作を修正できます。ワードクエリが値クエリと異なるのは、次の 2 つの点です。つまり、値クエリのデフォルトの完全一致を緩めることと、ドキュメント内の値のサブセットとマッチさせることです。

例えば、次のクエリは、author に「twain」が含まれているものにマッチされますが、大文字 / 小文字は区別されません。そのため、「Mark Twain」、「M. Twain」、「mark twain」など、元のクエリではマッチされなかった値とマッチされます。

形式	例
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" } }
	}

詳細については、「値クエリ」（195 ページ）および「ワードクエリ」（197 ページ）を参照してください。

5.1.5 検索結果のカスタマイズ

response という XML 要素または JSON プロパティを含めることで、返される検索結果のコンテンツをカスタマイズできます。デフォルトの検索結果には、マッチした XML 要素または JSON プロパティが強調表示されたスニペットが含まれます。QBE の response セクションを使用して、スニペットを生成しない、追加の要素を抽出する、ドキュメント全体を返すなどを設定できます。

詳細については、「検索結果のカスタマイズ」（230 ページ）を参照してください。

5.1.6 検索動作を制御するためのオプション

QBE の文法には、検索を制御するためのいくつかのフラグとオプションが含まれます。フラグは通常、検索結果のスコア計算方法など、検索に関してグローバルな影響があります。一方オプションは、特定の XML 要素または JSON プロパティの値に対して完全一致を実行するかどうかなど、クエリの一部分に影響します。

次の例では、exact オプションを使用して値クエリの完全一致をオフにしています。

形式	例
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"
	  }
	  }
	}

詳細については、「QBE へのオプションの追加」（225 ページ）を参照してください。

5.2 例

このセクションでは、QBE のほとんどのクエリ機能を使用する例を紹介します。

- XML の例
- JSON の例

5.2.1 XML の例

この例では、データベースに次の構造を持つドキュメントが含まれていると仮定しています。

```
<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>
```

次のクエリは、QBE のほとんどの機能を使用しており、上記のドキュメントとマッチされます。クエリの直接の子であるサブクエリは暗黙的に AND 結合されるので、マッチされるドキュメントはこれらのすべての条件を満たす必要があります。

```
<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>
```

次の表では、クエリの各コンポーネントが表現している要件について説明します。この例で使用されている各サブクエリタイプの詳細については、「QBE のサブクエリタイプの概要」（195 ページ）を参照してください。

要件	条件の例

タイトルが「Tom Sawyer」であること。完全一致がオフになっているので、マッチではスペース、句読点、発音符号が無視されます。マッチでは大文字 / 小文字を区別します。これはこの値（「Tom Sawyer」）では大文字と小文字が混在しているためです。	`<title>`
	`  <q:value exact="false">Tom Sawyer</q:value>`
	`</title>`

作者（author）に「mark」ならびに「twain」という単語が含まれていて、互いの距離が 2 単語以内になっていること。	`<q:near distance="2">`
	`  <author><q:word>mark</q:word></author>`
	`  <author><q:word>twain</q:word></author>`
	`</q:near>`

書籍の形態（edition format）が「paperback」であり、出版社（publisher）が「Clipper」または「Daw」であること。このサブクエリ内の個々の値では、値そのものに完全一致セマンティックを使用します。	`<edition format="paperback">`
	`  <q:or>`
	`  <publisher>Clipper</publisher>`
	`  <publisher>Daw</publisher>`
	`  </q:or>`
	`</edition>`

価格が 8.00 以上、10.00 未満であること。	`<q:and>`
	`  <price><q:lt>10.00</q:lt></price>`
	`  <price><q:ge>8.00</q:ge></price>`
	`</q:and>`

フィルタリングされない検索を使用すること。`price` にレンジインデックスがある場合は、このフラグを省略できます。詳細については、「インデックス付けがクエリに与える影響」（224 ページ）を参照してください。	`<q:filtered>true</q:filtered>`

5.2.2 JSON の例

この例では、データベースに次の構造を持つドキュメントが含まれていると仮定しています。

```
{"book": {
  "title": "Tom Sawyer",
  "author" : "Mark Twain",
  "edition": [
    { "format": "paperback",
      "publisher": "Clipper",
      "pub-date": "2011-08-01",
      "price" : 9.99,
      "isbn": "1613800917",
    }
  ]
} }
```

次のクエリは、QBE のほとんどの機能を使用しており、上記のドキュメントとマッチされます。クエリの直接の子であるサブクエリは暗黙的に AND 結合されるので、マッチされるドキュメントはこれらのすべての条件を満たす必要があります。

```
{"$query": {
    "title": {
      "$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
} }
```

次の表では、クエリの各コンポーネントが表現している要件について説明します。この例で使用されている各サブクエリタイプの詳細については、「QBE のサブクエリタイプの概要」（195 ページ）を参照してください。

要件	条件の例

タイトルが「Tom Sawyer」であること。完全一致がオフになっているので、マッチではスペース、句読点、発音符号が無視されます。マッチでは大文字 / 小文字を区別します。これはこの値（「Tom Sawyer」）では大文字と小文字が混在しているためです。	`"title": {`
	`  "$value": "Tom Sawyer",`
	`  "$exact": false`
	`}`

作者（author）に「mark」ならびに「twain」という単語が含まれていて、互いの距離が 2 単語以内になっていること。	`"$near": [`
	`  { "author": { "$word": "mark" } },`
	`  { "author": { "$word": "twain" } }`
	`],`
	`"$distance": 2`

版の形式が「paperback」であり、出版社が「Clipper」または「daw」であること。このサブクエリ内の個々の値では、値そのものに完全一致セマンティックを使用します。	`"edition": {`
	`  "format": "paperback",`
	`  "$or" : [`
	`  { "publisher": "Clipper" },`
	`  { "publisher": "Daw" }`
	`  ]`
	`}`

価格が 8.00 以上、10.00 未満であること。	`"$and": [`
	`  {"price": { "$lt": 10.00 }},`
	`  {"price": { "$ge": 8.00 }}`
	`]`

フィルタリングされない検索を使用すること。`price` にレンジインデックスがある場合は、このフラグを省略できます。詳細については、「インデックス付けがクエリに与える影響」（224 ページ）を参照してください。	`"$filtered": true`

5.3 QBE のサブクエリタイプの概要

QBE の `query` 部分はサブクエリから構成されます。QBE を使用すると、ドキュメントに合わせてモデル化したシンタックスを使用してサブクエリを表現できますが、このモデリングが表現しているクエリのタイプを理解する必要があります。QBE では、次のクエリタイプを表現できます。

- 値クエリ
- ワードクエリ
- レンジクエリ
- 複合クエリ
- コンテナクエリ

5.3.1 値クエリ

値クエリは、リテラル値（文字列、日付、数など）全体とマッチされます。

デフォルトでは、XML 要素または JSON プロパティの条件は、完全一致セマンティックの値クエリを表現しています。

- 条件内の値は、大文字 / 小文字の区別、発音符号による区別、句読点による区別、スペースの区別がオンになった状態でマッチされます。
- ステミング（語幹処理）とワイルドカード機能はオンになりません。
- 指定される値は、それを含む XML 要素または JSON プロパティの直接の子である必要があります。
- クエリ内の値が、ドキュメント内で値のサブセットになっている場合は一致しません。

例えば、次の条件は、author という XML 要素または JSON プロパティにテキスト「Twain」だけがこれと同じ形で含まれているドキュメントとのみマッチされます。「Mark Twain」や「twain」のような作者（author）の値とはマッチされません。

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

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

完全一致セマンティックのいくつかは、オプションで上書きできます。例えば、大文字 / 小文字の区別をオフにできます。詳細については、「QBE へのオプションの追加」（225 ページ）を参照してください。

値クエリは、明示的な場合もあれば、暗黙的な場合もあります。上記の例は、暗黙的な値クエリです。明示的な値クエリは、QBE キーワード value を使用して作ることができます。これは、値クエリにオプションを追加する場合に役立ちます。次の例は、case-sensitive オプションを使用する明示的な値クエリです。

形式	例
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 ワードクエリ

ワードクエリは、テキスト内のあらゆる場所に出現する単語（ワード）または語句（フレーズ）とマッチされます。ワードクエリは、テキスト値のサブセットとマッチされます。デフォルトでは、ワードクエリは完全一致セマンティックを使用しません。

- 条件内の値は、大文字 / 小文字の区別、発音符号による区別、句読点による区別、スペースの区別がオフになった状態でマッチされます。
- 語幹（ステミングされたもの）マッチが含まれます。
- データベースでワイルドカード機能がオンになっている場合、ワイルドカードマッチが実行されます。
- 指定される単語または語句は、それを含む XML 要素または JSON プロパティの値に出現しても、子コンポーネントの値に出現しても構いません。

一致セマンティックのいくつかは、オプションで上書きできます。詳細については、「QBE へのオプションの追加」（225 ページ）を参照してください。

ドキュメント内のコンテンツを記述する XML 要素や JSON プロパティのような他のコンテナ内に出現するワードクエリは、そのコンテナ内の出現とマッチされます。最上位レベルの QBE `query` ラッパーの直接の子であるワードクエリなど、コンテナ内にないワードクエリは、ドキュメント内のあらゆる場所の出現とマッチされます。詳細については、「コンテナクエリ」（201 ページ）および「ドキュメント全体の検索」（209 ページ）を参照してください。

次の QBE の例は、`author` に「twain」が含まれている場合にマッチされ、大文字 / 小文字は区別されません。そのため、「Mark Twain」、「M. Twain」、「mark twain」のような値とマッチされます。

形式	例
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" } }
	}

JSON では、ワードクエリ内の値は、文字列にすることも、文字列の配列にすることもできます。値の配列は、配列項目に対するワードクエリが AND の関係を持つリストとして扱われます。このため、文字列が配列項目として出現する場合にのみマッチされます。例えば、次のクエリは、`author` に配列項目内の「mark」および「twain」と単語がマッチするドキュメントとマッチされます。

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

5.3.3 レンジクエリ

レンジクエリは、文字列、数、日付、時刻、または dateTime 値に適用されている関係式（「5 未満」、「10 と等しくない」など）を満たす値とマッチされます。このセクションは、次の内容から構成されます。

- JSON プロパティ値のレンジクエリ
- XML 要素の値のレンジクエリ
- XML 要素の属性値のレンジクエリ
- レンジ式内での型変換

注： レンジクエリには、レンジインデックスを利用するか、`filtered` フラグを使用する必要があります。詳細については、「インデックス付けがクエリに与える影響」（224 ページ）を参照してください。

5.3.3.1 JSON プロパティ値のレンジクエリ

JSON プロパティ値のレンジクエリを作成するには、名前として “$” によるプレフィックスが付いた演算子名を持ち、値として境界の値を持つ JSON プロパティを作成します。

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

次の例の条件は、`format` が「paperback」と等しくないことをチェックします。

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

配列項目とマッチするように制約されたレンジクエリは作成できません。

5.3.3.2 XML 要素の値のレンジクエリ

XML 要素の値に対するレンジクエリを作成するには、次のシンタックスを使用します。ここで、q は http://marklogic.com/appservices/querybyexample を表す名前空間プレフィックスです。

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

次の例の条件は、刊行日が 2010-01-01 よりも後であることをチェックしています。

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

5.3.3.3 XML 要素の属性値のレンジクエリ

XML 要素の属性値に対するレンジクエリを作成するには、演算子名に「$」というプレフィックスを付け、それを含む要素条件において属性の文字列値内に比較式を置きます。

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

次の例の条件は、edition の @format が「paperback」と等しくないことをチェックします。

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

5.3.3.4 レンジ式内での型変換

デフォルトでは、レンジクエリ内の値は、xs:boolean、xs:double、xs:dateTime、xs:date、または xs:time としてキャスト可能な場合はそのように扱われ、それ以外の場合は文字列として扱われます。

xsi:type（XML）または $datatype（JSON）オプションを使用して、特定の型変換を強制できます。詳細については、「QBE へのオプションの追加」（225 ページ）を参照してください。

5.3.4 複合クエリ

複合クエリは、and、or、not、near のような論理演算子によって結合されたサブクエリから構成されるクエリです。次の例は、author（作者）の値が「Mark Twain」または「Robert Frost」であるドキュメントとマッチされます。

形式	例
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"
	  ]
	} }

near 演算子は、cts:near-query をモデリングしたものです。またこれにはオプションとして、オペランドのクエリとのマッチの許容可能な最大距離（単語数）を指定する distance XML 属性または JSON プロパティが利用できます。例えば、次の near クエリは、最大距離として 2 単語を指定しています。デフォルトの距離は 10 です。

形式	例
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 コンテナクエリ

コンテナクエリは、特定の XML 要素または JSON プロパティのスコープ内でサブクエリの条件が満たされる場合にマッチされます。コンテナクエリでは、名前付きコンテナと、サブクエリで使用される XML 要素または JSON プロパティの名前の間の関係は、単なる「〜の子」ではなく「〜に含まれる」になります。

コンテナクエリは、ドキュメントをモデリングし、複合クエリまたは構造化サブクエリ（XML 要素、XML 属性、または JSON プロパティ）を含む検索条件を使用したときに、暗黙的に定義されます。

例えば、次のような XML 条件は、他の要素 price に対する暗黙的な値クエリを含んでいるため、edition に対するコンテナクエリを定義しています。

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

対照的に、次の条件は author に対する値クエリであり、コンテナクエリではありません。

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

同様に、次の JSON 条件は、他のプロパティ price に対する暗黙的な値クエリを含んでいるため、edition に対するコンテナクエリです。

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

対照的に、次のような条件は author に対する値クエリであり、コンテナクエリではありません。

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

下記の例は、book に含まれている price に対するコンテナクエリが複数のレベルでどのようにマッチされるかを示しています。

クエリ	マッチされるドキュメントの例
<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}] } }

XML 要素の属性に対するクエリは、要素が属性を「含んでいる」という意味でコンテナクエリです。ただし、それを含んでいる要素の属性のみがマッチされます。次の条件は、@format が edition の属性として出現する場合にのみマッチされます。edition の子要素で出現した @format とはマッチされません。

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

次の表に、コンテナクエリの XML での例を示します。

説明	コンテナクエリ
要素 price は、要素 edition に含まれていて、その値はちょうど 8.99 です。 price は、edition の直接の子である必要はありません。	<edition> <price>8.99</price> </edition>
属性 format は、要素 edition に含まれていて、その値は「paperback」です。	<edition format="paperback"/>
要素 price は、要素 edition に含まれていて、その値はちょうど 8.99 です。また、要素 publisher は要素 edition に含まれていて、その値は「Fawcett」です。	<edition> <q:or> <price>8.99</price> <publisher>Fawcett</publisher> </q:or> </edition>

次の表に、コンテナクエリの JSON での例を示します。

説明	コンテナクエリ
JSON プロパティ price は、edition というプロパティに含まれていて、その値はちょうど 8.99 です。 price は、edition の直接の子である必要はありません。	"edition": {"price": 8.99 }

JSON プロパティ price は、edition というプロパティに含まれていて、その値はちょうど 8.99 です。また、publisher というプロパティは、edition という JSON プロパティに含まれていて、その値は「Fawcett」です。	"edition": {
	  "$or": [
	  "price": 8.99,
	  "publisher": "Fawcett"
	  ]
	}

5.4 検索条件のクイックリファレンス

このセクションでは、複合クエリの作成、およびドキュメントのモデリングに頻繁に使用される条件のテンプレートを提供します。

- XML 検索条件のクイックリファレンス
- JSON 検索条件のクイックリファレンス
- ドキュメント全体の検索

5.4.1 XML 検索条件のクイックリファレンス

下の表に、XML で QBE 検索条件および複合クエリを作成するためのクイックリファレンスを示します。これらの例を自分で条件を作成する際のテンプレートとして使用してください。詳細については、「QBE 構造リファレンス」（211 ページ）を参照してください。

下記の例では、名前空間のプレフィックス q が
`http://marklogic.com/appservices/querybyexample` にバインドされていると仮定しています。

条件の説明	例

要素 *e* が値 *v* を持つ	`<e>v</e>`
	
	`<e><q:value>v</q:value></e>`

要素 *e* の属性 *a* の値が *v* である	`<e a="v"/>`
	
	`<e a="$value v"/>`

要素 *e* が、要素コンテンツのサブ構造のどこかで単語 *w* を含んでいる	`<e><q:word>w</q:word></e>`
要素 *e* の属性 a の値が単語 *v* を含む	`<e a="$word w"/>`
要素 *e* が 5 よりも大きい値を持つ	`<e><q:gt>5</q:gt></e>`
要素 *e* の属性 a の値が 5 よりも大きい	`<e a="$gt 5"/>`
要素 *e* が存在する	`<e><q:exists/></e>`
要素 *e* の属性 *a* が存在する	*サポートされていません*

要素 *e1* が値 v を持つ要素 *e2* を含んでいて、*e2* が要素コンテンツのサブ構造内のどこかで出現する	`<e1>`
	`  <e2>v</e2>`
	`</e1>`

条件の説明	例

要素 *e1* が子孫要素 *e2* を含んでいて、*e2* が要素コンテンツのサブ構造内のどこかに単語 w を含んでいる	`<e1>`
	`  <e2><q:word>w</q:word></e2>`
	`</e1>`

要素 *e1* が子孫要素 *e2* を含んでいて、その子孫要素が *v* を持つ属性 *a* を持つ	`<e1>`
	`  <e2 a="v"/>`
	`</e1>`

要素 *e1* が子孫要素 *e2* を含んでいて、その子孫要素が値に単語 *w* を持つ属性 a を持つ	`<e1>`
	`  <e2 a="$word w"/>`
	`</e1>`

要素 *e* の子孫が属性 *a* を持つ	*サポートされていません*

要素 *e* が値 *v1* または *v2* を持つ	`<q:or>`
	`  <e>v1</e>`
	`  <e>v2</e>`
	`</q:or>`

要素 *e* が、要素コンテンツのサブ構造内のどこかに単語 *w1* または *w2* を含んでいる	`<q:or>`
	`  <e><q:word>w1</q:word></e>`
	`  <e><q:word>w2</q:word></e>`
	`</q:or>`

要素 *e1* が子孫要素 *e2* を含んでいて、*e2* が値 *v1* または *v2* を持つ	`<e1>`
	`  <q:or>`
	`  <e2>v1</e2>`
	`  <e2>v2</e2>`
	`  </q:or>`
	`</e1>`

要素 *e* の属性 *a* の値が *v1* または *v2* である	`<q:or>`
	`  <e a="v1"/>`
	`  <e a="v2"/>`
	`</q:or>`

要素 *e* の属性 *a* の値が単語 *w1* または *w2* を含む	`<q:or>`
	`  <e a="$word w1"/>`
	`  <e a="$word w2"/>`
	`</q:or>`

要素 *e1* の子孫要素である *e2* の属性 *a* の値が *v1* または *v2* である	`<e1>`
	`  <q:or>`
	`  <e2 a="v1"/>`
	`  <e2 a="v2"/>`
	`  </q:or>`
	`</e1>`

5.4.2 JSON 検索条件のクイックリファレンス

下の表に、JSON で QBE 検索条件および複合クエリを作成するためのクイックリファレンスを示します。サンプルのプロパティ名が *c* で始まっている場合、その条件はコンテナクエリを表します。詳細については、「QBE 構造リファレンス」（211 ページ）を参照してください。

条件の例のリストは、すべてを網羅しているわけではありません。これ以外のものもサポートされています。例えば、特定の条件について明示的および暗黙的な値クエリのすべてのバリエーションが示されているわけではありません。

条件の説明	例

値 *v* を持つプロパティ *k*	`{ "k": "v"}`
	
	`{ "k": {"$value": "v"}}`

値 *v* を持つ配列項目	`["v"]`
	
	`{"$value": ["v"]}`

プロパティ値のサブ構造内のどこかに単語 *w* を含んでいるプロパティ *k*	`{"k":{"$word":"w"}}`
単語 *w* を含む値を持つ配列項目	`{"$word":["w"]}`
5 よりも大きい値を持つプロパティ *e*	`{"k":{"$gt":5}}`
5 よりも大きい値を持つ配列項目	*サポートされていません*
プロパティ *k* が存在する	`{"k":{"$exists":{}}}`
配列項目が存在する	*サポートされていません*

値 *v* を持つ *k* というプロパティを含んでいる *c* というプロパティ。ただし、`k` は、*c* の値のサブ構造内のどこにでも出現できる。	`{"c": {"k":"v"} }`
	
	`{"c": {`
	`  "k": {"$value": "v"}`
	`} }`

単語 *w* を含む値を持つ *k* というプロパティを含んでいる *c* というプロパティ。ただし、*k* は、*c* の値のサブ構造内のどこにでも出現できる。	`{"c":{`
	`  "k":{"$word":"w"}`
	`}}`

条件の説明	例

値 *v* を持つ配列項目を含んでいる *c* というプロパティ。ただし、v は *c* の値のサブ構造内のどこにでも出現できる。	`{"c": ["v"] }`
	
	`{"c":{`
	`  "k":{"$value": ["v"]}`
	`}}`

単語 *w* を含む配列項目を含んでいる *c* というプロパティ。マッチする配列項目は、*c* の値のサブ構造内のどこにでも出現できる。	`{"c": {"$word": ["w"]} }`

値 *v1* または *v2* を持つ *k* というプロパティ	`{"$or":[`
	`  {"k":"v1"},`
	`  {"k":"v2"}`
	`]}`
	
	`{"$or":[`
	`  {"k": {"$value": "v1"}},`
	`  {"k": {"$value": "v2"}}`
	`]}`
	
	`{"k": {"$or": [`
	`  {"$value": "v1"},`
	`  {"$value": "v2"}`
	`]}}`

値に単語 *w1* または *w2* を含む *k* というプロパティ。	`{"$or":[`
	`  {"k":{"$word":"w1"}},`
	`  {"k":{"$word":"w2"}}`
	`]}`
	
	`{"k": {"$or": [`
	`  {"$word": "v1"},`
	`  {"$word": "v2"}`
	`]}}`

値 *v1* または *v2* を持つ *k* というプロパティを含んでいる *c* というプロパティ。ただし、k は *c* の値のサブ構造内のどこにでも出現できる。	`{"c":{"$or":[`
	`  {"k":"v1"},`
	`  {"k":"v2"}`
	`]}}`

値 *v1* を持つ *k* というプロパティと、値 *v2* を持つ *k* というプロパティを含んでいる *c* というプロパティ。ただし、k は *c* の値のサブ構造内のどこにでも出現できる。マッチするドキュメントは、*k* というプロパティの出現を少なくとも 2 つ含んでいる。	`{"c":{"$and":[`
	`  {"k":"v1"},`
	`  {"k":"v2"}`
	`]}}`
	
	`{"c":{"$and":[`
	`  {"k": {"$value": "v1"}},`
	`  {"k": {"$value": "v2"}}`
	`]}}`

条件の説明	例

値 *v1* を持つ *k1* というプロパティと、値 *v2* を持つ *k2* というプロパティ。*k1*、*k2* は異なるオブジェクト内に存在できる。	`{`
	`  "k1":"v1",`
	`  "k2":"v2"`
	`}`
	
	`{"$and":[`
	`  {"k1":"v1"},`
	`  {"k2":"v2"}`
	`]}`
	
	`{`
	`  "k1": {"$value": "v1"},`
	`  "k2": {"$value": "v2"}`
	`}`
	
	`{"$and":[`
	`  {"k1": {"$value": "v1"}},`
	`  {"k2": {"$value": "v2"}}`
	`]}`

値 *v1* を持つ配列項目または値 *v2* を持つ配列項目	`{"$or": ["v1", "v2"] }`
	
	`{"$or": [`
	`  {"$value": ["v1"]},`
	`  {"$value": ["v2"]}`
	`] }`

値 *v1* を持つ配列項目と値 *v2* を持つ配列項目	`["v1", "v2"]`
	
	`{"$value": ["v1", "v2"]}`
	
	`{"$and": ["v1", "v2"] }`
	
	`{"$and": [`
	`  {"$value": ["v1"]},`
	`  {"$value": ["v2"]}`
	`] }`

条件の説明	例

単語 *w1* を含む値を持つ配列項目または単語 *w2* を含む値を持つ配列項目	{"$or":[
	  {"$word": ["w1"]},
	  {"$word": ["w2"]}
	]}

単語 *w1* を含む値を持つ配列項目と単語 *w2* を含む値を持つ配列項目。これらは異なる配列内に存在できる。	{"$word": ["w1", "w2"]}
	
	{"$and": [
	  {"$word": ["w1"]},
	  {"$word": ["w2"]}
	] }

値 *v1* を持つ配列項目と値 *v2* を持つ配列項目を含んでいる *c* というプロパティ。ただし、v1 と v2 は *c* の値のサブ構造内のどこにでも出現できる。	{"c":["v1", "v2"]}
	
	{"c":{"$value":["v1", "v2"]}}
	
	{"c":{"$and":["v1", "v2"]}}
	
	{"$and":[
	  {"c":["v1"]},
	  {"c":["v2"]}
	]}
	
	{"$and":[
	  {"c": {"$value": ["v1"]}},
	  {"c": {"$value": ["v2"]}}
	]}

値 *v1* を持つ配列項目または値 *v2* を持つ配列項目を含んでいる *c* というプロパティ。ただし、v1 と v2 は *c* の値のサブ構造内のどこにでも出現できる。	{"c":{"$or":["v1", "v2"]}}
	
	{"$or":[
	  {"c":["v1"]},
	  {"c":["v2"]}
	]}
	
	{"$or":[
	  {"c": {"$value": ["v1"]}},
	  {"c": {"$value": ["v2"]}}
	]}

5.4.3 ドキュメント全体の検索

このセクションでは、マッチを特定の XML 要素、XML 属性、または JSON プロパティ内の出現に制約するのではなく、ドキュメント内のいずれの場所に出現する単語（ワード）または語句（フレーズ）とマッチさせるクエリを作成する方法について説明します。

ワードクエリは、XML 要素または JSON プロパティの条件に含まれていない場合、スコープはドキュメントです。例えば、最上位レベルのクエリ要素の直接の子であるワードクエリ、複合クエリ（and、or、not、near）の階層の任意の深さにある子であるワードクエリがそうです。これは、query クエリの直接の子を結合する暗黙的な and クエリも該当します。

例えば、次のクエリは、「moonlight sonata」という語句を含んでいるすべてのドキュメントとマッチされます。

形式	例
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" } }

次の例は、「moonlight sonata」という語句または「sunlight」という単語のいずれかを含んでいるすべてのドキュメントとマッチされます。

形式	例
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"}
	  ]
	} }

単語および語句の間の AND という関係性は、明示的にも暗黙的にもできます。次のクエリの例は、「moonlight sonata」という語句と「sunlight」という単語の両方を含んでいるすべてのドキュメントとマッチされます。

形式	例
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 構造リファレンス

このセクションでは、QBE のシンタックスとセマンティックについて説明します。次の内容を扱います。

- 最上位レベルの構造
- クエリのコンポーネント
- レスポンスのコンポーネント
- XML 固有の考慮事項
- JSON 固有の考慮事項

5.5.1 最上位レベルの構造

最上位レベルでは、QBE は query を入れる必要があり、オプションで response フラグや format フラグを入れることができます。QBE には、次の最上位レベルパーツがあります。

- query：query でマッチさせるドキュメントの要件を定義します。
- response：response で検索結果をカスタマイズします。response がない場合、デフォルトの検索レスポンスが返されます。
- format：format フラグを使用して、無修飾名の解釈を名前空間なしの JSON プロパティ名または XML 要素名（クエリ形式に基づく）として上書きします。詳細については、「ドキュメントタイプによる検索のスコープ設定」（237 ページ）を参照してください。

次の表に、QBE の最上位レベルの概要を示します。

形式	例

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:qbe>

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

query は、要素またはプロパティの条件または複合クエリを定義している 1 つあるいは複数の XML 要素または JSON プロパティを含みます。ドキュメントの構造をモデリングするには、条件を使用します。and、or、not、near のような演算子でサブクエリを論理的に結合するには、複合クエリを使用します。

XML では、query と response は、qbe ラッパー要素に入れられます。qbe、query、word など、QBE 文法によって事前定義された要素および属性の名前は、名前空間 http://marklogic.com/appservices/querybyexample 内にあります。他のすべての要素および属性の名前は、ドキュメント内の要素および属性の名前を表します。詳細については、「名前空間の管理」（216 ページ）を参照してください。

JSON では、QBE 文法によって事前定義されたすべてのプロパティ名には、$query、$word のように「$」というプレフィックスがあります。プレフィックス「$」がないすべてのプロパティ名は、ドキュメント内のプロパティを表します。詳細については、「プロパティの命名規則」（218 ページ）を参照してください。

通常は、format フラグを設定する必要はありません。format フラグを設定する必要があるのは、JSON QBE を使用して XML ドキュメントをマッチさせる（またはその逆でマッチさせる）場合だけです。詳細については、「ドキュメントタイプによる検索のスコープ設定」（237 ページ）を参照してください。

5.5.2 クエリのコンポーネント

下の表では、QBE の query 部分のコンポーネントについて説明します。形式固有のその他の詳細については、「XML 固有の考慮事項」（216 ページ）および「JSON 固有の考慮事項」（218 ページ）で扱っています。

コンポーネントタイプ	XML ローカル名	JSON プロパティ名	説明
クエリ	query	$query	検索条件を定義します。必須です。
条件	*要素名*	*プロパティ名*	ドキュメント内の XML 要素または JSON プロパティのスコープ内で適用される検索条件を定義します。この名前は、クエリによってマッチされるコンテンツ内の要素またはプロパティに対応します。 条件は、複合クエリまたは他の条件をラップする場合、コンテナクエリとなります。それ以外の場合は、値クエリ、ワードクエリ、またはレンジクエリとなります。

コンポーネントタイプ	XML ローカル名	JSON プロパティ名	説明
複合クエリ	and or not near	$and $or $nor $near	論理演算子を使用してサブクエリを結合する複合クエリを定義します。 near 演算子には、オプションの distance XML 属性または JSON プロパティが利用できます。 • <q:near distance="5">...</q:near> • { "$near":"$distance":5, [..]}
レンジクエリ	lt, le gt, ge eq, ne	$lt, $le $gt, $ge $eq, $ne	XML 要素、XML 属性、または JSON プロパティ内で値の関係式を定義します。
修飾子	value word exists	$value $word $exists	値のマッチ方法として、値クエリ（修飾子がない場合のデフォルト）、ワードクエリ、または存在テストのいずれを使用するかを定義する値の修飾子。
フラグ	filtered score	$filtered $score	フラグは、検索動作の修飾子です。 ブール型の filtered フラグを使用して、検索がフィルタリングありとフィルタリングなし（デフォルト）のいずれであるかを制御します。詳細については、「インデックス付けがクエリに与える影響」（224 ページ）を参照してください。 score フラグを使用して、検索結果のスコア計算関数を上書きします。使用可能な値：logtf、logtfidf、random、simple、zero。デフォルト：logtfidf。詳細については、「関連度スコア：概要とカスタマイズ」（289 ページ）を参照してください。
options			オプションを使用して、検索条件および結果を細かく設定します。詳細については、「QBE へのオプションの追加」（225 ページ）を参照してください。

次の表に、各コンポーネントタイプを使用できる場所をまとめます。オプションについては、「QBE へのオプションの追加」（225 ページ）で扱います。

コンポーネントタイプ	含まれる内容	含まれる場所
クエリ	1 つあるいは複数の条件、複合クエリ、および `filtered` または `score` フラグ	`qbe`（XML） ルートオブジェクト（JSON）
条件	• なし（空）、または • 1 つの値、または • 1 つの単語、（明示的な）値、レンジクエリのいずれか、または • 1 つあるいは複数の条件または複合クエリ	`query`、複合クエリ、条件
複合クエリ（`and`、`or` など）	1 つあるいは複数の条件、複合クエリ、ワードクエリ。ワードクエリは、複合クエリが `query` の直接の子である場合にのみ許可されます。	`query`、複合クエリ、または条件
レンジクエリ（`lt`、`gt` など）	値	条件
ワードクエリまたは値クエリ	値（文字列、数、日付、時刻、dateTime）	`word`：`query`、条件、または複合クエリ `value`：条件、条件によって含まれている複合クエリ
`exists`		条件
フラグ		`query`

5.5.3 レスポンスのコンポーネント

QBE の response 部分を使用して、検索結果の形式をカスタマイズできます。次の表では、response のコンポーネントについて説明します。response はオプションであり、query の兄弟として QBE の最上位レベルでのみ出現できます。

response には、次のフォーマッタコンポーネントを入れることができます。

XML ローカル名	JSON プロパティ名	説明
snippet	$snippet	snippet 要素は、検索のマッチとして返される内容を制御します。マッチしたものの表示方法を指定したり、何を表示するのかに関するポリシー（default、document、none）を設定したりできます。
extract	$extract	extract 要素は、マッチしたドキュメントから抽出する XML 要素または JSON プロパティをリスト化することによって snippet を補足します。リスト化された要素やプロパティ内でマッチが出現するかどうかは問いません。

詳細については、「検索結果のカスタマイズ」（230 ページ）を参照してください。

5.5.4 XML 固有の考慮事項

このセクションでは、QBE を XML で作成するときに知っておく必要がある構造およびセマンティックの詳細について扱います。

- 名前空間の管理
- 属性のクエリ

5.5.4.1 名前空間の管理

qbe、query、word など、QBE 文法内のすべての事前定義された要素名に対して、名前空間 http://marklogic.com/appservices/querybyexample を使用してください。この名前空間は、ドキュメントをモデリングする条件要素からクエリの構造を区別するものです。この名前空間は QBE の最上位レベルで定義します。例えば以下のようになります。

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

使用する要素条件に必要な名前空間を、条件や要素コンテナにおいて定義します。QBE 内で同一の名前空間のプレフィックスを異なる名前空間にバインドすることはできません。

次の例は、ルートの `qbe` 要素、含んでいる要素、および要素条件上でユーザー定義の名前空間を宣言する方法を示しています。

```
<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 属性のクエリ

要素属性に対するクエリを実行するには、その属性を含む要素条件を作成します。次の例は、「paperback」という値を指定した属性 `edition/@format` の値クエリを表しています。

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

属性の値は、上記の例のような暗黙的な値クエリ、明示的な値クエリ、ワードクエリ、レンジクエリのいずれにもできます。属性に対するワードクエリ、レンジクエリ、または明示的な値クエリを作成するには、次の属性値のテンプレートを使用します。ここで、*keyword* は、修飾子（`word` または `value`）または比較子（`lt`、`gt` など）です。

```
$keyword
```

例えば、次の QBE は、属性 `edition/@price` に対するレンジクエリを表しています。

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

属性値内では、`exists` 修飾子は使用できません。

要素条件の複数の属性は AND 結合されます。例えば、次の QBE では、`edition/@price` に対するレンジクエリと、`edition/@format` ペーパーバックに対するワードクエリを使用して、価格が 9.00 未満のすべてのペーパーバック版を探します。

```
<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>
```

属性条件では `exact`、`min-occurs`、`score-function` など、レンジクエリ、ワードクエリ、および値クエリのオプションを使用することはできません。属性クエリを対象とするこのレベルの制御が必要な場合は、QBE ではなく構造化クエリを使用してください。詳細については、「構造化クエリを使用した検索」（74 ページ）を参照してください。

5.5.5 JSON 固有の考慮事項

このセクションでは、QBE を JSON で作成するときに知っておく必要がある構造およびセマンティックの詳細について扱います。次の内容を扱います。

- プロパティの命名規則
- 配列項目のマッチ
- 配列およびオブジェクトコンテナの検索
- Node.js QueryBuilder を使用した QBE の作成

5.5.5.1 プロパティの命名規則

JSON では、QBE 文法で事前定義されたすべての JSON プロパティ名を、ドキュメント内で出現する名前と区別するため「`$`」プレフィックスを使用します。例えば、JSON QBE の `query` 部分のプロパティ名は `$query` です。

ドキュメントに「$」で始まるプロパティ名が含まれる場合、この名前が、事前定義されたプロパティ名と競合する可能性があります。このような場合は、QBE ではなく構造化クエリを使用する必要があります。詳細については、「構造化クエリを使用した検索」（74 ページ）を参照してください。

事前定義されたプロパティ名のリストについては、「クエリのコンポーネント」（213 ページ）および「レスポンスのコンポーネント」（216 ページ）を参照してください。

5.5.5.2 配列項目のマッチ

ワードクエリ、値クエリ、またはリテラル値は、角括弧内に含まれている場合、配列項目とのみマッチされます。例えば、次のクエリは、k というプロパティの下にある配列項目として「v」が出現した場合にのみ、「v」とマッチされます。

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

配列項目の値は、ワードクエリまたは値クエリとして表すことができます。例えば、次のクエリは、「k」というプロパティのサブ構造内のいずれかの場所で単語「v」を含む配列項目とマッチされます。

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

プロパティ名と配列項目の値の間の関係性は、次のように包含となります。マッチされる配列項目は、k に関連付けられている値のサブ構造内のどこにでも出現できます。例えば、次のすべてのドキュメントは上記のクエリとマッチします。

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

{ "k": { "k2": ["v"] } }

{ "k": { "k2": ["v","v2"] } }
```

クエリ内の各配列項目の値は、独立したマッチを表します。配列とマッチされるクエリを作成することはできません。個々の配列項目に対するマッチのみを実行できます。例えば、次のクエリについて考えてみましょう。

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

これは次のすべてのドキュメントとマッチします。

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

{ "k": [
  { "k1": ["v1"] },
  { "k2": ["v2"] }
] }

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

詳細については、「配列およびオブジェクトコンテナの検索」（220 ページ）を参照してください。

5.5.5.3 配列およびオブジェクトコンテナの検索

コンテンツ内の JSON プロパティの名前を指定する条件プロパティによって表されるクエリのタイプは、プロパティ内の値の型によって異なります。値は、配列、オブジェクト、または複合クエリである場合、コンテナクエリとなります。それ以外の場合は、値クエリ、ワードクエリ、またはレンジクエリです。コンテナクエリが JSON ドキュメントの検索にどのように適用されるかを理解することが重要です。

値がリテラル値、ワードクエリ、値クエリ、またはレンジクエリである場合、条件プロパティは「これらの条件を満たす値を持つ *k* という JSON プロパティとマッチされる」ということになります。このような条件はコンテナクエリではありません。下の表では、これらの形式について説明します。

条件テンプレート	条件の例	説明
name : *value*	{ "price" : 8.99 }	値が 8.99 である「price」というプロパティとマッチされます

name : {
  word-or-value : *value*
}
	{ "title" : {
	  "$word" : "sawyer"
	} }
		値に「sawyer」が含まれる「title」というプロパティとマッチされます

name : {
  relational-op : *value*
}
	{ "price" : {
	  "$lt" : 9
	} }
		値が 9 未満である「price」というプロパティとマッチされます

値がオブジェクト、配列、または複合クエリである条件プロパティはコンテナクエリです。このようなクエリは、「サブ構造内のどこかでこれらの条件を満たす値を含んでいる *c* というプロパティとマッチさせる」というようになります。下の表では、これらの形式について説明します。

条件テンプレート	条件の例	説明
name : *object*	{"edition": {"price" : 8.99 } }	値が 8.99 で、「edition」というプロパティ内のどこかに含まれている「price」という JSON プロパティとマッチされます

条件テンプレート	条件の例	説明

name : [
  array-item+
]
	{"format" : [
	  "paperback",
	  "hardback"
	] }
		値「paperback」を持つ配列項目と値「hardback」を持つ配列項目を含んでいる「format」という JSON プロパティとマッチされます。配列項目は、同一の配列内に出現する必要はなく、配列項目の値がこれだけである必要もありません。

name : {
  logical-op : [
  sub-query+
  ]
}
	{"edition" : {
	  "$or": [
	  {"format": "paperback"},
	  {"format": "hardback"}
	  ]
	} }
		「edition」というプロパティ内のどこかに含まれていて、値が「paperback」または「hardback」である「format」という JSON プロパティとマッチされます。

コンテナクエリは常に、コンテナサブ構造内のどこかにあるそのサブクエリとマッチされるので、「*k* という名前のプロパティがあり、その値がこれらの項目の値そのものを単独で含む配列であるコンテナ」や「*k* という名前のプロパティがあり、その値がこのオブジェクトそのものであってそれ以外ではないコンテナ」とマッチされる JSON QBE を作成することはできません。

下の表に、値クエリおよびいくつかの種類のコンテナクエリにマッチするドキュメントの例を示します。マッチするドキュメントの例は、すべてを網羅しているわけではありません。各クエリには、ドキュメントにマッチされる条件を説明する注釈が付いています。その他の例については、「JSON 検索条件のクイックリファレンス」（206 ページ）を参照してください。

QBE	マッチ
{"$query": { "k": "v" } } プロパティ *k* が値「*v*」を持つ	{ "k": "v" }
{"$query": { "c": ["v"] } } プロパティ *c* が、値「*v*」を持つ配列項目を含んでいる	{ "c": ["v"] } { "c": { "k": ["v"] } } { "c": { "k": ["v","v2"] } }
{"$query": {"c": ["v1", "v2"]} } プロパティ *c* が、値「*v1*」を持つ配列項目と値「*v2*」を持つ配列項目を含んでいる	{ "c": ["v1", "v2"] } { "c": { "k": ["v1","v2"] } } { "c": [{ "k1": ["v1"] }, { "k2": ["v2"] }] } { "c": [{"k" : ["v1", "v3"] }, {"k" : ["v2", "v4"] }] }
{"$query": {"c": {"k": "v"}} } プロパティ *c* が、値「*v*」を持つプロパティ *k* を含んでいる	{ "c" : {"k" : "v" } } { "c" : {"k1": "v", "k2": "v2" } } { "c" : {"c2": {"k": "v" }} }

5.5.5.4 Node.js QueryBuilder を使用した QBE の作成

このトピックでは、この章の内容を Node.js クライアント API で使用する方法について説明します。

Node.js クライアント API を使用すると、`QueryBuilder.byExample` 関数を使用した QBE を作成できます。`byExample` のパラメータは、JavaScript オブジェクトで表される、生の QBE の `$query` 部分内の条件に対応します。例えば、下の表は、この章の他の箇所からの QBE の例、および等価な `QueryBuilder.byExample` 呼び出しを示しています。

生の QBE	QueryBuilder.byExample

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

QBE の `$query` 部分全体を JavaScript オブジェクトとして `byExample` に指定することもできます。例えば以下のようになります。

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

ただし、`QueryBuilder.byExample` を通じて生の QBE の `$format` または `$response` 部分を指定することはできません。ただし、レスポンスのカスタマイズは、`QueryBuilder.extract` および `QueryBuilder.snippet` を通じて行うことが可能です。

Node.js クライアント API を使用して XML ドキュメントを検索する場合は、QBE を使用することはできません。

詳細については、『Node.js Application Developer's Guide』の「Querying Documents and Metadata」を参照してください。

5.6 インデックス付けがクエリに与える影響

QBE を使用するためにインデックスを定義する必要はありません。このため、QBE をすぐに使い始めることができます。ただし、インデックスは、検索のパフォーマンスを大幅に向上さます。

データベースが小さい場合や、クエリが返すのが事前にフィルタリングされた少数の結果だけの場合以外では、XML 要素、XML 属性、および JSON プロパティにインデックスを定義してください。インデックスを設定するには、『Administrator's Guide』の「Range Indexes and Lexicons」を参照してください。

QBE にレンジクエリが含まれる場合は、レンジクエリで使用される XML 要素、XML 属性、または JSON プロパティにインデックスを設定するか、`filtered` フラグを使用して検索のフィルタリングを強制する必要があります。

フィルタリングされる検索では、使用可能なインデックス（存在する場合）が使用されますが、その後で各候補がクエリの要件を満たすかどうかをチェックされます。これにより、フィルタリングされる検索は正確になりますが、フィルタリングされない検索よりもずっと遅くなります。フィルタリングされない検索では、インデックスのみでマッチを特定します。この識別は高速ですが、誤検知（false positive）が発生する可能性があります。詳細については、『Query Performance and Tuning Guide』の「Fast Pagination and Unfiltered Searches」を参照してください。

利用できるインデックスがない場合、フィルタリングされない検索ではレンジクエリを使用できません。フィルタリングされる検索をオンにするには、次の例に示すように、QBE の `query` 部分で `filtered` フラグを true に設定します。

形式	例

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 QBE へのオプションの追加

オプションを使用すると、QBE をきめ細かく制御できます。ほとんどのオプションは、値クエリ、ワードクエリ、またはレンジクエリに関連付けられます。

- XML でのオプションの指定
- JSON でのオプションの指定
- オプションリスト
- 永続的なクエリオプションの使用

5.7.1 XML でのオプションの指定

XML QBE では、オプションは、<q:lt/>、<q:word/>、<q:value> など、変更可能な事前定義された QBE 要素の属性です。次のクエリは、値クエリで exact オプションを使用する方法を示しています。

```
<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>
```

レンジクエリ、ワードクエリ、または値クエリは属性値に埋め込まれているため、属性に対するクエリにはオプションを適用できません。例えば、case-sensitive オプションを次のような属性に対するワードクエリに追加することはできません。

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

要素属性クエリをこのように制御する必要がある場合は、構造化クエリまたは複合クエリを使用する必要があります。

5.7.2 JSON でのオプションの指定

JSON QBE では、オプションは、値クエリ、ワードクエリ、レンジクエリなど、変更可能な QBE オブジェクトの兄弟です。オプションの名前には常に、プレフィックス「$」があります。

次の例のクエリでは、exact オプションを使用して値クエリを修正します。これを行うには、JSON プロパティとして、$value オブジェクトと同じレベルに含めます。

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

5.7.3 オプションリスト

次の表では、QBE で使用可能なオプションについて説明します。MarkLogic サーバー search API は、文字列、構造化クエリなどの他のクエリ形式を通じて、および永続的なクエリオプションの使用を通じて、追加のオプションをサポートしています。詳細については、「クエリオプションを使用した検索のカスタマイズ」（255 ページ）を参照してください。

オプションの属性またはプロパティ名	説明
case-sensitive	大文字 / 小文字の区別あり（ケースセンシティブ）のマッチを実行するかどうか。デフォルト：マッチされるテキストがすべて小文字である場合は false、それ以外の場合は true。値の型：ブール型。使用可能なクエリ：ワードクエリまたは値クエリ。詳細については、「cts:word-query」または「cts:value-query」を参照してください。
diacritic-sensitive	発音符号による区別あり（ダイアクリティックセンシティブ）のマッチを実行するかどうか。デフォルト：コンテキストに依存し、マッチされるテキストに発音符号（ダイアクリティック）が含まれていない場合は false、それ以外の場合は true。値の型：ブール型。使用可能なクエリ：ワードクエリまたは値クエリ。詳細については、「cts:word-query」または「cts:value-query」を参照してください。
punctation-sensitive	句読点による区別あり（パンクチュエーションセンシティブ）のマッチを実行するかどうか。デフォルト：コンテキストに依存し、マッチされるテキストに句読点が含まれていない場合は false、それ以外の場合は true。値の型：ブール型。使用可能なクエリ：ワードクエリまたは値クエリ。詳細については、「cts:word-query」または「cts:value-query」を参照してください。

オプションの属性またはプロパティ名	説明
whitespace-sensitive	スペースによる区別あり（ホワイトスペースセンシティブ）のマッチを実行するかどうか。デフォルト：false。値の型：ブール型。使用可能なクエリ：ワードクエリまたは値クエリ。詳細については、「cts:word-query」または「cts:value-query」を参照してください。
stemmed	ステミング（語幹処理）を使用するかどうか。デフォルト：コンテキストおよびデータベース設定に依存します。詳細については、「cts:word-query」を参照してください。値の型：ブール型。使用可能なクエリ：ワードクエリまたは値クエリ。詳細については、「cts:word-query」または「cts:value-query」を参照してください。
exact	完全一致を実行するか、*-sensitive オプションにビルトインのコンテキストセンシティブなデフォルトの動作を使用するか。true の場合、exact は、大文字 / 小文字の区別、発音符号による区別、句読点による区別、スペースの区別があり、ステミングなし、およびワイルドカード機能なしになります。デフォルト：値クエリおよびレンジクエリの場合は true、ワードクエリの場合は false。値の型：ブール型。使用可能なクエリ：ワードクエリまたは値クエリ。
score-function	選択されたスコア計算関数を使用します。使用可能な値：linear、reciprocal。使用可能なクエリ：レンジクエリ。詳細については、「スコア計算にレンジクエリや位置クエリを含める」（298 ページ）を参照してください。
slope-factor	スコア計算関数の傾きにスケーリングファクタとして特定の数値を適用します。デフォルト：1.0。値の型：double。使用可能なクエリ：レンジクエリ。詳細については、「スコア計算にレンジクエリや位置クエリを含める」（298 ページ）を参照してください。
min-occurs	必要な出現の最小数。出現がこれよりも少ないフラグメントはマッチしません。デフォルト：1。値の型：integer。使用可能なクエリ：レンジクエリ、ワードクエリ、または値クエリ。詳細については、「cts:word-query」を参照してください。

オプションの属性またはプロパティ名	説明
max-occurs	必要な出現の最大数。出現がこれよりも多いフラグメントはマッチしません。デフォルト：バインドなし。値の型：integer。使用可能なクエリ：レンジクエリ、ワードクエリ、または値クエリ。詳細については、「cts:word-query」を参照してください。
lang	コンテンツの解釈に使用する言語。オプションの値は大文字 / 小文字の区別がありません。使用可能な値：ISO 639 言語コード。デフォルト：データベースに対して設定されているデフォルトの言語。使用可能なクエリ：query。レンジクエリ、ワードクエリ、または値クエリ。XML では、qbe 要素でも使用できます。JSON では、最上位レベルのプロパティとして使用できます。
weight	このクエリの重み。重みが大きいほど、検索結果が関連度順で上へ移動します。使用可能な値：-16 以上、64 以下（-16 〜 64）。デフォルト：1.0。使用可能なクエリ：ワードクエリ、値クエリ、またはレンジインデックスが利用されているレンジクエリ。詳細については、「cts:word-query」、「cts:value-query」、または「cts:element-range-query」を参照してください。
constraint	検索に関連付けられている永続的なクエリオプションで同じ XML 要素または JSON プロパティに対して指定されるレンジ制約、値制約、または単語（ワード）制約の名前。使用可能なクエリ：レンジクエリ、ワードクエリ、または値クエリ。詳細については、「永続的なクエリオプションの使用」（229 ページ）を参照してください。
@xsi:type (XML) $datatype (JSON)	レンジクエリで指定された値のキャスト先の xsi:type。デフォルト：値は、xs:boolean、xs:double、xs:date、xs:dateTime、または xs:time としてキャスト可能な場合はそのように扱われ、それ以外の場合は文字列として扱われます。使用可能なクエリ：レンジクエリ。

5.7.4 永続的なクエリオプションの使用

REST API および Java API を使用すると、REST インスタンスに永続的なクエリオプションをインストールし、以後の検索に適用できます。Node.js クライアント API でも永続的なクエリオプションを使用できますが、この API には、永続的なオプションを作成および保守するための機能はありません。

永続的なクエリオプションを QBE とともに使用すると、QBE の文法で直接サポートされていないオプションを使用できます。永続的なオプションを QBE とともに使用すると、クエリ全体に適用されるグローバルオプションを定義できます。例えば、QBE 内の各ワードクエリで `case-sensitive` オプションを指定する代わりに、すべてのワードクエリを大文字 / 小文字の区別あり（ケースセンシティブ）にできます。

`constraint` オプションを通じて適用されたクエリオプションは、QBE 上でインライン指定されたオプションを上書きします。

`constraint` オプションを使用して、永続的なクエリオプションを QBE に適用できます。このオプションを使用するには、次の手順に従います。

1. 使用する API に該当する指示に従って、名前付きの永続的なクエリオプションをインストールします。

2. QBE のワードクエリ、値クエリ、またはレンジクエリの `constraint` オプションの値として、手順 1 の永続的なオプションで定義されている制約の名前を指定します。次の例を参照してください。

3. QBE で検索を実行するときは、クライアント API（REST、Java、または Node.js）で規定されている方法で、手順 1 の永続的なクエリオプションを検索に関連付けます。

永続的なクエリオプションの定義、インストール、および使用の詳細については、『REST Application Developer's Guide』の「Configuring Query Options」または『Java Application Developer's Guide』の「Query Options」を参照してください。

`constraint` オプションによって名前を指定された事前定義された制約は、適用先のクエリのタイプと一致する必要があります。つまり、レンジクエリの場合はレンジ制約、値クエリの場合は値制約、ワードクエリの場合は単語制約となります。

次の例では、「w-t」という単語制約を事前定義しています。この単語制約は、XML 要素または JSON プロパティ `title` でのマッチに重み 2.0 を与え、これを `title` に対するワードクエリを含んでいる QBE に適用します。これにより、`constraint` オプションを省略することにより、`title` のワードクエリのデフォルトの重みを上書きできます。

QBE で実行される検索へのパラメータとして指定された、次の永続的なクエリオプションがインストールされている場合：

XML オプション	JSON オプション

<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
	}
	}]
	} }

次の QBE は、「w-t」オプションを `title` に対するワードクエリに適用して、`title` 要素でのマッチに重み 2.0 を与えます。

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 検索結果のカスタマイズ

`response` という XML 要素または JSON プロパティを含めることで、返される検索結果のコンテンツをカスタマイズできます。QBE の `response` セクションで `snippet` および `extract` フォーマッタを使用して、デフォルトの検索結果を修正または補足できます。

このセクションでは、次のトピックについて扱います。

- クエリにレスポンスを含める状況
- スニペットフォーマッタの使用
- 抽出フォーマッタの使用

5.8.1 クエリにレスポンスを含める状況

次の 1 つあるいは複数のことを実行する場合は、QBE にオプションの `response` セクションを追加します。

- スニペットではなくマッチしたドキュメントを返します。(`snippet`)
- データベースの URI、ドキュメント形式、関連度スコアなど、ドキュメントおよびマッチに関する情報のみを返します。(`snippet`)
- スニペットを作成するときに優先する XML 要素または JSON プロパティを指定します。(`snippet`)
- XML 要素または JSON プロパティがマッチするかどうかに関係なく、マッチしたドキュメントから抽出する XML 要素または JSON プロパティを指定します。(`extract`)

結果のデコレータ、変換、および永続的なクエリオプションを使用して、高度なカスタマイズを行うことができます。詳細については、『REST Application Developer's Guide』の「Customizing Search Results」または『Java Application Developer's Guide』の「Transforming Search Results」を参照してください。

5.8.2 スニペットフォーマッタの使用

snippet を使用して、検索がマッチしたスニペット部分に含める内容（ある場合）を制御し、スニペットを含めるために優先する要素を識別します。デフォルトのスニペットは、テキストの一部の抜粋であり、マッチしたテキストには強調表示のためのタグが付けられています。次の表に、デフォルトのポリシーで生成された検索レスポンスのスニペットセクションの抜粋を示します。

形式	デフォルトのスニペットの例

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" } ]**`
	`  } ]`
	`  } ],`
	`  ...`
	`}`

スニペットフォーマッタは次の形式を取ります。

XML	JSON

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

policy、*preferred-element*、および *preferred-property* はオプションです。

スニペット生成ポリシーは、スニペットを出力に含めるかどうか、およびスニペットがオンの際にテキストの一部の抜粋（デフォルト）あるいはドキュメント全体のどちらを含めるかを制御します。ポリシーには、次のいずれかの要素名またはプロパティ名を使用します。

XML	JSON	説明
`default`	`$default`	マッチしたタームの周囲のテキストの一部の抜粋を含め、マッチしたテキストに強調表示のためのタグを付けます。
`document`	`$document`	ドキュメント全体を返します。
`none`	`$none`	スニペットを含めません。

次の例は、スニペットポリシーを `none` に設定することによってスニペットの生成をオフにします。JSON では、ポリシーのプロパティに対して空のオブジェクト値を指定します。

XML	JSON

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

スニペットを生成するときに優先する 1 つあるいは複数の XML 要素または JSON プロパティの名前を指定することもできます。例えば、`title` 要素またはプロパティを優先するように指定していて、`title` と `author` の両方がマッチした場合、スニペットは `title` でのマッチのほうから生成されます。JSON では、優先するプロパティを空のオブジェクト値で指定します。

XML	JSON

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

5.8.3 抽出フォーマッタの使用

extract フォーマッタを使用して、検索出力に含める追加の XML 要素または JSON プロパティを指定します。スニペットが含まれている場合、抽出されるコンポーネントは、スニペットを置き換えるのではなくマッチしたあらゆるスニペットを補足します。

XML	JSON

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

例えば、次のレスポンスは、title および author をマッチしたドキュメントから抽出するように指定しています。title および author に、マッチするタームおよび値が含まれている必要はありません。

XML	JSON

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

抽出された要素またはプロパティは、マッチ結果の metadata セクションに入ります。例については、「例：検索のカスタマイズ」（235 ページ）を参照してください。

5.8.4 例：検索のカスタマイズ

次の QBE は、スニペットを除外し、`title` XML 要素または JSON プロパティを抽出して検索結果の `metadata` セクションに入れるように検索結果を修正します。

形式	例
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": {} }
	  }
	}

次の表に、上記のクエリによって生成されたデフォルトの出力および修正された出力を示します。

形式	デフォルトの出力	カスタマイズされた出力
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 ドキュメントタイプによる検索のスコープ設定

このセクションでは、QBE での無修飾名の扱いが、クエリがマッチされるドキュメントのタイプにどのように影響するかについて説明します。この説明は、Java クライアント API と REST クライアント API に適用されます。Node.js クライアント API は、JSON ドキュメントを対象とする QBE のみをサポートしています。

JSON QBE の無修飾名は、プレフィックス「$」を含まないプロパティ名です。XML QBE の無修飾名は、名前空間なしの要素名です。

デフォルトでは、無修飾名の解釈はクエリ形式と一致します。つまり、JSON QBE の無修飾名は、コンテンツ内のプロパティ名を表し、XML QBE の無修飾名は、名前空間なしのコンテンツ内の要素名を表します。実際の影響として、XML QBE は XML ドキュメントとのみマッチされ、JSON QBE は通常、JSON ドキュメントとのみマッチされます。

デフォルトの動作を上書きするには、`format` オプションを使用します。次の例に示すように、QBE のクエリ部分に `format` フラグを設定する必要があります。

形式	例

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": {...}`
	`}`

`format` オプションを適切に使用する方法を理解するには、JSON ドキュメントがデータベース内にどのように格納されているかを知っている必要があります。ドキュメントは、REST API または Java API によって JSON ドキュメントとしてデータベースに挿入された場合、JSON ドキュメントと見なされます。読み込みの際に、このようなドキュメントは、`http://marklogic.com/xdmp/json/basic` をそのデフォルトの名前空間として使用する、内部の XML 表現に変換されます。

デフォルトでは、QBE の文法に含まれていない JSON QBE 内のプロパティ名は、この名前空間とマッチされます。つまり、これらのプロパティ名は、JSON ドキュメントとのみマッチされます。名前空間なしの XML QBE 内の要素名には名前空間が追加されないため、それらは JSON ドキュメント内のプロパティとはマッチされません。

JSON QBE で `format` を `xml` に設定した場合、QBE の文法に含まれていないプロパティ名は、`http://marklogic.com/xdmp/json/basic` 名前空間内で検索されません。このため、XML ドキュメント内の名前空間なしの要素名とマッチされ、JSON ドキュメント内のプロパティ名とはマッチされません。XML QBE で `format` を `json` に設定した場合、デフォルトの名前空間は、`http://marklogic.com/xdmp/json/basic` であり、JSON ドキュメント内でのみマッチされます。

この動作の 1 つの例外は、XML QBE でのグローバルなワードクエリです。これは、`query` の直接の子として、または `query` の直接の子である複合クエリの子として出現するワードクエリです。このようなクエリは、特定の XML 要素と JSON プロパティのいずれにもスコープ設定されないので、XML ドキュメントと JSON ドキュメントの両方とマッチされることがあります。

JSON ドキュメントの内部の XML 表現の詳細については、『Application Developer's Guide』の「Working With JSON」を参照してください。

5.10 QBE の複合クエリへの変換

QBE の主なユースケースは、開発中にクエリのプロトタイプ作成を素早く行うことです。最高のパフォーマンスを実現し、search API のすべての機能にアクセスするには、最終的に QBE を複合クエリに変換する必要があります。複合クエリは、構造化クエリとクエリオプションを組み合わせる、より詳細な表現です。

REST API および Java API には、QBE から複合クエリを生成するためのインターフェイスが含まれます。詳細については、次を参照してください。

- 『Java Application Developer's Guide』の「Convert a QBE to a Combined Query」
- 『REST Application Developer's Guide』の「Generating a Combined Query from a QBE」
- 「構造化クエリを使用した検索」(74 ページ)

6.0 cts:query 式の構成

MarkLogic サーバーでの検索では、cts:query タイプの式を使用します。この章では、さまざまなタイプの cts:query 式の作成方法、および登録済みの cts:query 式を使用して将来のクエリのパフォーマンスを向上させるために、いくつかの複雑な式を登録する方法について説明します。

MarkLogic サーバーには、cts:query 式を構成するための多くのビルトイン XQuery 関数が用意されています。さまざまな API のシグネチャと説明については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

この章は、次のセクションから構成されます。

- cts:query について
- 複数の cts:query 式の組み合わせ
- cts:properties-query または cts:document-fragment-query によるドキュメントおよびプロパティの結合
- 検索のスピードを向上させるための cts:query 式の登録
- cts:query 式への関連度の情報の追加
- cts:query コンストラクタの XML シリアライゼーション
- 例：cts:query のパーサーの作成

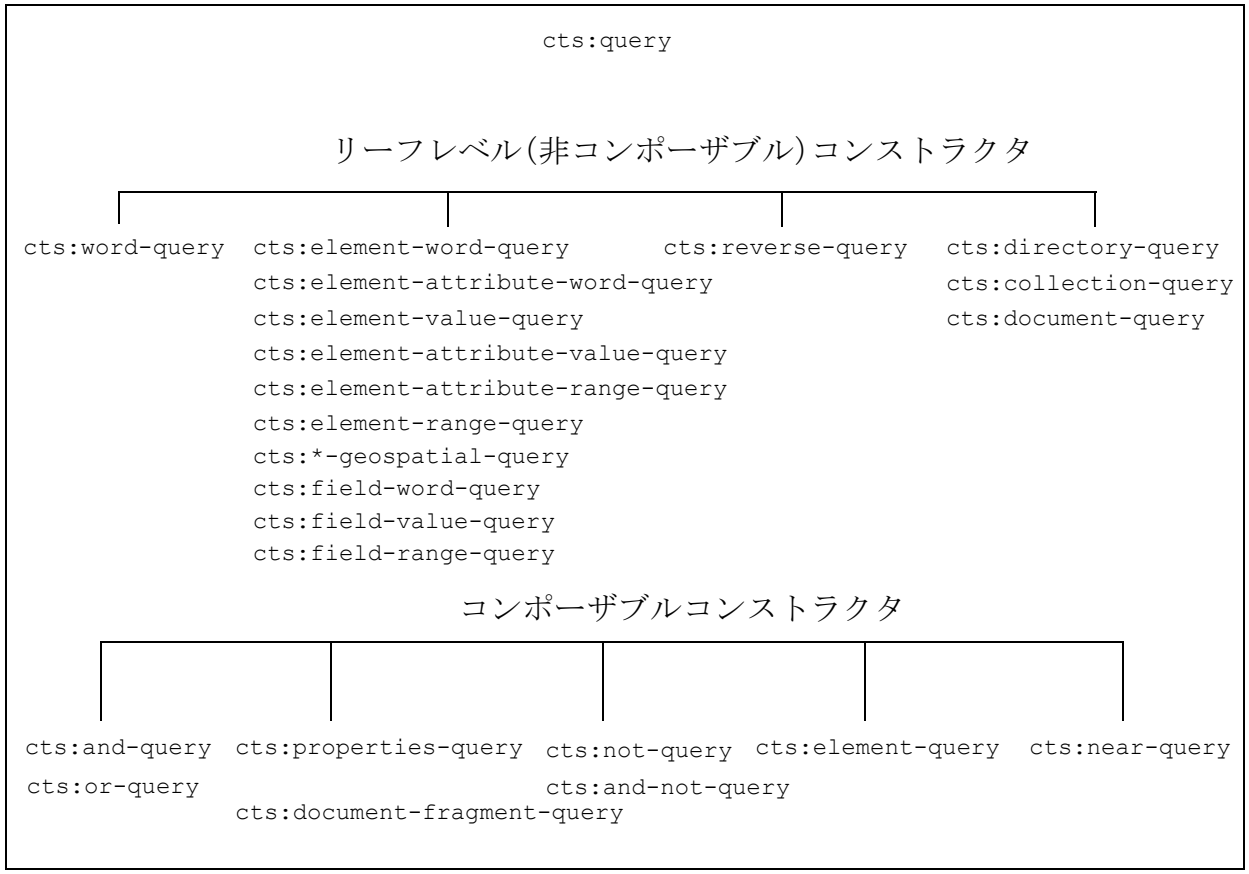
6.1 cts:query について

cts:search の 2 番目のパラメータは、cts:query タイプのパラメータです。cts:query 式の内容により、検索によってドキュメントやノードが返される際の条件が決まります。このセクションでは、cts:query について説明します。次のパートから構成されます。

- cts:query とは
- 検索を絞り込むための使い方
- cts:element-query とは
- cts:element-word-query とは
- フィールドワードクエリコンストラクタおよび値クエリコンストラクタとは
- レンジクエリコンストラクタとは
- リバースクエリコンストラクタとは
- 位置クエリコンストラクタとは
- cts:query での言語の指定

6.1.1 cts:query とは

cts:query タイプは階層となっています。このため、ユーザーは複数の式を組み合わせることで複雑な cts:query 式を構築できます。この階層には、コンポーザブルおよび非コンポーザブルな cts:query コンストラクタが含まれます。コンポーザブルコンストラクタは、複数の cts:query コンストラクタを組み合わせるために使用されます。リーフレベルコンストラクタは、他の cts:query コンストラクタと組み合わせて使用できません（ただし、コンポーザブルコンストラクタを使用して組み合わせることはできます)。次の図に、リーフレベルの、つまりコンポーザブルではない cts:query コンストラクタと、コンポーザブルな cts:query コンストラクタを示します。後者を使用して、リーフレベルコンストラクタと他のコンポーザブルな cts:query コンストラクタを組み合わせることができます。cts:query コンストラクタを組み合わせる方法の詳細については、この章の残りの部分を参照してください。



6.1.2 検索を絞り込むための使い方

cts:query API の核となるのは cts:word-query です。cts:word-query 関数は、$text パラメータと一致する単語または語句に対して true を返すので、検索が、クエリと一致するタームを含んでいるフラグメントに絞り込まれます。必要に応じて、他の cts:query API を使用して cts:word-query 式を組み合わせて、より複雑な式にすることができます。同様に、他のリーフレベルの cts:query コンストラクタを使用して検索の結果を絞り込むこともできます。

6.1.3 cts:element-query とは

cts:element-query 関数は、指定された要素とそのすべての子を検索します。この関数は、データ内の XML 構造を利用して、検索のフィールドを指定された要素階層に絞り込むために使用されます。さらに、他の cts:element-query 関数とコンポーザブルであるため、cts:query 式内で複雑な階層条件を指定できます。

例えば、次の Shakespeare データベースに対する検索は、「room」と「castle」の両方の単語を含む SPEECH 要素を持つ SCENE 要素があるすべての戯曲のタイトルを返します。

```
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]
```

このクエリは、戯曲の最初の TITLE 要素を返します。TITLE 要素は、戯曲とシーンの両方のタイトルに使用されていますが、戯曲内の最初のものがこの戯曲のタイトルになります。

cts:element-query を使用するときに、Admin 画面で word positions インデックスと element word positions インデックスの両方をオンにしている場合、いくつかの誤検出（false positive）の結果が除外されるため、複数タームのクエリ（例えば、"the long sly fox"）の多くにおいてスピードが向上します。

6.1.4 cts:element-word-query とは

cts:element-query は、要素とそのすべての子を検索しますが、cts:element-word-query は、指定された要素の直接の子テキストノードのみを検索します。例えば、次の XML 構造について考えてみましょう。

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

次のクエリは、「goodbye」が a という要素の直接のテキストノードではないため、false を返します。

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

6.1.5 フィールドワードクエリコンストラクタおよび値クエリコンストラクタとは

cts:field-word-query および cts:field-value-query コンストラクタは、フィールド内で単語（ワード）または値を検索します。フィールド値は、フィールド内のテキストすべてとして定義され、異なる要素からのテキストの間にはスペースが入ります。例えば、次の XML 構造について考えてみましょう。

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

名前を firstname lastname という形式で正規化する場合は、この構造のフィールドを作成できます。このフィールドには要素 name が含まれる一方、要素 middle は除かれます。この場合、このフィールドのインスタンスの値は、Raymond Carver になり、first と last という 2 つの異なる要素値からのテキストの間にスペースが入っています。このドキュメントに他にも同じ構造の name 要素が含まれていた場合、それらの値も同様に生成されます。このフィールドの名前が my-field である場合、cts:field-value-query("my-field", "Raymond Carver") は、この XML を含むドキュメントについて true を返します。同様に、cts:field-word-query("my-field", "Raymond Carver") も true を返します。

フィールドの詳細については、『Administrator's Guide』の「Fields Database Settings」を参照してください。フィールドのレキシコンの詳細については 「フィールド値レキシコン」（318 ページ）を参照してください。

6.1.6 レンジクエリコンストラクタとは

cts:element-range-query、cts:element-atribute-range-query、cts:path-range-query、および cts:field-range-query コンストラクタを使用すると、cts:query 式の値に制約を指定できます。レンジクエリコンストラクタでは、指定された要素または属性においてレンジインデックスが必要です。レンジクエリの詳細については、「cts:query 式でのレンジクエリの使用」（328 ページ）を参照してください。

6.1.7 リバースクエリコンストラクタとは

cts:reverse-query コンストラクタを使用すると、データベース内に格納されているクエリとノードのマッチングを行うことができます。リバースクエリは、アラートアプリケーションの基礎となります。詳細については、「アラートアプリケーションの作成」（367 ページ）を参照してください。

6.1.8 位置クエリコンストラクタとは

位置クエリコンストラクタは、位置データに関する cts:query 式を制約するために使用します。位置検索は、緯度データと経度データがマークアップされているドキュメントに対して使用され、「ニューヨーク市から 100 マイル以内にある場所に言及しているすべてのドキュメントを表示」のようなクエリに回答できます。位置検索の詳細については、「位置検索アプリケーション」（347 ページ）を参照してください。

6.1.9 cts:query での言語の指定

すべてのリーフレベルの cts:query コンストラクタは言語を認識します。ユーザーがオプションとして言語値を明示的に指定することもできますが、指定していない場合は、データベースのデフォルトの言語が使用されます。言語オプションにより、クエリのトークン化に使用される言語、ステミング検索の言語、および検索対象のコンテンツ言語が指定できます。

cts:query で言語オプションを指定するには、lang=*language_code* を使用します。*language_code* は、2 ～ 3 文字の ISO 639-1 または ISO 639-2 言語コード（http://www.loc.gov/standards/iso639-2/php/code_list.php）です。例えば、次のクエリについて考えてみましょう。

```
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")))]
```

は、フランス語のノードのみを返します。

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

cts:query の言語やコンテンツの言語に応じて、文字列が異なる方法でトークン化され、それが検索結果に影響します。言語ならびに xml:lang 属性がトークン化および検索にどのように影響するかの詳細については、「MarkLogic サーバーの言語サポート」（469 ページ）を参照してください。

6.2 複数の cts:query 式の組み合わせ

cts:query 式はコンポーザブルなので、複数の式を組み合わせて 1 つの式を作ることができます。cts:query 式は好きなだけ複雑にできます。cts:*（例えば、cts:query、cts:and-query など）のような形式で結果を返す API は、他の cts:query 式と組み合わせ、新しい式を作ることができます。このセクションには、次のパートがあります。

- cts:and-query および cts:or-query の使い方
- cts:near-query を使用した近接クエリ
- 制限付きの cts:query 式の使い方
- いかなるものともマッチしない / すべてのものとマッチする

6.2.1 cts:and-query および cts:or-query の使い方

1 つの cts:query 式の中で cts:and-query コンストラクタと cts:or-query コンストラクタを組み合わせることで、任意の複雑なブール論値を構築できます。

例えば、比較的シンプルな入れ子の cts:query 式を含む次の検索は、単語 alfa または maserati の一方を含み、かつ単語 saab または volvo の一方を含んでいるすべてのフラグメントを返します。

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

さらに、cts:and-not-query および cts:not-query を使用して、ブール論理に否定を追加できます。

6.2.2 cts:near-query を使用した近接クエリ

cts:near-query を使用して、近接性のチェックを cts:query 式に追加できます。近接クエリでは、データベース内の word positions インデックスが使用されます。cts:element-query を使用している場合は、element word positions インデックスも使用されます。近接クエリはこれらのインデックスがなくても機能しますが、これらのインデックスがあると、cts:near-query を使用するクエリのスピードが向上します。

近接クエリは、指定された距離以内でクエリにマッチするものが出現した場合に true を返します。詳細については、『MarkLogic XQuery and XSLT Function Reference』の「cts:near-query」を参照してください。

6.2.3 制限付きの cts:query 式の使い方

次の cts:query コンストラクタを使用すると、cts:query 式の対象を 1 つあるいは複数のドキュメント、1 つのディレクトリ、または 1 つあるいは複数のコレクションに制限できます。

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

これらの制限コンストラクタを使用すると、cts:search の 2 番目のパラメータとして検索結果の対象を絞り込むことができます。cts:query 式内でクエリを制限するのは、where 節で結果をフィルタリングするよりもずっと効率的であり、多くの場合、cts:search の 1 つめのパラメータで XPath を修正するよりも便利です。制限された cts:query コンストラクタを他のコンストラクタと組み合わせる場合は、cts:and-query コンストラクタまたは cts:or-query コンストラクタを使用します。

例えば、次は、検索を特定のディレクトリに制約して、cts:query とマッチするドキュメントの URI を返します。

```
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)
```

このクエリは、指定されたディレクトリ以下において、クエリ「all's well that」を満たすすべてのドキュメントの URI を返します。

注： このクエリにおいて、クエリ「all's well that」は、cts:word-query("all's well that") と等価です。

6.2.4 いかなるものともマッチしない / すべてのものとマッチする

空の cts:word-query は常にどのフラグメントともマッチせず、空の cts:and-query は常にすべてのフラグメントとマッチします。このため、次のクエリは true になります。

```
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
```

空の cts:word-query の使い方の 1 つとしては、エンドユーザーがタームを入力する検索ボックスがあります。何も入力せずに実行ボタンが押した場合、この cts:search は該当なしを返します。

すべてのものとマッチする空の cts:and-query は、すべてのものとマッチする cts:query が必要な場合に役に立つことがあります。

6.3 cts:properties-query または cts:document-fragment-query によるドキュメントおよびプロパティの結合

cts:properties-query を使用して、プロパティドキュメント内のコンテンツとマッチできます。ドキュメントを対象に検索する場合、cts:properties-query は、ドキュメントの URI にあるプロパティドキュメントの中を検索します。cts:properties-query は、プロパティドキュメントとそれに対応するドキュメントを結合します。cts:properties-query は、パラメータとして cts:query を取り、そのクエリはプロパティドキュメントとのマッチに使用されます。cts:properties-query は、コンポーザブルなので、他の cts:query コンストラクタと組み合わせて任意の複雑なクエリを作成できます。

cts:search 内で cts:properties-query を使用すると、ドキュメント内のコンテンツと対応するプロパティドキュメント内のコンテンツを結合した結果を返すクエリを簡単に作成できます。例えば、ある本の章に該当するドキュメントがあり、このドキュメントには出版社情報を含んだプロパティがあるとします。このような場合、次の例のように特定の出版社に関する cts:query とマッチしたドキュメントを返すクエリを作成できます。

```
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") )) )
```

このクエリは、a small good thing という語句を含み、かつそれに対応するプロパティドキュメント内の publisher 要素が My Press という値であるすべてのドキュメントを返します。

同様に、cts:document-fragment-query を使用して、プロパティを検索する際にプロパティにドキュメントを結合できます。

6.4 検索のスピードを向上させるための cts:query 式の登録

複雑な cts:query 式を繰り返し使用し、それらをフィルタリングなしの cts:query コンストラクタとして使用したい場合は、この cts:query 式を登録しておけば後で再利用できます。cts:query 式を登録すると事前評価済みの式が格納されます。このため今後この式をクエリで使用した場合、より高速になります。フィルタリングなしコンストラクタは、インデックスから直接結果を返し、検索に対するすべての候補フラグメントを返しますが、各フラグメントが検索条件を完全に満たしていることを検証するポストフィルタリングは実行しません。フィルタリングなしの検索の詳細については、『Query Performance and Tuning Guide』の「Using Unfiltered Searches for Fast Pagination」を参照してください。

このセクションでは、登録済みクエリについて説明し、その使用方法の例をいくつか示します。次のトピックから構成されます。

- クエリ API の登録
- フィルタリングなしでの使用が必要な場合
- 登録はシステムの再起動で削除される
- 登録済みクエリの ID の格納
- 登録済みクエリおよび関連度の計算
- 例：cts:query 式の登録と使い方

6.4.1 クエリ API の登録

cts:query 式のフィルタリングなしの検索を登録して再利用するには、次の XQuery API を使用します。

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

これらの関数のシンタックスについては、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

6.4.2 フィルタリングなしでの使用が必要な場合

登録済みクエリはフィルタリングなしコンストラクタでのみ使用できます。登録済みクエリをフィルタリング付きコンストラクタとして使用すると、XDMP-REGFLT 例外がスローされます。フィルタリングなしコンストラクタを指定するには、cts:registered-query で「unfiltered」オプションを使用します。フィルタリングなしの検索の詳細については、『Query Performance and Tuning Guide』の「Using Unfiltered Searches for Fast Pagination」を参照してください。

6.4.3 登録はシステムの再起動で削除される

登録済みクエリは、メモリキャッシュ内にしか格納されません。キャッシュが大きくなりすぎた場合、いくつかの登録済みクエリがキャッシュから溢れてしまうことがあります。また、MarkLogic サーバーが停止または再起動した場合、登録されたクエリは失われ、再登録する必要があります。

cts:search 内で cts:registered-query を呼び出そうとしたとき、その時点でクエリが登録されていなかった場合、XDMP-UNREGISTERED 例外がスローされます。登録済みクエリが使用時に毎回登録済みであることは保証されないため、cts:registered-query の呼び出しに関して try/catch を使用し、XDMP-UNREGISTERED 例外がスローされた場合は catch でクエリを再登録することをお勧めします。

例えば、次のサンプルコードは、XQuery で try/catch 式とともに使用される cts:registered-query 呼び出しを示しています。

```
(: 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 )
}
```

このコードは、ある程度単純化されています。つまり、XDMP-UNREGISTERED 例外をキャッチし、新しい登録済みクエリの ID をレポートするだけです。登録済みクエリを使用するアプリケーションでは、新しい登録済みの ID でクエリを再実行したほうがいいでしょう。また、このサンプルでは、XQuery で try/catch を実行しています。XCC を使用して MarkLogic サーバーに対してクエリを発行する場合は、ミドルウェアの Java または .NET レイヤーで try/catch を実行することもできます。

6.4.4 登録済みクエリの ID の格納

cts:query 式を登録すると、cts:register 関数は整数を返します。これは、登録済みクエリの ID です。cts:register 呼び出しが返した後、システムに対してクエリを実行して登録済みクエリの ID を確認する方法はありません。このため、ID をどこかに格納したほうがいいでしょう。これは、ミドルウェアレイヤー（XCC を使用して、MarkLogic サーバーに対してクエリを発行する場合）、あるいは MarkLogic サーバー内のドキュメントに格納できます。

登録済みクエリの ID は実際のクエリのハッシュに基づいて生成されるので、同じクエリを複数回登録しても ID は同じになります。登録済みクエリの ID は、クラスタ全体におけるデータベースに対するすべてのクエリで有効です。

6.4.5 登録済みクエリおよび関連度の計算

登録済みクエリを使用した検索では、登録済みではないクエリを使用した同等の検索とは、結果のスコアが異なります。これは、登録済みクエリが関連度の計算で 1 つのタームとして扱われるためです。関連度の計算の詳細については、「関連度スコア：概要とカスタマイズ」（289 ページ）を参照してください。

6.4.6 例：cts:query 式の登録と使い方

登録済みクエリを実行するには、まずクエリを登録してから、その ID を指定して登録済みクエリを実行します。このセクションでは、クエリを登録してから、登録済みクエリを実行するためのいくつかのステップの例について説明します。

1. 最初に、次の例のように実行する `cts:query` 式を登録します。

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

2. 最初のステップでは、整数が返されます。その整数値を記録します（例えば、ドキュメント内に格納します）。

3. 次のように、その整数値を使用して登録済みクエリによる検索を実行します（「`unfiltered`」オプションを指定します）。

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

6.5 cts:query 式への関連度の情報の追加

リーフレベルの `cts:query` API（`cts:word-query`、`cts:element-word-query` など）には、重みパラメータがあります。このパラメータを使用すると、クエリのマッチによって生成されるスコアに、ある倍率を適用できます。これを使用して、特定のクエリの重み係数を増減できます。スコア、重み、および関連度の計算の詳細については、「関連度スコア：概要とカスタマイズ」（289 ページ）を参照してください。

6.6 cts:query コンストラクタの XML シリアライゼーション

cts:query の XML シリアライゼーションを作成できます。XML シリアライゼーションは、cts:reverse-query コンストラクタを使用するアラートアプリケーションによって使用され、また cts:query に対するさまざまなプログラミングタスクを実行するのにも役立ちます。アラートアプリケーション（「アラートアプリケーションの作成」（367 ページ）を参照）はノードにマッチするクエリを見つけ、何らかのアクションを実行します。このセクションでは、シリアライズされた XML について説明します。次のパートから構成されます。

- cts:query の XML へのシリアライズ
- XML から cts:query を構築する関数

6.6.1 cts:query の XML へのシリアライズ

シリアライズされた cts:query は、スキーマ *<marklogic-dir>*/Config/cts.xsd に準拠した XML となります。このスキーマは、http://marklogic.com/cts 名前空間内にあり、この名前空間は、cts プレフィックスにバインドされています。この XML は直接構築することもできますが、要素のコンテキスト内で何らかの cts:query 式を使用した場合、MarkLogic サーバーによってその cts:query が XML に自動的にシリアライズされます。次の例について考えてみましょう。

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

上の式を実行すると、次の XML にシリアライズされます。

```
<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>
```

アラートアプリケーションを使用している場合は、この XML をデータベースに格納して、cts:reverse-query コンストラクタを含む検索とマッチできます。アラートの詳細については、「アラートアプリケーションの作成」（367 ページ）を参照してください。

6.6.2 cts:annotate による任意の注釈の追加

cts:annotate 要素で cts:query XML に注釈を付けることができます。cts:annotate 要素は、cts:query XML 内で任意の要素の子にすることができ、任意の有効な XML コンテンツ（例えば、単一のテキストノード、単一の要素、複数の要素、複雑な要素など）から構成できます。MarkLogic サーバーは、クエリ XML を処理するときにこれらの注釈を無視しますが、このような注釈は多くの場合、アプリケーションで役立ちます。例えば、クエリの出所に関する情報や、アプリケーションの特定の部分で使用する、または使用しないクエリの一部に関する情報などを格納できます。次に、cts:annotation 要素を含む XML の例をいくつか示します。

```
<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>
```

cts:annotate を使用して、文字列から cts:query を生成する関数にオリジナルのクエリ文字列を格納する他の例については、「cts:query コンストラクタの XML シリアライゼーション」（250 ページ）の最後の部分を参照してください。

6.6.3 XML から cts:query を構築する関数

cts:query 関数を使用することで、cts:query の XML シリアライゼーションをシリアライズされていない cts:query に戻すことができます。例えば、シリアライズされた cts:query を次のように cts:query に戻すことができます。

```
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 例：cts:query のパーサーの作成

次のサンプルコードは、二重引用符は語句（フレーズ）であるとパースし、他の 1 つあるいは複数のスペースで区切られているものはすべてタームと見なすシンプルなクエリ文字列パーサーを示しています。必要な場合は、同じデザインパターンを使用して、より複雑なパーシング（例えば、OR 処理や NOT 処理）を実行する他のロジックを追加できます。

```
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)
```

これにより、次が返されます。

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

これで、上記で生成されたトークン化された XML から cts:query 式を生成できるようになりました。これは、次のように、cts:and-query ですべてのタームを構成します（上記の local:get-query-tokens 関数が利用できると仮定しています)。

```
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)
```

これにより、次が返されます（読みやすくするために、スペースと改行を追加しています)。

```
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)
    ), ()) ,
  () )
```

これで、生成された cts:query 式を cts:search に追加できるようになりました。

同様に、次のようにシリアライズされた cts:query を生成できます
(local:get-query-tokens 関数が利用できると仮定しています)。

```
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)
```

これにより、次の XML シリアライゼーションが返されます。

```
<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 クエリオプションを使用した検索のカスタマイズ

XQuery Search API、REST API、または Java API を使用する場合、クエリオプションを使用して検索をカスタマイズおよび制御できます。この章では、主要なクエリオプションの機能を取り上げます。次のトピックを扱います。

- はじめに
- デフォルトのクエリオプションの取得
- クエリオプションにエラーがないかどうかをチェックする
- 制約オプション
- 演算子オプション
- 戻り値のオプション
- 検索可能な式のオプション
- フラグメントスコープのオプション
- 結果のスニペットの修正
- デコレータによる検索結果のカスタマイズ
- その他の検索オプション
- クエリオプションの例

各オプションのシンタックスの詳細については、『MarkLogic XQuery and XSLT Function Reference』の `search:search` 関数の項または『REST Application Developer's Guide』の付録を参照してください。

7.1 はじめに

クエリオプションを使用すると、コンテンツおよび値の検索のさまざまな要素を制御できます。例えば、検索のスコープの制限、文字列検索の文法のカスタマイズ、ソート順序の定義、検索結果のコンテンツと形式の指定などを行うことができます。

MarkLogic サーバーでは、検索にカスタムのクエリオプションが利用されない場合には、定義済みのデフォルトのクエリオプションが使用されます。REST API や Java API では、デフォルトのクエリオプションを修正できます。カスタムのクエリオプションを定義して個別の検索に適用することで、デフォルトのオプションを上書きできます。

すべての API において、クエリオプションは XML で指定できます。REST API と Java API では、JSON もサポートしています。XML クエリオプションは常に、次の名前空間内の `search:options` 要素として表現されます。

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

XQuery、REST、および Java API による検索操作のほとんどでは、クエリのオプションが利用できます。以下のようなものがあります。

- XQuery：関数 `search:search`、`search:resolve`、`search:values`、`search:parse`
- REST：`/search`、`/values`、`/keyvalue` サービス
- Java：`com.marklogic.client.query.QueryManager` クラスを使用して実行される検索。

7.2 デフォルトのクエリオプションの取得

検索にクエリオプションを含めていない場合、デフォルトのクエリオプションが使用されます。XQuery または REST API を使用して、デフォルトのクエリオプションの定義を取得できます。

XQuery を使用してデフォルトのクエリオプションを取得するには、`search:get-default-options` を呼び出します。

REST を使用してデフォルトのクエリオプションを取得するには、`/config/query/default` に対する GET リクエストを実行します。詳細については、『REST Client API Reference』を参照してください。

7.3 クエリオプションにエラーがないかどうかをチェックする

クエリオプションはかなり複雑になることがあります。XQuery、REST、および Java API では、クエリオプションのエラーをチェックできます。

XQuery では、関数 `search:check-options` を使用します。この関数はオプションを検証し、エラーがあればレポートします。オプションが有効である場合、空を返します。エラーは、見つかった場合、1 つあるいは複数の `search:report` ノードの形式で返されます。

REST API では、`/config/query` サービスを使ってクエリオプションを指定したときに同等のチェックを実行できます。Java API では、`com.marklogic.client.admin.QueryOptionsManager.writeOptions` を呼び出したときにこのチェックが実行されます。

検索のたびにオプションをチェックすると、クエリが遅くなるので、クエリオプションの検証は開発時にのみ使用することをお勧めします。`search:options` ノードで `debug` オプションを true に設定して、`search:check-options` の出力をレスポンスの一部として返すこともできます。

一般的な MarkLogic XQuery のデザインパターンとしては、`$debug` オプション（デフォルトは `false` です。`true` の場合、`search:check-options` が実行されます）をコードに追加するか、`debug` オプションをクエリオプションに追加します。開発時は `$debug` 変数を true に設定し、実稼働時は false に設定します。

7.4 制約オプション

制約とは、search、REST、および Java API で使用される、データベースのスライスに基づいた検索の制約方法を定義することです。制約は、検索操作に、データベースに関する情報を提供し、データベースのそれらの細部に対してクエリを実行する方法を指定します。

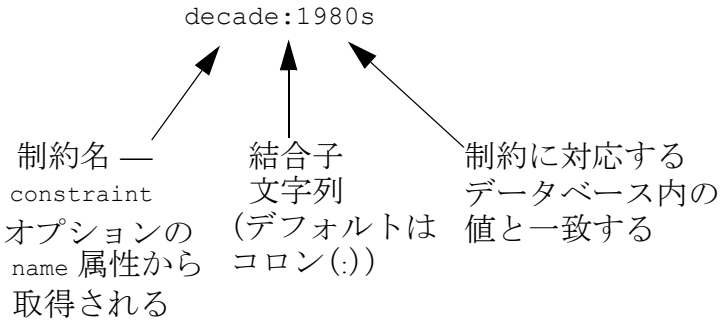
制約は、レンジインデックス、データベース内に存在するその他の設定オブジェクト（ワードレキシコン、コレクションレキシコン、フィールドなど）、およびデータベース内のドキュメントの構造（要素値、属性値、単語など）を利用します。制約は、主に次の目的で使用します。

- 文字列クエリにおける制約を指定するための手段を提供する。検索パーシングおよび文法については、「デフォルトの文字列クエリの文法」（61 ページ）を参照してください。
- アプリケーションにおいてファセットを作成するための情報を返す。ファセットについては、「制約付き検索とファセットナビゲーション」（26 ページ）を参照してください。
- `search:suggest` によって作成される検索候補を向上させる。検索候補については、「検索ターム入力支援」（28 ページ）を参照してください。

各制約には名前が付けられます。この名前は、クエリオプション内のすべての演算子および制約で一意にする必要があります。またスペースは使用できません。文字列クエリにおいてクエリテキストとして制約を指定するときは、検索文法における制約として名前を使用し、その後に、文字列クエリの文法で定義されている結合子記号（デフォルトではコロン（:））を指定します。結合子文字列は、制約（または演算子）と値を結合します。例えば、次のクエリテキストについて考えてみましょう。

```
decade:1980s
```

これは、decade という制約と、値 1980s を指定しています。次の図に、制約のクエリテキストの各部分を示します。



検索文法の詳細については、「自動クエリテキストパーシング機能および文法」(24 ページ)および「デフォルトの文字列クエリの文法」(61 ページ)を参照してください。

次の表に、クエリオプションで構築できる制約のタイプのリストを示します。

制約	説明	制約に相当する cts:query	ファセットに相当するレキシコン API
value	要素値、属性値、またはフィールド値で制約します。	cts:element-value-query, cts:element-attribute-value-query, cts:field-value-query	値制約に対するファセットはありません。

	value 制約の例：
	
	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="my-value">`
	`  <value>`
	`  <element ns="my-namespace" name="my-localname"/>`
	`  </value>`
	`  </constraint>`
	`</options>`
	
	詳細については、「値制約の例」(263 ページ)を参照してください。		

<table>
<tr><th>制約</th><th>説明</th><th>制約に相当する
cts:query</th><th>ファセットに相当する
レキシコン API</th></tr>
<tr><td rowspan="2">word</td><td>element、attribute、または field のいずれかのワードクエリで制約します。</td><td>cts:element-word-query、
cts:element-attribute-word-query、
cts:field-word-query</td><td>ワード制約に対するファセットはありません。</td></tr>
<tr><td colspan="3">word 制約の例：

<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>
詳細については、「単語（ワード）制約の例」（264 ページ）を参照してください。</td></tr>
<tr><td rowspan="2">collection</td><td>データベースでコレクションレキシコンをオンにする必要があります。</td><td>cts:collection-query</td><td>cts:collections</td></tr>
<tr><td colspan="3">collection 制約の例：

<pre>
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="subject">
    <collection prefix="/my-collections/"/>
  </constraint>
</options>
</pre>
詳細については、「コレクション制約の例」（265 ページ）を参照してください。</td></tr>
</table>

制約	説明	制約に相当する cts:query	ファセットに相当する レキシコン API
range	基盤となるレンジインデックスがデータベース内に存在する必要があります。すべてのレンジ制約は、要素または属性値、あるいはフィールド値についてタイプを認識します。また、制約は、オプションで bucket 要素または computed-bucket 要素を含むことができます。例については、「バケットレンジ制約の例」(266 ページ)、「バケットの例」(56 ページ)、「計算されるバケットの例」(58 ページ)、および『MarkLogic XQuery and XSLT Function Reference』の search:search options ノードの説明を参照してください。	次のようなレキシコン API： cts:element-range-query、cts:element-attribute-range-query、cts:path-range-query、および cts:field-range-query	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
container	qtext を特定の XML 要素または JSON キーに制限します。最高のパフォーマンスを得るには、データベースで位置インデックスがオンになっている必要があります。	cts:element-query	コンテナ制約に対するファセットはありません。

	element-query 制約の例：
	
	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <constraint name="sample-element-constraint">`
	`  <container>`
	`  <element name="title" ns="http://my/namespace" />`
	`  </container>`
	`  </constraint>`
	`</options>`		

<table>
<tr><th>制約</th><th>説明</th><th>制約に相当する
cts:query</th><th>ファセットに相当する
レキシコン API</th></tr>
<tr><td rowspan="2">properties</td><td>対応するプロパティドキュメントにおいてマッチを探します。</td><td>cts:properties-query</td><td>プロパティ制約に対するファセットはありません。</td></tr>
<tr><td colspan="3">properties 制約の例：

<pre>
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="sample-property-constraint">
    <properties />
  </constraint>
</options>
</pre></td></tr>
<tr><td rowspan="2">geo-attr-pair
geo-elem-pair
geo-elem</td><td>これらの位置情報制約は、位置情報データにおいてマッチを探します。ファセットとして使用するには、<constraint> 要素に <heatmap> という子が必要です。</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">geo-* 制約の例：

<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>
</table>

制約	説明	制約に相当する cts:query	ファセットに相当する レキシコン API
custom	パース用およびファセット作成用にカスタム関数を実装することにより、自分が好きなタイプの制約を作成できます。例については、「カスタム制約の作成」(34 ページ) を参照してください。	カスタムコードで実装する内容によって異なります。	カスタムコードで実装する内容によって異なります。

制約は、高速化を目的としています。ファセットを作成する場合、個々の値とそれらの正確な件数を素早く生成する必要があります。このため、ファセットを許可する制約の場合、制約が適用される要素または属性にレンジインデックスが必要です。あるいは特定のレキシコンがデータベース内に存在する必要があります。他の制約（value および word 制約）は、特別なインデックスが存在しないので、ファセットの作成に使用できません。

MarkLogic サーバーでは、（search:parse、search:search などを使用して）クエリ内の制約をパースするとき、結合子文字列を探し、結合子文字列の右側にある値を適用することで、cts:query のように値をパースします。クエリオプションで制約が定義されず、値が 1 つの検索タームである場合、結合子文字列は検索タームの一部として扱われます。例えば次のようになります。

```
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>
```

制約がクエリオプションで定義されず、値が引用符で囲まれたテキストである場合、search API は、クエリのパーシング時に制約と結合子を無視しますが、元のテキストを属性として保存します。例えば次のようになります。

```
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>
```

次の例は、次のタイプの制約を示しています。

- 値制約の例
- 単語（ワード）制約の例
- コレクション制約の例
- バケットレンジ制約の例
- 完全一致（バケット化されていない）レンジ制約の例

カスタム制約の例については、「カスタム制約の作成」（34 ページ）を参照してください。

7.4.1 値制約の例

次のクエリオプションでは、2 つの値制約、つまり、要素に対する値制約と属性に対する値制約を定義しています。

```
<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>
```

これらの制約を使用する場合は、次のような文字列クエリを使用してこれらの制約を使用できます。

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

上記の両方のクエリは、次のドキュメントとマッチします。

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

7.4.2　単語（ワード）制約の例

次のクエリオプションでは、次の 2 つの単語（ワード）制約を定義しています。つまり、http://authors-r-us.com 名前空間内の <name/> 要素に対する単語制約と、my-field フィールドに対する単語制約です。これは、cts:element-word-query に対するものと、cts:field-word-query に対するものです。

```
<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>
```

これらの制約を使用する文字列クエリおよび構造化クエリを作成できます。例えば、次の文字列クエリは、name 制約を使用しています。

```
name:raymond
```

このクエリは、パースされると、cts:element-word-query になります。

```
<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>
```

このクエリは、次のドキュメントとマッチします（cts:word-query("raymond") がマッチするため）。

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

同様に、description 制約を使用する次の文字列クエリは、パースされると、cts:field-word-query になります。

```
description:author
```

このクエリは、name 要素が my-field というフィールドの定義に含まれる場合に上記のドキュメントとマッチします。フィールドの詳細については、『Administrator's Guide』の「Fields Database Settings」を参照してください。

単語制約は、構造化クエリでも使用できます。例えば、次の構造化クエリは `name:raymond` 文字列クエリと等価です。

```
<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>
```

詳細については、「構造化クエリを使用した検索」（74 ページ）を参照してください。

7.4.3 コレクション制約の例

次のクエリオプションは、コレクション制約を定義しています。コレクション制約を使用すると、検索を、指定されたコレクション内のドキュメントに制約できます。

注： コレクション制約を使用するには、データベース内のコレクションレキシコンをオンにする必要があります。コレクションレキシコンがオンになっていない場合、コレクション制約を含むクエリを使用すると、例外がスローされます。

コレクション制約の定義に `prefix` 属性を含めた場合、コレクション名は、制約の値と連結された `prefix` から生成されます。

コレクション制約の用途の 1 つとして、コレクションに基づくファセットナビゲーションが挙げられます。例えば、科目に基づくコレクション（例えば、`history` や `math` など）がある場合、コレクション制約を使用して検索をいずれかの科目に絞り込むことができます

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

データベース内のすべてのドキュメントに、次のような、文字列 `/my-collections/` で始まるコレクション URI があると仮定します。

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

この場合、次のクエリテキストの例は、対応するコレクション内のドキュメントとマッチされます。

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

データベースの指定したコレクション内にドキュメントが含まれていない場合、検索はマッチなしを返します。コレクションについては、「コレクション」（416 ページ）を参照してください。

collection-constraint-query を使用すると、構造化クエリでもコレクション制約を使用できます。詳細については、「構造化クエリを使用した検索」（74 ページ）を参照してください。

7.4.4 バケットレンジ制約の例

レンジ制約は、対応するレンジインデックスをデータベース内に持つ型付き要素、要素属性、または JSON キー / バリューに対して動作します。正しいレンジインデックスがない場合、レンジ制約を使用しているクエリは、ランタイム例外をスローします。

レンジ制約の値は、要素、属性、またはキーの個別の値すべて、または指定されたバケット（値の名前付きレンジ）とマッチできます。バケットには 2 つのタイプがあり、range という制約指定の中の bucket 要素および computed-bucket 要素で指定されます。bucket という指定は絶対的なレンジを使用し、computed-bucket という指定はある時点における相対的なレンジを使用します。computed-bucket レンジ制約の詳細については、「計算されるバケットの例」（58 ページ）を参照してください。

次の例では、search:parse を、bucket レンジ制約を定義するクエリオプションと組み合わせて使用しています。次の例は、Oscars サンプルアプリケーションから生成されたものです。このアプリケーションは、Application Builder を使用して構築されています。

```
xquery version "1.0-ml";

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

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

このクエリは、次の cts:query を返します。

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

このクエリを実行できる対象のサンプルデータについては、Application Builder から生成した Oscars サンプルアプリケーションを参照してください。他のレンジ制約の例については、「バケットの例」（56 ページ）、「計算されるバケットの例」（58 ページ）、および次の例を参照してください。

7.4.5 完全一致（バケット化されていない）レンジ制約の例

次の例は、Oscars サンプルアプリケーションに対する完全一致の年のレンジ制約を示しています。1964 という年と一致する結果を返します。出力を参照するには、Query Console を使用して、Oscars データベースに対してこのクエリを実行します。

```
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)
```

これらのクエリオプションは Java API および REST API で試すこともできます。詳細については、『Java Application Developer's Guide』の「Apply Dynamic Query Options to Document Searches」または『REST Application Developer's Guide』の「Specifying Dynamic Query Options with Combined Query」を参照してください。

7.5 演算子オプション

検索演算子を使用すると、文字列クエリの文法内で、ランタイムでユーザー制御の設定および検索の選択肢を提供する演算子を定義できます。典型的な検索演算子としてはソートを制御するものがあります。これにより文字列クエリまたはクエリテキスト内で直接ソート順序を指定できます。

例えば、次のオプションの XML では、`sort` という演算子を定義しています。この演算子を使用して、関連度または日付でソートできます。

```
<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>
```

クエリオプション内のこの演算子定義を使用すると、次のようなテキストを文字列クエリに追加できます。これにより、MarkLogic サーバーが文字列をパースし、演算子の指定に従ってソートします。

```
sort:date

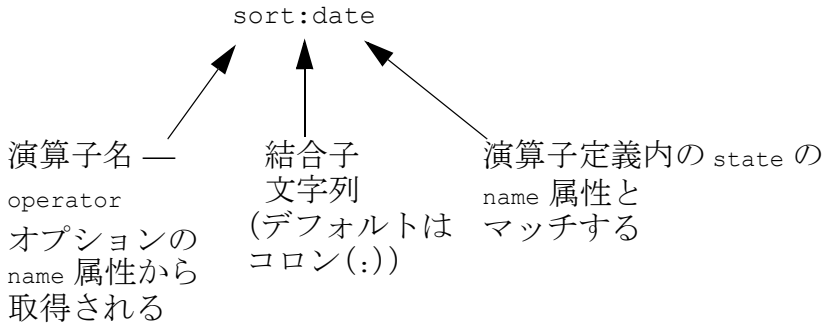
sort:relevance
```

各演算子には名前が付けられます。名前は、クエリオプション内のすべての演算子および制約にわたって一意である必要があります。文字列クエリで演算子を指定するときは、検索文法での演算子として名前を使用し、その後に、`apply="constraint"` 結合子文字列（デフォルトではコロン（:））を指定します。結合子文字列は、演算子（または制約）と値を結合します。例えば、次のクエリテキストについて考えてみましょう。

```
sort:date
```

これは、`sort` という演算子と、値 `date` を使用することを指定しています。`operator-state` 要素を使用すると、構造化クエリにも演算子の設定を含めることができます。詳細については、「operator-state」（177 ページ）を参照してください。

次の図は、演算子のクエリテキストの各部分を示しています。



検索文法の詳細については、「自動クエリテキストパーシング機能および文法」（24 ページ）および「デフォルトの文字列クエリの文法」（61 ページ）を参照してください。

`search:state` 要素は `search:operator` 要素の子であり、`search:state` 要素の子として次のオプションの XML 要素が使用できます。

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

演算子は、制約と同じシンタックスを使用しますが、返される検索結果に加えて、検索の他の要素（例えば、ソート順序など）を制御します。

7.6 戻り値のオプション

XQuery Search API の関数 search:search、REST API の /search サービスに対する GET リクエスト、Java API のメソッド com.marklogic.client.query.QueryManager.search によって返される結果など、検索レスポンスの内容を制御する多くのオプションがあります。これらには、次のブール型オプションが含まれます。

- return-aggregates
- return-constraints
- return-facets
- return-frequencies
- return-metrics
- return-plan
- return-qtext
- return-query
- return-results
- return-similar
- return-values

これらのオプションのいずれかを true に設定すると、指定した情報が、検索が返す search:response に含められます。false に設定すると、その情報はレスポンスに含まれません。例えば次のクエリでは、結果としてクエリの統計およびファセットは返しますが、検索がマッチした内容自体は返さないように指定しています。

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

必要な部分のみが処理されるので、上記の例のようにマッチ部分あるいは他のデータを返さない場合、検索が高速に実行されます。

各 return オプションの詳細とそのデフォルト値については、『MarkLogic XQuery and XSLT Function Reference』の search:search 関数の項を参照してください。

7.7 検索可能な式のオプション

searchable-expression オプションを使用して、検索の対象とする式および検索結果で返す内容を指定できます。この式は、cts:search の最初のパラメータに対応し、完全に検索可能な式である必要があります。完全に検索可能な式の詳細については、『Query Performance and Tuning Guide』の「Fully Searchable Paths and cts:search Operations」を参照してください。

デフォルトでは、検索はデータベース全体（fn:collection()）が対象です。searchable-expression は、フラグメントルートよりも下の検索も許可されますが、ほとんどの場合、フラグメントルートを対象として検索します。

次の例に、CITATION 要素と html 要素の両方を対象として検索する検索可能な式を示します。

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

式が完全に検索可能であるわけではない場合、ランタイムで XDMP-UNSEARCHABLE 例外をスローします。

7.8 フラグメントスコープのオプション

fragment-scope オプションを指定して、検索または制約の対象となるフラグメントを制御できます。fragment-scope は、documents または properties にすることができます。デフォルトでは、スコープは documents です。fragment-scope が documents である場合はドキュメントフラグメントを対象として検索し、fragment-scope が properties である場合はプロパティフラグメントを対象として検索します。

fragment-scope のオプションには、2 つのタイプがあります。検索と検索内の制約の両方に適用されるグローバルフラグメントスコープと、特定の制約に適用されるローカルフラグメントスコープです。グローバルな fragment-scope は、<options> の子として指定し、ローカルフラグメントスコープは、<term> の子または制約の一種（例えば、<range>、<value>、または <word> の子）として指定します。あらゆるローカルフラグメントスコープは、グローバルフラグメントスコープよりも優先されます。

レンジ制約のローカルフラグメントスコープ properties を、グローバルフラグメントスコープ documents と一緒に使用すると、プロパティフラグメント内のデータに関するファセットを作成できます。例えば、次のクエリは、ドキュメントから結果、prop:last-modified システムプロパティから dateTime の last-modified ファセットをそれぞれ返します。

```
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>)
```

<term> 定義内にフラグメントスコープを設定すると、指定したスコープ内でタームクエリが評価されます。ただし、次の状況では、ローカルフラグメントスコープが無視されます。

- <term> 定義にインラインの単語、値、レンジ制約、または他の制約への参照が含まれる。この場合、制約定義のフラグメントスコープが適用されます。
- <term> 定義で、apply/ns/at を使用したカスタムのターム処理関数が指定されている。この場合、カスタム関数がスコープを制御します。

7.9 結果のスニペットの修正

transform-results オプションを使用すると、アプリケーションのスニペットコードに対するオプションを指定できます。スニペットは、検索結果の文章（短縮され、強調表示された要約）であり、通常は検索結果として表示されます。スニペットは、マッチした検索結果ノードを取得し、変換コードを適用することで作成されます。この変換により、通常は、必要とされる結果の一部が結果ページに表示されます。その際クエリがマッチした部分が強調表示されるほか、その周囲のテキストも表示されます。多くの場合、それ以外の部分は破棄されます。このセクションでは、search API からのスニペット結果を制御および修正するための次の方法について説明します。

- transform-results のオプションの指定
- transform-results での独自のコードの指定

7.9.1 transform-results のオプションの指定

デフォルトでは、search API には、検索がマッチした部分を取得して、検索結果として使用されるスニペットに変換する独自のコードがあります。デフォルトでは、search API は、transform-results オプションで apply="snippet" 属性を使用します。スニペットはアプリケーションによって大きく異なり、ビルトインの apply="snippet" オプションには、transform-results options ノードで制御できるパラメータがいくつかあります。

次に、デフォルトの transform-results のオプションノードを示します。

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

次の表で、apply="snippet" の場合の transform-results のオプションについて説明します。これらのそれぞれは、検索のランタイムにおいてオプションの値を自分で指定することによって設定可能です。

transform-results 子要素	説明
per-match-tokens	スニペット内で強調表示されたタームの周囲に表示される、1 つのマッチノードあたりのトークン（通常は単語）の最大数。
max-matches	スニペット内に表示される、強調表示されるタームを含むノードの最大数。
max-snippet-chars	スニペットの最大文字数。

preferred-elements	マッチを探すためにスニペットのアルゴリズムが最初に対象とする要素を 0 個以上指定します。例えば、TITLE 要素のマッチを優先させる場合、次の例のように、優先される要素として TITLE を指定します。
	`<transform-results apply="snippet">`
	`  <preferred-elements>`
	`  <element ns="" name="TITLE"/>`
	`  </preferred-elements>`
	`</transform-results>`

ビルトインのスニペット作成オプションは他にも次の 3 つがあり、これらは、transform-results options ノードの属性として公開されています。

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

注： transform-results 要素の apply 属性は、search:search および search:resolve 関数にのみ適用できます。search:snippet は、常にデフォルトのスニペット作成オプション snippet を使用し、apply 属性で指定されたあらゆる内容を無視します。

スニペット作成オプション apply="raw" は、次のようになります。

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

apply="raw" オプションにより、search:response の出力としてノード全体（強調表示なし）が返されます。その後、このノードを取得し独自の変換を実行するか、あるいはこのまま返すだけにするかなど、アプリケーションに応じた任意の処理ができます。

スニペット作成オプション apply="empty-snippet" は、次のようになります。

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

apply="empty-snippet" オプションは、結果ノードを返しませんが、search:result ごとに空の search:snippet 要素を返します。search:result ラッパー要素には、マッチした検索ノードに対する独自の変換に必要な情報（例えばノードの URI やパス）があります。このため、search API 以外の独自のコードを記述して結果を処理できます。

スニペット作成オプション apply="metadata-snippet" は、次のようになります。

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

apply="metadata-snippet" オプションは、指定された優先される要素をプロパティドキュメントから返します。<preferred-elements> 要素が指定されていない場合、metadata-snippet オプションは、そのスニペットの prop:last-modified 要素を返します。prop:last-modified 要素が存在しない場合、空のスニペットを返します。

7.9.2 transform-results での独自のコードの指定

デフォルトのスニペットコードではアプリケーションの要件を満たせない場合、この検索には独自のスニペットコードを使用できます。

独自のスニペットコードを指定するには、「オプションおよび拡張による検索のカスタマイズ」（28 ページ）で説明されているデザインパターンを使用してください。実装する関数には、次のシグネチャとの互換性がある必要があります。

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

search API は、この関数に結果ノードおよび `cts:query` の XML 表現を渡します。その後、カスタム関数でこれを適宜変換します。カスタム変換を指定する options ノードは、次のようになります。

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

カスタム関数を使用する場合は、オプションを `transform-results` オプションの子として追加することでカスタム関数に渡すこともできます。search API は、`transform-results` 要素をカスタム関数に渡します。オプションの一部だけを使用したい場合は、オプションをパースして必要な任意の内容を抽出するコードを記述します。

7.10 デコレータによる検索結果のカスタマイズ

このセクションでは、カスタムの検索結果デコレータ関数を実装して、追加情報をアプリケーションの検索結果に追加する方法について説明します。次のトピックを扱います。

- 検査結果デコレータとは
- カスタムの検索結果デコレータの記述
- カスタムの検索結果デコレータのインストール
- カスタムの検索結果デコレータの使い方

7.10.1 検査結果デコレータとは

クエリを実行すると、MarkLogic サーバーにより、クエリの条件を満たすドキュメントまたはフラグメントごとに、<search:result> を含む <search:response/> が返されます。検索結果デコレータを使用すると、基本的な構造やデフォルトのコンテンツを変更せずに追加情報を各 <search:result/> に追加できます。

例えば MarkLogic の REST API では、内部的な結果デコレータが使用されて、href、mimetype、および format 属性が検索結果に追加されます。次の出力は、REST API のデフォルトのデコレータによって追加された情報を示しています。

```
<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>
```

XQuery Search API を使用するアプリケーションでは、カスタムの結果デコレータを使用して、同様に属性と要素を <search:result/> に追加できます。

> **注：** REST API および Java API では、検索デコレータではなく、検索結果を変換して修正する必要があります。これをビルトインの REST API 結果デコレータよりも優先させることはできますが、推奨されません。REST API または Java API でカスタムの結果デコレータを使用した場合、結果に href、mimetype、および format データを追加するデフォルトのデコレータを完全に置き換えます。また、クライアントが JSON 形式の結果をリクエストしている場合でも、デコレータが追加しデータはすべて、シリアライズされた XML として返されます。

検索結果デコレータを作成および使用するには、次の手順を実行します。

1. デコレータのインターフェイスに準拠した XQuery 関数を記述します。

2. 記述した関数を、アプリケーションサーバーに関連付けられているモジュールデータベースまたはモジュールディレクトリ内にインストールします。

3. 検索で使用する result-decorator クエリオプションでその関数を指定し、この関数が使用されるように MarkLogic サーバーにします。

このセクションの残りの部分では、これらの手順について詳しく扱います。

7.10.2　カスタムの検索結果デコレータの記述

カスタムデコレータを作成するには、次のインターフェイスに準拠した XQuery 関数を実装します。

```
declare function your-name($uri as xs:string) as node()*
```

`$uri` 入力パラメータは、マッチ部分を含むドキュメントの URI です。この関数によって返されるノードは、代わりに呼び出された `search:result` 要素の属性または子ノードになります。

JSON ドキュメントを扱っており、クライアントが JSON 形式の検索結果を要求している場合でも、結果デコレータは常に XML を生成します。

次の例は、デフォルトの REST API デコレータ（`href`、`mimetype`、および `format`）と同じ情報を返すカスタムのデコレータ関数です。ここでは、使用状況がわかるように属性名が変更されています。この例は、検索結果に `<my-elem/>` 要素も追加しています。

```
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" }
  )
};
```

関数を使用するには、これをアプリケーションサーバーにモジュールとしてインストールし、`result-decorator` クエリオプションで関数を指定します。詳細については、「カスタムの検索結果デコレータのインストール」（278 ページ）および「カスタムの検索結果デコレータの使い方」（278 ページ）を参照してください。

7.10.3 カスタムの検索結果デコレータのインストール

デコレータ関数を含む XQuery ライブラリモジュールをモジュールデータベースにインストールするか、アプリケーションサーバーに関連付けられたファイルシステムのルートの下にインストールします。

例えば、Query Console で次のクエリを実行すると、XQuery モジュールが `/my.domain/decorator.xqy` という URI でモジュールデータベースに読み込まれます。ここでは、Query Console がコンテンツソースとしてモジュールデータベースを選択していると仮定しています。

```
xquery version "1.0-ml";
xdmp:document-load(
  "/space/rest/decorator.xqy",
  <options xmlns="xdmp:document-load">
    <uri>/my.domain/decorator.xqy</uri>
  </options>)
```

REST API または Java API を使用する場合は、REST API インスタンスに関連付けられたモジュールデータベースにこのモジュールをインストールします。

7.10.4 カスタムの検索結果デコレータの使い方

カスタムの検索結果デコレータを使用するには、クエリ内の `result-decorator` クエリオプションで指定します。

例えば、「カスタムの検索結果デコレータの記述」（277 ページ）から `/my.domain/decorator.xqy` としてカスタムデコレータをインストールした場合は、クエリオプションで次のように参照できます。

```
<options xmlns="http://marklogic.com/appservices/search">
  <result-decorator apply="decorator"
      ns="http://marklogic.com/example/my-lib"
      at="/my.domain/decorator.xqy"/>
</options>
```

次の例では、検索で上記のクエリオプションを使用しています。

```
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>
)
```

デコレータでは、次のように、検索結果に my-href、my-mimetype、および my-format 属性、ならびに my-elem 要素を追加します。

```
<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>
```

7.11 その他の検索オプション

search API には、ほかにもいくつかのオプションがあります。例えば additional-query（検索におけるアクティブなクエリに and-query として組み合わされる追加の cts:query）、term-option（case-sensitive など、任意の cts:query オプションを cts:query に渡すもの）などがあります。他のオプションの動作内容の詳細については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

7.12 クエリオプションの例

このセクションは、次の追加のクエリオプションの例から構成されます。

- 例：値およびタプルクエリオプション
- 例：フィールド制約クエリオプション
- 例：コレクション制約クエリオプション
- 例：パスレンジインデックス制約クエリオプション
- 例：要素属性レンジ制約クエリオプション
- 例：位置情報制約クエリオプション

7.12.1 例：値およびタプルクエリオプション

この例では、次の設定をします。

- 値タプル /v1/values/pop。これは、（名前空間なしの）xs:QName("popularity") の要素レンジインデックスから値を取得し、xs:int として型付けします。

- 値タプル /v1/values/score。これは、名前空間「http://test.aggr.com</1685>」とローカル名「score」を使用する QName の要素レンジインデックス（xs:decimal として型付けされます）から値を取得します。
- タプル /v1/values/pop-rate-tups。これは、（名前空間なしの）xs:QName("popularity") からのレンジインデックス（xs:int として型付けされます）と、名前空間 "http://test.aggr.com" とローカル名「score」を使用する QName のレンジインデックス（xs:decimal として型付けされます）を組み合わせます。

形式	オプション
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>

形式	オプション

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>

7.12.2 例：フィールド制約クエリオプション

この例では、「bbqtext」というフィールドに基づいて 1 つの制約のみがあるシンプルな options ノードを構築しています。これは、単語制約を作成するため、

```
summary:"hot dog"
```

を検索した場合、検索は語句「hot dog」が「bbqtext」というフィールドにあるドキュメントに制約されます。

形式	オプション
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" }
	}
	}]
	} }

7.12.3 例：コレクション制約クエリオプション

この例のクエリオプションは、次の処理を実行します。

- 検索結果を修正するフラグ（return-metrics および return-qtext）を設定します。
- 生のドキュメント検索結果を返す、ビルトインの transform-results 関数を指定します。
- コレクションの URI によるコレクション制約を定義します。

「coll」というコレクション制約は、「http://test.co"」部分を除いたコレクション URI を使用します。

形式	オプション
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-elements/> </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-elements": ""
	}
	} }

7.12.4 例：パスレンジインデックス制約クエリオプション

この例は、パスレンジインデックスで制約を設定する方法を示しています。これを使って、次のものを検索します。

```
pindex:low
```

（名前空間なしの）/Employee/fn にあるノードから、5 未満の値が検索されます。これは、ファセット化され、ドキュメントがスコープとなっている文字列レンジインデックスで、わかりやすい Unicode ラベルが付けられています。検索では常にファセット値が返されます。

形式	オプション
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",
	"ge": "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"
	}
	]
	}
	}]
	} }

7.12.5 例：要素属性レンジ制約クエリオプション

この例は、要素属性レンジインデックスを示しています。バケットも計算されています。検索の際には、`{http://example.com}entry/@date` の値に応じてファセットに値が入力されます。

形式	オプション
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>

形式	オプション

JSON	<pre>{ "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"
	  }
	  ]
	}}]}}</pre>

7.12.6 例：位置情報制約クエリオプション

この例は、位置情報制約クエリオプションを示しています。検索結果の設定（page-length）に加えて、3 つの位置情報制約と 3 つの要素制約を設定しています。

形式	オプション
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>

形式	オプション

JSON	<pre>{ "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"
	          }
	        }
	      }
	    ]
	} }</pre>

8.0 関連度スコア：概要とカスタマイズ

MarkLogic サーバーでの検索結果は、関連度順で返されます。つまり、検索の `cts:query` 式に関して最も関連性が高い結果が、検索から返されたシーケンスの最初の項目になり、最も関連度が低い結果が最後の項目になります。検索結果の項目に関連付けられる関連度スコアの制御に利用できるツールが、いくつか用意されています。この章では、関連度の計算に利用できるさまざまな方法について説明します。次のセクションから構成されます。

- スコアと関連度の計算方法の概要
- フラグメント化およびインデックスオプションがスコアに与える影響
- 重みでスコアに影響を与える方法
- distance-weight オプションによる近接ブースト
- セカンダリクエリによる関連度スコアのブースト
- スコア計算にレンジクエリや位置クエリを含める
- スコアとクオリティの関係
- cts:score、cts:confidence、cts:fitness の使い方
- cts:search の関連度順と XPath のドキュメント順の比較
- 関連度スコアの計算を調べる
- cts:search 式の例

8.1 スコアと関連度の計算方法の概要

`cts:search` 操作を実行すると、MarkLogic サーバーにより、`cts:query` 式とマッチする項目と、そのスコアを含む結果セットが生成されます。スコアは、データベース内のドキュメントの数、データベース内でのこの検索タームの出現頻度、ドキュメント内でのこの検索タームの出現頻度などの統計情報に基づいて計算される数値です。返された検索項目の関連度は、結果セット内の他のスコアと比較して決定されます。スコアが高い項目ほど、この検索に対する関連度が高いと見なされます。デフォルトでは検索結果は関連度順で返されるので、このスコアが変わると、返される検索結果の順序が変わる可能性があります。

`cts:search` 式の一部として、スコア計算用に次のさまざまなメソッドを指定できます。各メソッドでは、異なる式でスコアが計算されます。

- log(tf)*idf の計算
- log(tf) の計算
- シンプルなタームマッチ計算

- ランダムなスコア計算
- ターム頻度の正規化

relevance-trace オプションと cts:relevance-info を使用することで、スコア計算の詳細を調べることができます。詳細については、「関連度スコアの計算を調べる」（311 ページ）を参照してください。

8.1.1 log(tf)*idf の計算

関連度計算の logtfidf メソッドは、デフォルトの関連度計算であり、cts:search のオプション score-logtfidf です。logtfidf メソッドは、スコアを計算するときに、ターム頻度（単一のフラグメント内でタームが出現する頻度）およびドキュメント頻度（タームが出現するドキュメントの数）を考慮します。ほとんどの検索エンジンでは、ターム頻度とドキュメント頻度を何らかの方法で計算に使用した関連度式を利用します。

logtfidf メソッド（デフォルトのスコア計算メソッド）は、次の式を使用して関連度を計算します。

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

term frequency（ターム頻度）は、ドキュメント内のターム数を表す正規化された数値です。ターム頻度はドキュメントのサイズを考慮して正規化されるので、100 語のドキュメント内で 10 回出現する単語のスコアは、1,000 語のドキュメント内で 100 回出現する単語のスコアよりも高くなります。

inverse document frequency は、次のように定義されています。

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

df（ドキュメント頻度）は、タームが出現するドキュメントの数です。

検索エンジンの関連度計算においては、一般的に score-logtfidf メソッドが最も有用な関連度スコアを提供します。逆ドキュメント頻度（IDF）は、ドキュメントの「情報の豊かさ」を測定するものです。例えば、「the」や「dog」の検索では通常、ターム「the」よりもターム「dog」の出現のほうが重要視されます。

8.1.2 log(tf) の計算

cts:search の score-logtf オプションは、logtf メソッドを使用してスコアを計算し、タームを含むドキュメントの数を考慮しません。logtf メソッドは、次の式を使用してスコアを計算します。

```
log(term frequency)
```

ここで、term frequency（ターム頻度）は、ドキュメント内のターム数を表す正規化された数値です。ターム頻度はドキュメントのサイズを考慮して正規化されるので、100 語のドキュメント内で 10 回出現する単語のスコアは、1,000 語のドキュメント内で 100 回出現する単語のスコアよりも高くなります。

logtf メソッドでは、個々のドキュメントが検索タームとマッチした回数のみに基づいてスコアが計算されます。検索タームの「情報の豊かさ」は考慮されません。

8.1.3 シンプルなタームマッチ計算

cts:search 上の score-simple オプションは、シンプルなタームマッチでスコアを計算します。score-simple メソッドは、cts:query 式内のマッチした個々のタームごとに 8* 重みというスコアを与え、そのスコアに 256 を掛けることで拡大します。ここでは個々のタームのマッチ回数（ターム頻度）は関係ありません。マッチごとに「8* 重み」がスコアに影響します。例えば、次のクエリ（デフォルトの重みが 1 であると仮定）では、「hello」が 1 回以上マッチしたフラグメントは 8×256=2048 というスコアになり、「hello」に加えて「goodbye」が 1 回以上マッチしたフラグメントは 16×256=4096 というスコアになります。どちらのタームともマッチしないフラグメントのスコアは 0 になります。

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

このオプションはクエリ内のタームとマッチしたドキュメントにのみスコアを与えるもので、タームの頻度や「情報の豊かさ」は考慮しません。

8.1.4 ランダムなスコア計算

cts:search 上の score-random オプションは、検索のマッチごとにランダムに生成されたスコアを計算します。これを使用することで、クエリとマッチするフラグメントをランダムに選ぶことができます。score-random オプションを使用して同じ検索を複数回実行した場合は、毎回順序が変わります（検索ごとに、スコアがランダムで生成されるため）。

8.1.5 ターム頻度の正規化

ターム頻度を考慮するスコア計算メソッド（score-logtfidf および score-logtf）は、デフォルトでは、ドキュメントのサイズに基づいてターム頻度（このドキュメント内での検索タームのマッチ数）を正規化します。正規化は、データベース内の他のドキュメントと比較して、ドキュメント内でのタームの出現頻度を相対的に把握するためのものです。これは、単なるタームの頻度ではなくドキュメント内のタームの密度と考えることができます。ターム頻度の正規化では、例えば、1000 万語のドキュメント内で単語「dog」が 10 回出現した場合の関連度は、100 語のドキュメント内で単語「dog」が 10 回出現した場合の関連度よりも低くなります。ターム頻度の正規化手法のデフォルトは scaled-log で、この場合ドキュメントが小さいほどスコアが高くなります（つまり検索への関連度も高くなります）。これは、単語「dog」の「ターム密度」が大きくなるためです。ほとんどの検索アプリケーションでは、この動作が望まれます。

この動作を変更する場合は、データベース設定の tf normalization オプションで、スコア計算におけるマッチドキュメントのサイズの影響を少なく（あるいはゼロに）設定できます。これにより、相対的にターム頻度（ドキュメント内でのマッチ数）の影響が強くなります。unscaled-log オプションは、ドキュメントサイズに基づいたスケーリングを実行しません。scaled-log オプション（デフォルト）は、ドキュメントサイズに基づいたドキュメントのスケーリングの中で最大レベルのものです。これら 2 つの間に、4 つのレベル（weakest-scaled-log、weakly-scaled-log、moderately-scaled-log、strongly-scaled-log）があります。これらは、スケーリング「なし」から「最大」の間で順にスケーリングの度合いが大きくなっています。データベース内でこの設定を変更し、reindexer enable を true に設定した場合、データベースの再インデックス付けが開始されます。

8.2 フラグメント化およびインデックスオプションがスコアに与える影響

スコアは、インデックスデータに基づいて計算されます（つまりフィルタリングがない検索に基づいて計算されるということです）。このことは、スコアに関していくつかの意味を持ちます。

- スコアはフラグメントごとが基準となるので、ターム頻度およびドキュメント頻度はそれぞれ、フラグメントごとのターム頻度およびフラグメント頻度として計算されます。
- スコアはフィルタリングされない検索に基づくため、結果に誤検出（false-positive）が発生します。

スコアは、フラグメントおよびフィルタリングされない検索に基づいています。このためインデックスオプションによってスコアに影響が出ます。この際、場合によってはスコアの「精度」が高くなります。つまり、誤検出（false-positive）がより少ない検索に基づいてスコアを計算します。例えば、データベース設定で word positions をオンにした場合、3 つ以上のタームを含む語句の検索ではマッチの誤検出が少なくなります。この結果、スコアの精度が高くなります。

フィルタリングがない検索について、またマッチの誤検出の有無の判断方法の詳細については、『Query Performance and Tuning Guide』の「Using Unfiltered Searches for Fast Pagination」を参照してください。

8.3 重みでスコアに影響を与える方法

クエリサブ式で重みを使用することで、関連度スコアに対するサブ式の影響を増減できます。

例えば、`cts:word-query` や `cts:element-value-query` など、リーフレベルの `cts:query` コンストラクタの重みを指定できます。詳細については、『XQuery and XSLT Reference Guide』を参照してください。重みは、構造化クエリのコンストラクト `value-query` や `word-constraint-query` など、等価の search API 抽象化で指定することも、クエリオプションで単語（ワード）制約または値制約を定義するときに指定することもできます。

デフォルトの重みは 1.0 です。カスタムの重みを選択する場合は、次のガイドラインに従ってください。

- スコアへの影響度を大きくするには、1.0 よりも大きい重みを設定します。
- スコアへの影響度を小さくするには、0 ～ 1.0 の重みを設定します。
- スコアに影響しないようにするには、重みを 0 に設定します。
- スコアへの影響度を負にするには、重みを負の数値に設定します。

スコアは正規化されるので、重みによってスコアが単純に倍になるわけではありません。そうではなく、重みはあるクエリサブ式のタームに対する、同じ式内の他のサブ式と比較した場合の相対的な重み付けを示すものです。重みが 2.0 の場合、そのクエリとマッチするタームのスコアへの影響度は 2 倍になります。同様に、重みが 0.5 の場合、そのクエリとマッチするタームのスコアへの影響度は半分になります。もしスコアが最大値に達した場合、重み 2.0 と重み 20,000 のスコアへの影響度が同じになることがあります。

重みがとりわけ役に立つのは、クエリ式内に複数のコンポーネントがあり、マッチに関してそのうちのいくつかを他のものよりも重要視したい場合です。この例については、「あるタームのスコアを上げ、他のタームのスコアを下げる」（313 ページ）を参照してください。

8.4 distance-weight オプションによる近接ブースト

データベース内で word positions というインデックス付けオプションがオンになっている場合、リーフレベルの cts:query コンストラクタで distance-weight オプションを使用できます。この場合、この cts:query コンストラクタに渡されるすべてのタームに対して、スコア計算においてタームの近接（ターム間の距離）が考慮されます。この近接ブーストでは、マッチ部分が接近しているドキュメントのほうがスコアが高くなります。検索結果はスコア順でソートされるので、検索タームが接近しているドキュメントのほうが、関連度順が高くなります。このセクションでは、distance-weight オプションを使用する例とその説明をいくつか示します。次のパートから構成されます。

- シンプルな近接ブーストの例
- cts:and-query セマンティックによる近接ブーストの使い方
- cts:near-query を使用した近接ブースト

8.4.1 シンプルな近接ブーストの例

距離の重みは、distance-weight がある cts:query コンストラクタのマッチにのみ適用されます。例えば、次の cts:query コンストラクタについて考えてみましょう。

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

あるドキュメントで「cat」のインスタンスが「dog」の非常に近くにあり、もう 1 つのドキュメントでこれと同じ個数の「cat」と「dog」があるが近くでは出現していない場合、「cat」が「dog」の近くにある前者のドキュメントの方が、スコアが高くなります。

例えば、次のコードについて考えてみましょう。

```
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)}}
```

これにより、次の結果が返されます。

```
<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."/>
```

最初の 3 つのヒットは distance-weight を使用しています。タームが接近しているヒットの方が、スコアが高くなります（検索結果の上位になります）。最後の 3 つのヒットはスコアが同じになっていますが、これは、cts:query の各タームの数がすべて同じであり、またスコアでは近接が考慮されていないためです。

8.4.2 cts:and-query セマンティックによる近接ブーストの使い方

distance-weight オプションは個々の cts:query コンストラクタにあるタームに適用されるため、タームは or-query として結合されます（つまり、いずれかのタームがマッチした場合、クエリがマッチしたことになります）。このため前の例では、「cat」を含むが「dog」を含んでいない（およびその逆の）ドキュメントの結果も返します。and-query のセマンティック（つまり、クエリがマッチするには、すべてのタームがマッチする必要がある）を使用し、近接ブーストも使用する場合は、distance-weight オプションを含む cts:query に加えて、すべてのタームの and を実行する cts:query を構築する必要があります。

例えば次のようになります。

```
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" ) )) )
```

このクエリと前のクエリの違いは、前のクエリは「cat」を含んでいても「dog」を含んでいない（あるいはその逆の）ドキュメントを返しますが、このクエリは、「cat」と「dog」の両方を含んでいるドキュメントのみを返します。

ドキュメントの大規模なコーパスが存在し、検索でマッチするものが大量にあると予想される場合は、cts:and-query のアプローチは不要かもしれません。コーパスが大規模になることによって影響が発生する理由は、「スコアと関連度の計算方法の概要」（289 ページ）で説明しているように、関連度の計算でドキュメント頻度が考慮されるためです。このため cts:and-query を使用しなくても、最も関連度が高いドキュメントが検索結果の一番上に表示されることがなります。どの方法を使えばいいのかは、アプリケーションの要件、自分の好み、およびデータに応じて変わってきます。

8.4.3 cts:near-query を使用した近接ブースト

距離が近い結果に高いスコアを与えるもう 1 つの手法として、cts:near-query があります。cts:near-query コンストラクタを使用する検索では、データベース内で word positions インデックスオプションがオンになっていれば、スコアの計算時に近接（距離の近さ）が考慮されます。さらに、distance-weight パラメータを使用することで、スコア計算における近接の効果を強調（ブースト）できます。

cts:near-query は distance 引数を取ります。ここで、どのくらい近ければマッチとみなすのかを指定します。cts:near-query の distance パラメータでは、distance の大きさ（対象範囲の広さ）とパフォーマンスの間でトレードオフが発生します。distance の数値が大きいほど、MarkLogic サーバーがクエリを解決するための作業量が増えます。多くのクエリではこの作業量はわずかでしかありませんが、複雑なクエリのなかには大きくなるものもあります。

cts:near-query を使用した近接ブーストを行うクエリを構築するには、cts:near-query の最初のパラメータとして検索用の cts:query を渡し、オプションとして distance-weight パラメータで、近接をさらにブーストします。cts:near-query のマッチではすでに距離が考慮されていますが、distance-weight を設定することで、近接の重みをさらにブーストできます。例えば、前の例と同じデータにおいて次のクエリがどのように同様の結果を生成するのかを考えてみましょう。

```
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) )
```

このクエリでは distance が 1,000 のため、「cat」と「dog」が 1,000 語よりも離れているドキュメントは結果に含まれません。この値の大きさは、扱っているデータおよび検索のパフォーマンス特性に応じて決定します。マッチ部分が 1,000 語以上離れているドキュメントを見つけられないことが心配な場合は、この値を大きくします。パフォーマンスに問題がありより速くしたい場合、また設定された距離以上に離れている結果（おそらく関連性がないと想定される）が見つからなくても問題ない場合は、この値を小さくします。大量のドキュメントが含まれるデータベースでは、遠く離れている単語を含むドキュメントを返さないようにした場合、最も関連度が高いヒットに関しては特に、検索結果は非常に似たようなものになるはずです（マッチ部分が遠く離れているものは、近くにあるものよりもスコアが低くなるため)。

8.5 セカンダリクエリによる関連度スコアのブースト

cts:boost-query を使用して、セカンダリ（「ブースト」）クエリとマッチする検索結果の関連度スコアを修正できます。次の例は、「dog」というタームを含んでいるすべてのドキュメントからの結果を返しますが、そのなかで「cat」というタームも含んでいる結果により高いスコアを割り当てます。最初のクエリでマッチしたものの関連度スコアは、2 番目のクエリがマッチしたことでブーストされます。

```
cts:search(fn:doc(),
  cts:boost-query(
    cts:word-query("dog"),
    cts:word-query("cat"))
)
```

「スコアと関連度の計算方法の概要」（289 ページ）で説明したように、多くの要因が関連度スコアに影響するので、関連度スコアに対するブーストクエリの定量的な効果はさまざまです。ただし、この効果は常にブーストクエリの重みに比例します。

例えば、次のコンテンツを含む /example/dogs.xml と /example/llamas.xml という 2 つのドキュメントがデータベース内にあるとします。

```
/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>
```

「dog」という単語のブーストなしの検索では、次のマッチが返されます。

```
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>
```

これらのマッチでは関連度スコアが同じと仮定します。この検索をブーストクエリとして（重みはデフォルト値）再度実行した場合、最初のマッチのスコアは 2 番目のもののおよそ 2 倍になります（この場合、実際のスコア値がどのようなものであれ、そのような関係になるということです）。

```
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
```

ブーストクエリの重みを 10.0 に増やした場合、両方のタームを含んでいるドキュメントの関連度スコアは、「dog」のみを含んでいるドキュメントのおよそ 10 倍になります。

```
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
```

プライマリ（「マッチング」）クエリが何も結果を返さなかった場合、ブーストクエリは評価されません。ブーストクエリは、XPath 式の場合、またはスコアが 0 あるいはランダム化されている場合には無視されます。

文字列クエリ演算子 BOOST を使用すると、文字列クエリで同様のブーストができます。詳細については、「クエリのコンポーネントと演算子」（61 ページ）を参照してください。構造化クエリコンポーネント boost-query にも、cts:boost-query と同じ機能があります。詳細については、「boost-query」（102 ページ）を参照してください。

8.6 スコア計算にレンジクエリや位置クエリを含める

デフォルトでは、レンジクエリは関連度スコアに影響しません。ただし、score-function オプションおよび slope-factor オプションを使用して、レンジクエリおよび位置クエリがスコアに影響するようにできます。このセクションでは、次のトピックについて扱います。

- レンジクエリのスコアへの影響
- レンジクエリのスコアへの影響のユースケース

- レンジクエリのスコアへの影響をオンにする
- 傾き係数とは
- パフォーマンスに関する考慮事項
- レンジクエリのスコア計算の例

8.6.1 レンジクエリのスコアへの影響

デフォルトでは、レンジクエリはスコアに影響しません。特定のレンジクエリに対してスコア計算をオンにした場合、そのクエリはワードクエリと同じ影響を与えるようになります。レンジクエリは、スコア全体に影響を与える多くの要因の 1 つにすぎません。複雑なクエリの場合は特にそうです。レンジクエリでは、他のクエリの場合と同様に重みを使用してスコアに与える影響を変更できます。詳細については、「重みでスコアに影響を与える方法」（293 ページ）を参照してください。

マッチしたものの値と参照値の差は、スコアには直接影響しません。関数が差分（delta）に適用され、データ型に応じてスケーリングされます。具体的には、結果として得られるレンジがワードクエリにおけるターム頻度（TF）の影響度と同等になるようにスケーリングされます。スケーリングの度合いは、関数の傾き係数で設定します。詳細については、「傾き係数とは」（301 ページ）を参照してください。

関数のタイプ（線形または反比例）により、参照値に最も近いあるいは最も遠い値のどちらがより大きくスコアに影響するかが決まります。参照値とは、クエリ内の制約値のことです。例えば、レンジクエリで「> 5」のような制約を表現した場合、参照値は 5 になります。関数自体は選択できませんが、関数のタイプは選択できます。

ドキュメント内にマッチする値が複数含まれている場合、全体のスコア計算では、最も大きい影響度（寄与度）が使用されます。

8.6.2 レンジクエリのスコアへの影響のユースケース

レンジクエリによるスコアへの影響は、次のような場合に役立ちます。

- 新しいドキュメントのスコアを類似する古いドキュメントよりもブーストする場合。この場合、「新しさ」は dateTime 型または他の数値型の要素値の関数です。例えば、最近発行されたドキュメントのスコアをブーストします。
- ある要素値と参照値の近さに基づいてスコアをブーストする場合。例えば、想定価格である 20 ドルに最も近い価格を含むドキュメントのスコアをブーストします。
- ある要素値と参照値の遠さに基づいてスコアをブーストする場合。例えば、最大価格である 20 ドルを最も大きく下回っている品目のスコアをブーストします。
- 位置的な距離を基にしてスコアをブーストします。例えば 5 マイル以内にあるホテルを探す際に、現在地に最も近いホテルのスコアをブーストします。

これらのユースケースを実現する方法の例については、「レンジクエリのスコア計算の例」（304 ページ）を参照してください。

8.6.3 レンジクエリのスコアへの影響をオンにする

レンジクエリまたは位置クエリのコンストラクタに score-function オプションを追加して、スコアへの影響をオンにします。slope-factor オプションで、影響度をスケーリングすることもできます。詳細については、「傾き係数とは」（301 ページ）を参照してください。

例えば、次の検索は評価の高い（参照値 0 から最も遠い）ドキュメントのスコアをブーストします。傾き係数を 10 に設定すると、個別に影響する値のレンジが狭くなり、影響度の違いが大きくなります。

```
(: 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")))
```

同様のクエリを他の MarkLogic サーバー API で構築する例については、「レンジクエリのスコア計算の例」（304 ページ）を参照してください。

score-function の値は次のいずれかの関数タイプに設定できます。

スコア関数	説明
zero	デフォルトです。レンジクエリのスコアへの影響度がゼロになります。
reciprocal	反比例の関数を使用して、スコアへの影響を計算します。参照値に近いドキュメントの値が、より高いスコアを獲得します。
linear	線形の関数を使用して、スコアへの影響を計算します。参照値から遠いドキュメントの値が、より高いスコアを獲得します。

スコア関数および傾き係数は、次の XQuery クエリコンストラクタ、あるいは等価の構造化レンジクエリコンストラクトまたは QBE レンジクエリコンストラクトで指定できます。

- 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`

8.6.4 傾き係数とは

レンジクエリへのスコア関数の指定に加えて、`slope-factor` オプションを使用して、レンジクエリに適用するスコア計算関数の傾き係数を指定できます。傾き係数によって、マッチング値と参照値の差によるスコアへの影響度が変わってきます。特定のレンジクエリに最適な傾き係数を決めるには、実際のアプリケーションで試してみる必要があります。このセクションでは、この試し方に関する詳しいガイドを提供します。

レンジクエリにおけるマッチの「差分」（delta）とは、マッチした値とレンジクエリ内の参照値の差のことです。

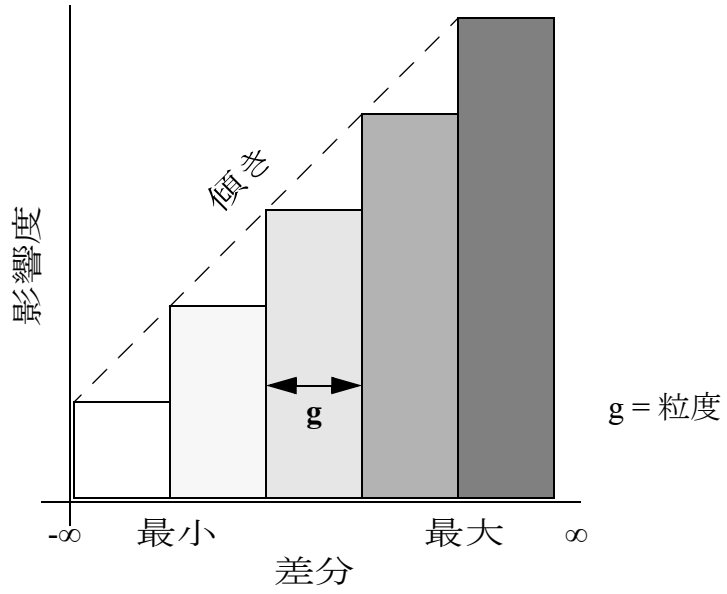
```
delta = reference_value - matching_value
```

例えば、レンジクエリが「5 よりも大きい」となっており、実際にマッチした値が 3 の場合、差分は 2 です。この差分自体が、直接スコアに影響するわけではありません。これは個々のマッチにおけるスコアへの影響度の基になります。

影響度はバケット化される（ある範囲のものをまとめる）ため、個々の差分値ごとにスコアへの影響度が異なるわけではありません。差分値は、最小差分値と最大差分値による範囲にまとめられ、この範囲外の差分の影響度はすべて同じになります。粒度（granularity。細かさのレベル）とは、その範囲内の各バケットのサイズを表します。同じバケット内のすべての差分はスコアへの影響度が同じです。粒度によって、スコアへの影響度が同一となる差分の範囲が決まります。

傾き係数を変えてもバケットの数は変わりません。傾き係数の変更によって影響を受けるのは、スコア関数の最小値、最大値、および粒度です。

下の図に、線形のスコア関数の傾き、最小差分、最大差分、粒度の関係を示します。



傾き係数が 1 よりも大きい場合、粒度がより細かくなりますが、差分値の範囲は狭くなります。傾き係数が 1 よりも小さい場合、粒度がより荒くなりますが、差分値の範囲は広くなります。線形関数で傾き係数を 2 倍にすると、範囲と粒度が半分になります。

特定の傾き係数における最小差分、最大差分、粒度は、レンジインデックスのタイプによって異なります。下の表に、デフォルトの傾き係数（1.0）における、各レンジインデックスタイプの最小差分、最大差分、および粒度を示します。反比例のスコア関数の場合、粒度は直線的に変化しません。

レンジインデックスタイプ	下限	上限	粒度
`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 マイル ～ 0.87 度	100.0 マイル ～ 1.45 度	～ 0.39 マイル ～ 0.34 分

レンジインデックスタイプ	下限	上限	粒度
point (raw)	1.0	100.0	～ 0.39
date	1 日	1 年	～ 1.5 日
time	1 分	24 時間	～ 5.5 分
dateTime	1 分	30 日	～ 2.6 時間
dayTimeDuration	1 分	24 時間	～ 5.5 分
yearMonthDuration	1ヵ月	25 年	1ヵ月
gYear	1 年	100 年	1 年
gMonth	1ヵ月	1 年	1ヵ月
gDay	1 日	1ヵ月	1 日
gYearMonth	1ヵ月	25 年	1ヵ月

例えば、この表は、dateTime を対象とした、デフォルトの傾き係数を持つレンジクエリに関する次の情報を含んでいます。

```
Min delta: 1 minute
Max delta: 30 days
Granularity: ~2.6 hours
```

ここから、傾き係数 1.0 の場合は次のことを推論できます。

- 1 分未満の差分の影響度はすべて 1 分と同じになる
- 30 日よりも大きい差分の影響度はすべて 30 日と同じになる
- 2.6 時間以内の差分の影響度は、すべて同じになる例えば、5 分という差分と 2 時間という差分はいずれも「1 分 < 差分 ≤ 2.6 時間」のバケットに含まれるため、影響度は同じです。

差分が時間レベルである dateTime レンジクエリでは、デフォルトの傾き係数により、影響度がきれいに分散されます。一方、数分または数秒単位の差分を区別する必要がある場合は、傾き係数を大きくして粒度をより細かくします。この場合、最小差分値と最大差分値が近くなるので、区別可能な差分値の範囲が小さくなります。

傾き係数のもう 1 つの考え方としては、ターゲットとなる最小差分または最大差分に基づくものがあります。例えば、使用しているデータのタイプにおいてデフォルトの最大差分が 1024 だけれども、レンジクエリにとって「関心がある」差分値は 1 〜 100 のみの場合、傾き係数を 10 に設定することで、最大差分を 100（`1024 div 10`）に減らすことができます。

8.6.5 パフォーマンスに関する考慮事項

レンジクエリがスコアに寄与している場合、それがパフォーマンスに与える影響はクエリによって異なります。コストが最も高いのは、多くのマッチを返すクエリと、文字列に対するクエリです。

マッチの数がコストに影響するのは、マッチごとにスコアが計算されるためです。値の型がコストに影響するのは、文字列値ではスコア計算が大幅に複雑になるためです。

次のいずれかの条件に該当する場合、レンジクエリのスコアへの影響度の計算はスキップされます（そのためパフォーマンスへ悪影響はありません）。

- `score-function` オプションが設定されていないか、`zero` に設定されている。
- レンジクエリの重みが 0 になっている。
- スコア計算メソッドがターム頻度を使用しない。つまり、スコア計算メソッドは `score-logtfidf` でも `score-logtf` でもない。

8.6.6 レンジクエリのスコア計算の例

このセクションには、「レンジクエリのスコアへの影響のユースケース」（299 ページ）に記載されているユースケースを紹介する例、その他の API（構造化クエリや QBE など）によるこの機能の使用方法の例が含まれています。

次の例が含まれます。

- 例：一番最近公開されたもの
- 例：ターゲットの価格に最も近い
- 例：最高価格以下の最安値
- 例：ある場所に最も近い
- 例：構造化クエリでの使い方
- 例：Query By Example での使い方

8.6.6.1 例：一番最近公開されたもの

新しいドキュメントのスコアを類似した古いドキュメントよりもブーストします。この場合、「新しさ」は dateTime 型または他の数値型の要素値の関数です。次の例は、最近公開されたドキュメントのスコアをブーストします。公開の日付は pubdate 要素に格納されています。

```
cts:element-range-query(
  xs:QName("pubdate"), "<=", current-dateTime(),
  "score-function=reciprocal")
```

この例では、反比例のスコア関数を使用するので、「現在」に最も近い pubdate 値がスコアに最も大きく影響します。つまり、差分が最も小さいものが影響度は最も大きくなります。

8.6.6.2 例：ターゲットの価格に最も近い

ある要素値と参照値の近さに基づいてスコアをブーストする場合。次の例では、20 ドルという想定価格に最も近い価格を含むドキュメントのスコアをブーストします。ここでは、price は item 要素の属性だと仮定します。

```
cts:element-attribute-range-query(
  xs:QName("item"), xs:QName("price"), ">=", 20.0,
  "score-function=reciprocal")
```

この例では、反比例のスコア関数を使用するので、実際の価格と想定価格（20 ドル）の差分が最も小さいものが、影響度が最大になります。

8.6.6.3 例：最高価格以下の最安値

ある要素値と参照値の遠さに基づいてスコアをブーストする場合。例えば、最高価格 20 ドルを最も大きく下回っている項目のスコアをブーストします。

```
cts:element-attribute-range-query(
  xs:QName("item"), xs:QName("price"), "<=", xs:decimal(20.0),
  ("score-function=linear","slope-factor=51.2"))
```

この例では線形関数を使用するので、実際の価格と最高価格（20 ドル）の差分が最も大きいものが、影響度が最大になります。

傾き係数を大きくすることで、関心がある差分値の範囲を小さくします。「傾き係数とは」（301 ページ）に示したように、xs:decimal のデフォルトの最大差分は 1024.0 です。ただしこの例では、関心のある差分はすべて 0 〜 20.0 の範囲内にあります。上限を 20.0 に下げるために、次のように傾き係数を計算します。

```
slope-factor = 1024.0 / 20.0 = 51.2
```

傾き係数を大きくすると粒度が細かくなるので、小さい価格差でもスコアの影響度が異なるようになります。デフォルトの傾き係数では、粒度は～ 3.98 になります。これは、差分レンジ 0 ～ 20.0 に対しては非常に荒くなっています。

8.6.6.4 例：ある場所に最も近い

位置的な距離を基にしてスコアをブーストします。例えば 10 マイル以内にあるホテルを探す際に、現在地に最も近いホテルのスコアをブーストします。

```
cts:and-query(("hotel",
  cts:element-geospatial-query(
    xs:QName("pt"), cts:circle(10, $current-location),
    ("score-function=reciprocal", "slope-factor=10.0"))))
```

この例では、反比例のスコア関数を使用するので、基準地に最も近い（差分が最も小さい）地点のスコアへの影響度が最も大きくなります。

傾き係数を大きくした理由は、関心のある差分値の範囲は 0 ～ 10（「10 マイル以内」）のみだからです。「傾き係数とは」（301 ページ）に示したように、地点（point）のデフォルトの最大差分は 100.0 マイルです。最大差分を 10.0 に減らすには、傾き係数を次のように計算します。

```
slope-factor = 100.0 / 10.0 = 10.0
```

8.6.6.5 例：構造化クエリでの使い方

次の例は、0 よりも大きい評価に関するレンジクエリを含む構造化クエリです。評価が高いほどスコアを大きくブーストしています。評価が高いドキュメントは、レンジクエリのスコアへの影響度がより大きくなります。

形式	クエリ
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>

形式	クエリ

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

詳細については、「構造化クエリを使用した検索」（74 ページ）および次のインターフェイスを参照してください。

インターフェイス	インターフェイス	詳細情報
Search API	`search:resolve`	『XQuery and XSLT Reference Guide』
REST API	`/search` サービスの GET/POST メソッド	『REST Application Developer's Guide』の「Querying Documents and Metadata」
Java API	`com.marklogic.client.query` 内の `RawStructuredQueryDefinition` または `StructuredQueryBuilder`	『Java Application Developer's Guide』の「Search Documents Using Structured Query Definition」

8.6.6.6 例：Query By Example での使い方

次の例は、0 よりも大きい評価に関するレンジクエリを含む QBE です。評価が高くなるほどスコアを大きくブーストしています。評価が高いドキュメントは、レンジクエリのスコアへの影響度がより大きくなります。

このクエリは、REST API `/qbe` サービスまたは Java API `RawQueryByExampleDefinition` インターフェイスでの使用に適しています。

形式	例
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
	  }
	} }

詳細については、「Query By Example を使用した検索」（181 ページ）および次のインターフェイスを参照してください。

インターフェイス	インターフェイス	詳細情報
Search API	`search:resolve`	『XQuery and XSLT Reference Guide』
REST API	`/qbe` サービスの GET/POST メソッド	『REST Application Developer's Guide』の「Using Query By Example to Prototype a Query」
Java API	`com.marklogic.client.query` 内の `RawQueryByExampleDefinition`	『Java Application Developer's Guide』の「Prototype a Query Using Query By Example」

8.7 スコアとクオリティの関係

各ドキュメントにはクオリティの値が含まれています。この値は読み込み時に設定されるか、xdmp:document-set-quality で設定されます。cts:search でオプションの $QualityWeight パラメータを使用して、ドキュメントクオリティが強制的にスコアに影響するようにできます。この場合、スコアは次の式によって決定されます。

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

QualityWeight のデフォルトは 1.0 であり、ドキュメントにおけるクオリティのデフォルトは 0 です。このためデフォルトにおいて、クオリティが設定されていないドキュメントではクオリティによるスコアへの影響はありません。一方、クオリティが設定されているドキュメントは、デフォルトでもスコアが影響を受けます（デフォルトの QualityWeight は 1 であるため、ドキュメントクオリティの分だけスコアがブーストされます)。

クオリティがスコアに与える影響を小さくしたい場合、QualityWeight を 0 と 1.0 の間に設定します。クオリティがスコアに影響を与えないようにしたい場合は、QualityWeight を 0 に設定します。クオリティがスコア向上に与える影響を大きくしたい場合は、QualityWeight を 1.0 よりも大きく設定します。クオリティがスコアにマイナスの影響を与えるようにしたい場合は、QualityWeight を負の数値に設定するか、ドキュメントクオリティを負の数値に設定します。

注： ドキュメントクオリティを負の数値に設定し、QualityWeight を負の数値に設定した場合、正の数値でスコアがブーストされます。

8.8 cts:score、cts:confidence、cts:fitness の使い方

結果ノードで cts:score を呼び出すと、そのノードのスコアを取得できます。スコアは数値であり、数値が大きいほど特定の結果セットの関連度が高いことになります。

同様に、結果ノードで cts:confidence を呼び出すと、信頼度を取得できます。信頼度は、0.0 〜 1.0 の（xs:float 型の）数値です。信頼度の数値は、ドキュメントにおけるクオリティの設定を含みません。信頼度スコアは、スコアを 0 〜 1.0 に制限してから、この数値の平方根を取ることによって計算されます。

cts:confidence の代わりに、結果ノードで cts:fitness を呼び出すことにより、適合度を取得できます。適合度は、0.0 〜 1.0 の（xs:float 型の）数値です。適合度の数値は、ドキュメントにおけるクオリティの設定を含みません。また計算にドキュメント頻度も使用しません。このため、cts:fitness は、返されたノードが発行されたクエリをどの程度満たしているかを数値化したものです。この数値は、データベース内の他のドキュメントを考慮しないため、関連度とはわずかながら違います。

8.9 cts:search の関連度順と XPath のドキュメント順の比較

式が返す順序を理解するには、次の 2 つの主要なルールを考慮する必要があります。

- cts:search 式は常に、関連度順（最も関連度が高いものから低いものへの順序）で返します。
- XPath 式は常に、ドキュメント順で返します。

これらのルールに関する注意点として、cts:search 式の後に Xpath のステップがいくつかある場合、この式自体が XPath 式になってしまい、結果がドキュメント順で返されることがあります。例えば、次のクエリについて考えてみましょう。

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

これは、指定された語句を含むドキュメントノードを関連度順で返します。cts:score を使用すると、各ノードのスコアを取得できます。この式に次のような Xpath のステップを追加した場合、この状況が変わります。

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

今度は、TITLE 要素のドキュメント順で返されます。さらにこのクエリへの回答を計算するために、MarkLogic サーバーは、まず検索を実行した後に、検索をドキュメント順に並べ替えて XPath 式を解決する必要があります。このタイプのクエリを実行する場合は通常、次のように XPath の述語で cts:contains を使用すると、効率が向上します（大幅に向上することもよくあります）。

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

注： ほとんどの場合、この形式（すべてが XPath 式）のクエリは、前述の形式（cts:search 式の後に XPath ステップがある形式）よりも効率が向上します。ただし、特にクエリの選択条件が厳しい（フラグメントとのマッチが多くない）場合など、場合によっては効率が低下することもあります。

クエリを XPath 式として記述すると、MarkLogic サーバーはスコアを計算しないので、スコアが必要な場合は、cts:search 式を使用する必要があります。さらに、前の例のようなクエリを使う必要があり、また関連度順の結果が必要な場合は、次のように検索を FLWOR 式内に入れることができます。

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

これにより、XPath ステップが後に続く cts:search よりも効率が向上し、スコアが計算された結果が関連度順で返されます。

8.10 関連度スコアの計算を調べる

relevance-trace 検索オプションを使用して、特定のクエリの関連度スコアの計算方法を調べることができます。例えば、この機能を使用して、さまざまなクエリの重みやドキュメントクオリティの重みの影響を調べることができます。

> **注：** 検索中にスコアの計算情報を収集するとコストがかかるので、relevance-trace オプションは、収集されたトレースからスコア計算レポートを生成する場合にのみ使用する必要があります。

検索で relevance-trace オプションを使用すると、MarkLogic サーバーにより、関連度スコアの計算方法に関する詳細情報が収集されます。この情報には、次のいずれかの方法でアクセスできます。

- cts:search を使用して検索した場合は、検査結果で cts:relevance-info を呼び出して、XML レポートを生成します。
- search API（search:search または search:resolve）、REST API、Java API を使用して検索した場合は、各検索結果の relevance-info セクションで XML レポートが自動的に返されます（REST API および Java API は、JSON レポートを返すこともできます）。

次の例では、cts:search の結果からスコア計算レポートを生成しています。

```
for $x in cts:search(fn:doc(), "example", "relevance-trace")
return cts:relevance-info($x)
```

結果のスコア計算レポートは、次のようになります。

```
<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>
```

各 qry:score 要素に含まれている @formula は計算式を示し、@computation は式に挿入される値を示します。score 要素内のデータが計算の結果です。例えば次のようになります。

```
<qry:score
    formula="(256*scoreSum/weightSum)+(256*qualityWeight*documentQuality)"
    computation="(256*154/2)+(256*1*0)">
  19712
</qry:score>
```

次の例では、XQuery Search API を使用してスコア計算レポートを生成しています。

```
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>
)
```

このクエリは、次のような結果を生成します。

```
<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>
```

REST および Java API は、上記の search API の例と同じクエリオプションを使用し、同じ方法、つまり、各 search:result の内部でレポートを返します。

8.11 cts:search 式の例

このセクションでは、重みやクオリティのパラメータを含むいくつかの cts:search 式のリストを示します。次の例が含まれます。

- ドキュメントのスコアブーストをクオリティで拡大する
- あるタームのスコアを上げ、他のタームのスコアを下げる

8.11.1 ドキュメントのスコアブーストをクオリティで拡大する

次の検索では、クオリティが設定されているドキュメント（クオリティは読み込み時に設定されるか、または xdmp:document-set-quality で設定されます）に、クオリティが設定されていないドキュメントよりもずっと高いスコアを与えます。

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

注： クオリティが -1.0 未満の負の数値に設定されているドキュメントの場合、この検索は、それらのドキュメントにマッチがあった場合にスコアを大きく下げる効果があります。

8.11.2 あるタームのスコアを上げ、他のタームのスコアを下げる

次の検索は、あるクエリを満たすドキュメントのスコアをブーストし、他のクエリを満たすドキュメントのスコアを下げます。

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

この検索は、alfa という単語を含んでいるドキュメントのスコアをブーストし、lada という単語を含んでいるドキュメントのスコアを低くします。両方のタームを含んでいるドキュメントの場合、alfa という単語によりスコアがブーストされる一方、lada という単語によりスコアが低くされます。

9.0 レキシコンを利用した閲覧

MarkLogic サーバーでは、レキシコンを作成できます。レキシコンは、一意の単語または値のリストです。データベース全体（単語のみ）あるいは名前付き要素または属性（単語や値）が対象です。また、データベース内のドキュメントおよびコレクションの URI にすばやくアクセスできるレキシコンを定義したり、名前付きフィールドに関するワードレキシコンを作成したりできます。この章では、MarkLogic サーバーで作成できるレキシコン、および API を使用してレキシコンを閲覧する方法について説明します。この章は、次のセクションから構成されます。

- レキシコンについて
- レキシコンの作成
- ワードレキシコン
- 要素 / 要素属性 / パス値レキシコン
- フィールド値レキシコン
- 値共起レキシコン
- 位置情報レキシコン
- レンジレキシコン
- URI レキシコンとコレクションレキシコン
- レキシコンベースのクエリの実行

9.1 レキシコンについて

*単語（ワード）レキシコン*は、データベース内、あるいは QName によって定義された要素や属性の、大文字 / 小文字の区別あり、発音符号による区別ありの一意の単語をすべて格納します。*値レキシコン*は、QName によって定義された要素や属性（つまり、指定された要素や属性のコンテンツそのまま全体）の一意の値をすべて格納します。*値共起レキシコン*は、同一フラグメント内に出現する値のペアをすべて格納します。*位置情報レキシコン*は、位置情報インデックスからの位置情報値を返します。*レンジレキシコン*は、指定された値の範囲内で出現する値のバケットを格納します。*URI レキシコン*は、データベース内のドキュメントの URI を格納し、*コレクションレキシコン*は、データベース内のすべてのコレクションの URI を格納します。

すべてのレキシコンでは、指定されたコレーションに基づいてその順序と一意性が決定されます（`xs:string` 型の場合）。つまり、同じオブジェクトに対してコレーションが異なるレキシコンを複数作成できます。コレーションについては、「コレーション」（493 ページ）を参照してください。文字列以外の値に対する値レキシコンを作成することもできます。

これらのすべてのタイプのレキシコンには、次の特性があります。

- レキシコンのタームおよび値は、大文字 / 小文字が区別されます（case-sensitive）。
- レキシコンのタームおよび値は、ステミングされていません（unstemmed）。
- レキシコンのタームおよび値は、発音符号によって区別されます（diacritic-sensitive）。
- レキシコンのタームおよび値には、関連度（重要度）の情報がありません。
- レキシコンの一意性は、レキシコンに対して指定されたコレーションに基づいています。
- ワードレキシコン内のタームは、句読点を含みません。例えば、データベース内の `case-sensitive` というタームは、レキシコン内では `case` と `sensitive` という 2 つのタームになります。
- 値レキシコン内の値には、句読点が含まれます。
- レキシコンに基づくクエリを実行するには、適切なレキシコンを作成する必要があります。レキシコンが作成されていない場合、レキシコンクエリは例外をスローします。
- レキシコンは、search API による制約とともに使用されます。レンジインデックスに基づいたレキシコンは、値制約に使用されます（値制約はファセットに使用されます）。search API、制約、およびファセットの詳細については、「search API：概要と使い方」（22 ページ）を参照してください。

レキシコンでは大文字 / 小文字の区別あり、ステミングなし、発音符号による区別ありでタームを格納しています。しかし適切なオプションを指定することで、大文字 / 小文字の区別なし、発音符号による区別なしのレキシコンベースのクエリも実行できます。シンタックスの詳細については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

9.2 レキシコンの作成

レキシコンベースのクエリを実行するには、適切なレキシコンを作成する必要があります。レキシコンは Admin 画面で作成します。レキシコンの作成の詳細については、『Administrator's Guide』の「Text Indexing」および「Element/Attribute Range Indexes and Lexicons」の章を参照してください。レキシコンベースのクエリを正常に実行するには、どのレキシコンにおいても、次のうちの少なくとも 1 つを完了する必要があります。

- データをデータベース内に読み込む前に、レキシコンを作成しオンにします。
- レキシコンを作成しオンにした後、データベースを再インデックス付けします。
- レキシコンを作成しオンにした後、データを再読み込みします。

次に、各種レキシコンの作成方法を簡単にまとめます。

- データベース全体のワードレキシコンを作成するには、Admin 画面の［Database Configuration］ページ（［Databases］>［*db_name*］）で `word lexicon` をオンにし、レキシコンのコレーション（例えば、UCA ルートコレーションの場合は `http://marklogic.com/collation/`）を指定します。
- 要素ワードレキシコンを作成するには、Admin 画面の［Element Word Lexicon Configuration］ページ（［Databases］>［*db_name*］>［Element Word Lexicons］）で要素の名前空間 URI、ローカル名、およびコレーションを指定します。
- 要素属性ワードレキシコンを作成するには、Admin 画面の［Element Attribute Word Lexicon Configuration］ページ（［Databases］>［*db_name*］>［Attribute Word Lexicons］）で要素および属性の名前空間 URI、ローカル名、およびコレーションを指定します。
- 要素値レキシコンを作成するには、Admin 画面の［Range Element Index Configuration］ページ（［Databases］>［*db_name*］>［Element Indexes］）で、要素の名前空間 URI、ローカル名、（`xs:string` の）コレーション、および型（例えば、`xs:string`）を指定します。
- 要素属性値レキシコンを作成するには、Admin 画面の［Range Element-Attribute Index Configuration］ページ（［Databases］>［*db_name*］>［Attribute Indexes］）で要素および属性の名前空間 URI、ローカル名、（`xs:string` の）コレーション、および型（例えば、`xs:string`）を指定します。
- フィールド値レキシコンを作成するには、まず、Admin 画面でフィールドを作成します（［Databases］>［*db_name*］>［Fields］）。次に、Admin 画面の［Field Range Index Configuration］ページ（［Databases］>［*db_name*］>［Field Range Indexes］）で型（例えば、`xs:string`）およびフィールド名を指定してフィールド値レキシコンを作成します。

  注： システムが再インデックス付け / 再フラグメント化するように設定されている場合、新しく作成されたレキシコンは、再インデックス付けが完了するまで利用できません

9.3 ワードレキシコン

次のように、ワードレキシコンには複数の種類があります。

- データベース全体のワードレキシコン
- 要素 / 要素属性ワードレキシコン
- フィールドワードレキシコン

9.3.1 データベース全体のワードレキシコン

ワードレキシコンはデータベース全体を対象とし、データベース内のすべての一意のタームを格納します。何をもって一意とするのかは、指定されたコレーションによって決まります。ワードレキシコンをオンにするには、Admin 画面のデータベースページで、`word lexicon` データベース設定をオンにします。データベースにコンテンツがすでに読み込まれている場合、レキシコンクエリを実行するには、データベースを再インデックス付けする必要があります。次に、ワードレキシコンの API を示します。

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

9.3.2 要素 / 要素属性ワードレキシコン

要素ワードレキシコンおよび要素属性ワードレキシコンは、指定された要素や属性内の一意のタームをすべて含んでいます。何をもって一意とするのかは、指定されたコレーションによって決まります。要素ワードレキシコンは、指定された要素の直接の子テキストノード、および Admin 画面で element-word-query-throughs または phrase-throughs として定義された要素のあらゆる子テキストノード内に存在する単語のみを含みます。指定された要素のその他の子からの単語は含みません。要素ワードレキシコンおよび要素属性ワードレキシコンは、Admin 画面の、レキシコンの作成対象であるデータベースの下にある `Element Range Indexes` および `Attribute Range Indexes` リンクで作成します。次に、要素ワードレキシコンおよび要素属性ワードレキシコンの API を示します。

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

9.3.3 フィールドワードレキシコン

フィールドは、データベースレベルで作成される名前付きオブジェクトです。ここでフィールドからまとめてアクセスできる要素の集まりを定義します。フィールドに関してはワードレキシコンを作成できます。このレキシコンは、フィールドに含まれる一意の値をすべてリスト化します。フィールドワードレキシコンは各フィールドの設定ページで作成できます。他のすべてのレキシコンと同様、フィールドワードレキシコンはコレーションに基づいています。このため必要に応じて、コレーションが異なる複数のレキシコンを作成可能です。フィールドの詳細については、『Administrator's Guide』の「Fields Database Settings」を参照してください。次に、フィールドワードレキシコンの API を示します。

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

9.4 要素 / 要素属性 / パス値レキシコン

要素値レキシコン、要素属性値レキシコン、またはパス値レキシコンは、指定された要素または属性内の一意の値をすべて含んでいます。値は、指定された要素または属性のコンテンツそのものの全体です。要素値レキシコンおよび要素属性値レキシコンを作成するには、Admin 画面で、値レキシコンの対象となる要素または属性に対して特定の型（例えば、xs:string）のレンジインデックスを作成します。次に、要素値レキシコン、要素属性値レキシコン、およびパス値レキシコンの API を示します。

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

cts:element-values および cts:element-value-match 関数は、要素レンジインデックスに基づく要素値レキシコンから値を返すために使用されます。cts:element-attribute-values および cts:element-attribute-value-match 関数は、属性レンジインデックスに基づく属性値レキシコンから値を返すために使用されます。cts:values および cts:value-match 関数は、パスレンジインデックスに基づくパス値レキシコンから値を返すために使用されます。パス値レキシコンは要素または属性のいずれかです。

> **注：** シンプルな要素（子要素がないもの）に対しては、要素値レキシコンしか作成できません。

要素または属性に関する値レキシコンがある場合は、API cts:frequency も使用して、値が出現する回数を高速かつ正確に取得できます。値のインスタンスが複数回出現するフラグメントの数を取得する（値レキシコン API のデフォルトの fragment-frequency オプションを使用）ことも、各項目内の値の合計件数を取得する（item-frequency オプションを使用）こともできます。詳細および例については、『MarkLogic XQuery and XSLT Function Reference』の cts:frequency や値レキシコン API のドキュメントを参照してください。

9.5 フィールド値レキシコン

フィールド値レキシコンは、指定されたフィールドの一意の値をすべて含んでいます。フィールド値レキシコンを作成するには、Admin 画面で、フィールドレキシコンの対象となるフィールドに対して特定の型（例えば、xs:string）のレンジインデックスを作成します。次に、フィールド値レキシコンの API を示します。

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

フィールドに関する値レキシコンがある場合は、API `cts:frequency` も使用して、値が出現する回数を高速かつ正確に取得できます。値のインスタンスが複数回出現するフラグメントの数を取得する（値レキシコン API のデフォルトの `fragment-frequency` オプションを使用）ことも、各項目内の値の合計件数を取得する（`item-frequency` オプションを使用）こともできます。詳細および例については、『MarkLogic XQuery and XSLT Function Reference』の `cts:frequency` や値レキシコン API のドキュメントを参照してください。

フィールド値レキシコンは、別個の値として扱いたいものが単一の要素や属性として出現しない場合に役立ちます。例えば、次の XML 構造について考えてみましょう。

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

名前を `firstname lastname` という形式で正規化したい場合は、この構造のフィールドを作成できます。このフィールドには要素 `name` が含まれ、要素 `middle` は除外されます。このフィールドにおいて、このインスタンスの値は `Raymond Carver` になります。ドキュメントに構造が同じ name 要素が他にあった場合、それらの値も同様に生成されます。フィールドのレンジインデックスは、フィールド値の一意の各インスタンスを格納します。

フィールドの詳細については、『Administrator's Guide』の「Fields Database Settings」を参照してください。

9.6 値共起レキシコン

値共起レキシコンは、同一フラグメント内で出現する要素または属性値のペアを探します。レンジインデックスの位置がオンの場合は、共起ペアとしてマッチの対象となるための値間の最大単語距離（`proximity=`*N* オプション）も指定できます。次の API が、このレキシコンをサポートしています。

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

これらの API は、共起値のペアを含んでいる XML 構造を返します。これらの関数の出力に `cts:frequency` を使用して、各共起の頻度（件数）を求めることができます。

また、「位置情報レキシコン」（321 ページ）の説明に従って位置情報レキシコンから共起を取得することもできます。

注： URI レキシコンとコレクションレキシコンはレンジインデックスとして実装されるため、ドキュメント URI レキシコンまたはコレクション URI レキシコンに対して特別な QName を指定して、それらの URI またはコレクションの値のリストを取得できます。QName は、名前空間 `http://marklogic.com/xdmp` 内にあります。URI インデックスはローカル名 `document` を持ち、コレクションインデックスはローカル名 `collection` を持ちます。いずれのローカル名も、コレーション `http://marklogic.com/collation/codepoint` を使用します。この場合、これらの QName（`xdmp` は、デフォルトでは `1.0-ml` ダイアレクトでその名前空間にバインドされるので、`xdmp:document` や `xdmp:collection` など）を `cts:element-value-co-occurrences` 内で、要素の QName の 1 つとして使用して、「要素値とドキュメント」の URI のペアまたは「要素値とコレクション」の URI のペアを探すことができます。必ず、これらの QName のコードポイントコレーションオプション（例えば、これらのいずれかの QName を `cts:element-value-co-occurences` の 2 番目の引数として指定する場合は、`"collation-2=http://marklogic.com/collation/codepoint"`）も指定してください。

次のような URI が `/george.xml` のドキュメントの例について考えてみましょう。

```
<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>
```

このドキュメントの作成前に、2 つの文字列要素レンジインデックス（`e:person` 要素に対するものと `e:location` 要素に対するもの）を作成しておきます。ここで、`e` は名前空間 `http://marklogic.com/entity` にバインドされています。

ここで、共起する人物と場所をすべて探す次の共起クエリが実行できます。

```
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"))
```

これにより、次の出力が生成されます。

```
<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>
```

（`item-frequency` またはデフォルトの `fragment-frequency` オプションのいずれを使用するかに応じて）各項目や各フラグメントにおける各共起ペアの頻度を取得する場合は、次のようにレキシコンのルックアップに `cts:frequency` を使用します。

```
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
:)
```

9.7 位置情報レキシコン

次の API が、位置情報レキシコンをサポートしています。

- `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`

位置情報レキシコンを使用するには、対応する適切な位置情報インデックスを作成する必要があります。例えば、cts:element-geospatial-values を使用するには、まず、位置情報要素インデックスを作成する必要があります。Admin 画面（[Databases] > [*database_name*] > [Geospatial Indexes]）または Admin API を使用して、データベース内に位置情報インデックスを作成します。

API *-boxes は、レンジのバケットを示す XML 要素を返します。各バケットは、1 つあるいは複数の cts:box 値を含んでいます。

9.8 レンジレキシコン

レンジレキシコンは、バケットごとにまとめられた値を返します。レンジは、各レキシコンにおける値の範囲です。レンジインデックスは、レンジレキシコンで指定された要素や属性で必要です。次の API が、このレキシコンをサポートしています。

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

さらに、バケットごとにまとめられた位置情報値の範囲を探すため、次のような位置情報ボックスレキシコンもあります。

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

レンジレキシコンは、XML ノードのシーケンスを返します。各バケットが 1 つのノードとなります。結果セットで cts:frequency を使用して、パケット内の項目（またはフラグメント）の個数を特定できます。「empties」オプションは、値がないバケット（つまり、頻度が 0 のバケット）には XML ノードを返します。デフォルトでは、空のバケットは結果セットに含まれません。レンジレキシコンに対するすべてのオプションの詳細については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

9.9 URI レキシコンとコレクションレキシコン

URI レキシコンおよびコレクションレキシコンはそれぞれ、データベース内のドキュメントの URI およびコレクションの URI をすべてリスト化します。これらのレキシコンをオンまたはオフにするには、Admin 画面の［Database Configuration］ページを使用します。これらのレキシコンを使用すると、データベース内のすべての URI をすばやく検索できます。次の API が、このレキシコンをサポートしています。

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

9.10 レキシコンベースのクエリの実行

レキシコンクエリは、適切なレキシコンから単語（または値レキシコンの場合は値）のシーケンスを返します。文字列値の場合、単語または値はコレーションの順序で返され、タームは大文字 / 小文字および発音符号が区別されます。他のデータ型の場合、「より大きい」値が「より小さい」値の前に返される順序で値が返されます。このセクションでは、レキシコン API のリストを示し、レキシコンベースのクエリの実行方法についていくつかの例と説明を記載します。次のパートから構成されます。

- レキシコン API
- cts:query 式へのレキシコン検索の制約
- マッチレキシコン API の使い方
- レキシコンのタームを含んでいるフラグメントの個数を求める

9.10.1 レキシコン API

次のビルトインの検索用 XQuery API を使用して、レキシコンベースのクエリを実行します。

- 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

レキシコンに基づくクエリを実行するには、適切なレキシコンを作成する必要があります。レキシコンが作成されていない場合、レキシコンクエリは例外をスローします。

API `cts:*-words` は、レキシコン内のすべての単語（またはオプションの `$start` パラメータが使用されている場合、開始ポイント以降のすべての単語）を返します。API `cts:*-match` は、ワイルドカードのパターンとマッチするレキシコン内の単語のみを返します。

個々の関数の詳細については、『MarkLogic XQuery and XSLT Function Reference』の「search APIs」を参照してください。

9.10.2 cts:query 式へのレキシコン検索の制約

レキシコン API の `$query` オプションを使用して、レキシコン検索を特定の `cts:query` 式と一致するフラグメントに制約できます。`$query` オプションを指定すると、レキシコン検索で、指定された `cts:query` 式と一致するフラグメント内のすべてのターム（レキシコン値クエリの場合は値）が返されます。

例えば、次は、SCENE レベルでフラグメント化されているシェークスピアの全戯曲のデータベースに対するクエリです。

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

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

このクエリは、次のクエリと一致するすべてのフラグメントのなかで、et という単語から始まる、ワードレキシコンの最初の 10 語を返します。

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

シェークスピアデータベースの場合、このクエリとマッチするシーンが 2 つあります。1 つは、『*ジュリアス・シーザー*』、もう 1 つは『*ヘンリー五世*』からのシーンです。これは、検索から $query パラメータを省略した場合とは異なる単語の集まりであることに注意してください。次に、$query パラメータがないクエリを示します。結果は、シェークスピアの全戯曲のワードレキシコン全体で、et という単語から始まる 10 語を表します。

```
cts:words("et")

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

レキシコンのルックアップを cts:query 式に制約すると、cts:query 式が true を返すあらゆるフラグメントのレキシコン項目が返されることに注意してください。cts:query 式に対しては、フラグメント内でマッチが実際に発生していることを検証するためのフィルタリングは実行されません。設定したインデックスオプションによっては、実際にはマッチがないのに true が返される可能性があります。例えば、データベース設定で fast element word searches がオンになっていない場合、単語と要素の両方がフラグメント内に存在するために cts:element-word-query がフラグメントとマッチしても、それらが同じ要素内には存在していない可能性があります。これらの矛盾は cts:search のフィルタリングで解決できますが、$query オプションを使用するレキシコン API では解決されません。これらの動作の詳細については、『Query Performance and Tuning Guide』の「Understanding the Search Process」および「Understanding Unfiltered Searches」を参照してください。

9.10.3 マッチレキシコン API の使い方

各タイプ（単語、要素単語、要素属性単語、要素値、および要素属性値）のレキシコンには、ワイルドカードパターンを使用して、返されるレキシコンエントリーを制約できる関数（cts:*-match）があります。API cts:*-match は、ワイルドカードパターンとマッチするレキシコン内の単語または値のみを返します。次のクエリは、レキシコン内の zou から始まるすべての単語を探します。

```
cts:word-match("zou*")

=> Zounds zounds
```

検索のデフォルトでは、ワイルドカードパターンのベースにある文字がすべて小文字であるときにデフォルトで大文字 / 小文字が区別されないため、大文字と小文字の両方の単語を返します。パターンのマッチに、大文字 / 小文字の区別あり、発音符号による区別あり、または他のオプションを使用したい場合は、適宜オプションをクエリに追加します。例えば次のようになります。

```
cts:word-match("zou*", "case-sensitive")

=> zounds
```

クエリオプションの詳細については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。ワイルドカード検索の詳細については、「ワイルドカード検索の概要と使い方」（405 ページ）を参照してください。

9.10.4 レキシコンのタームを含んでいるフラグメントの個数を求める

レキシコンは、データベース内の一意の（ユニークな）タームを含んでいます。クエリで制約されたワードレキシコン検索に続いて推定を実行する際、冗長なディスク I/O を最小限に抑えて、クエリをできる限り効率的に解決するには、`cts:word-query` が次の特性を備えている必要があります。

- `unstemmed`、`case-sensitive`、および `diacritic-sensitive` オプションを指定します。
- `weight` として 0 を指定します。

これらの特性により、推定される単語がレキシコンから返される単語とまったく同じになることが保証されます。

例えば、レキシコンのタームを含んでいるフラグメントの個数を出す場合は、次のようなクエリを実行できます。

```
<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>
```

このクエリは、レキシコンのタームごとに 1 つずつ word 要素を返します。また指定されたディレクトリ（/）内の、aardvark というタームから始まるマッチしたタームと、そのタームが含まれたフラグメントの件数も返します。次に、このクエリによる出力の例を示します。

```
<words>
  <word text="aardvark" count="10"/>
  <word text="aardvarks" count="10"/>
  <word text="aardwolf" count="5"/>
...
</words>
```

10.0 cts:query 式でのレンジクエリの使用

MarkLogic サーバーでは、`cts:query` 式内でレンジインデックスを利用して、要素または属性内の値のレンジで検索を制約できます。この章では、このレンジクエリの詳細について説明します。次のセクションから構成されます。

- レンジクエリの概要
- レンジクエリの cts:query コンストラクタ
- レンジクエリの例

10.1 レンジクエリの概要

このセクションでは、レンジクエリについて、またレンジクエリを使用する理由の概要を示します。次のセクションから構成されます。

- レンジクエリの使い方
- レンジクエリを使用するための要件
- レンジクエリのパフォーマンスとコーディング上の利点

10.1.1 レンジクエリの使い方

レンジクエリは、検索を値のレンジ（範囲）で制約するためのものです。例えば、2005 年に発行されたすべての記事を探す必要があるときに、コンテンツに `PUBLISHDATE` という `xs:date` 型の要素（あるいは属性やプロパティ）がある場合、要素 `PUBLISHDATE` に関するレンジインデックスを作成し、検索で `PUBLISHDATE` が 2004 年 12 月 31 日から 2006 年 1 月 1 日までのすべての記事を返すように指定できます。この要素にはレンジインデックスがあるため、MarkLogic サーバーはきわめて効率的にクエリを解決できます。

さまざまな XML データ型に対してレンジクエリを作成できます。このため、さまざまな種類のコンテンツにおいて柔軟にレンジクエリを利用し、検索を制約できます。一般的に、値での制約が必要な場合はレンジインデックスを作成し、結果の範囲を制約するためにレンジクエリを使用します。

10.1.2 レンジクエリを使用するための要件

`cts:search` 操作でレンジクエリを使用する際には、次の要件に留意してください。

- レンジクエリを使用するには、結果を制約する要素や属性にレンジインデックスを定義する必要があります。
- レンジインデックスには、レンジクエリに指定されたのと同じコレーションを使用する必要があります。
- レンジクエリにコレーションが指定されていない場合、クエリにはクエリ自体のコレーションが適用されます（例えば、XQuery のプロローグでコレーションが指定されている場合、それが使用されます）。コレーションのデフォルト値の詳細については、「コレーションのデフォルトの決定方法」（500 ページ）を参照してください。

レンジクエリにはレンジインデックスが必要です。このため、レンジクエリを使用するとレンジインデックスが容量を消費し、MarkLogic サーバーが実行されるマシンのメモリ使用量が増し、読み込みと再インデックス付けの時間が増えることに留意してください。このように、レンジクエリは完全に「タダで使える」わけではありません。ただし、レンジクエリの数が比較的少ない場合、大量のリソースを使用することはありません。使用されるリソースの量は、コンテンツに大きく依存します。つまり、指定された要素や属性を含むドキュメントの数、コンテンツ内でそれらの要素 / 属性が出現する頻度、コンテンツセットの大きさなどに依存します。多くのパフォーマンス改善の場合と同様に、考慮すべきトレードオフが存在します。影響を分析するのに一番いい方法は、実際に試してみてコストがそのパフォーマンス改善に見合うかどうかを確認することです。レンジインデックスの詳細とその作成手順については、『Administrator's Guide』の「Range Indexes and Lexicons」の章を参照してください。

10.1.3 レンジクエリのパフォーマンスとコーディング上の利点

レンジクエリで表現できる内容はほぼすべて、XPath 式でも述語を使用して表現できます。レンジクエリには、XPath の述語と比べて次の 2 つの大きな利点があります。

- パフォーマンス
- コーディングの容易さ

`cts:query` 式でレンジクエリを使用した場合、XPath の述語よりも高速になります。レンジインデックスはインメモリ構造です。レンジクエリでは必ずレンジインデックスを使用するため、通常は非常に高速です。XPath の述語を指定する際、レンジインデックスには制限がありません。このため、多数のフラグメントをスキャンする述語が指定される（つまり長い時間がかかる）可能性があります。また、レンジクエリは `cts:query` オブジェクトであり、これを登録済みのクエリで事前にコンパイルできます。これによりパフォーマンスが向上します。

レンジクエリには、XPath の述語と比べてコーディングに関する利点もあります。レンジクエリは、リーフレベルの cts:query コンストラクタであるため、他のコンストラクタ（他のレンジクエリコンストラクタも含む）と組み合わせて複雑な式を作ることができます。XQuery コードを記述し、フォーム（ドロップダウンリスト、テキストボックス、ラジオボタンなど）からユーザー入力を取得し、これを使用してきわめて複雑な cts:query 式を生成することがかなり簡単にできます。これは、XPath 式では非常に困難です。cts:query 式の詳細については、「cts:query 式の構成」（239 ページ）を参照してください。

10.2 レンジクエリの cts:query コンストラクタ

レンジクエリコンストラクタには、次の XQuery API が含まれます。

- cts:element-attribute-range-query
- cts:element-range-query
- cts:path-range-query
- 対応するアクセサ関数

各 API は、入力として QName、ある種の演算子（>=、<= など）、値、およびコレーションを取ります。これらの API およびシグネチャの詳細については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

> **注：** リリース 3.2 では、レンジクエリは、スコアに影響しません（cts:query コンストラクタで指定された重みに関わらず）。

10.3 レンジクエリの例

次に、レンジクエリコンストラクタを使用するいくつかの例を示します。

URI が /dates.xml である、次の構造のドキュメントについて考えてみましょう。

```
<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>
```

ここでは QName `date` に対する `xs:date` 型の要素レンジインデックスが定義済みであると仮定しています（この場合、レンジインデックスを定義してからドキュメントを読み込むか、あるいはデータベースの再インデックス付けを完了する必要があることに注意してください)。

これで、`cts:element-range-query` コンストラクタを使用してクエリを発行できるようになりました。次のクエリは、ドキュメント `/dates.xml` の `entry` 要素から、2000 年 1 月 1 日よりも前に発生したエントリーを検索します。

```
cts:search(doc("/dates.xml")/root/entry,
  cts:element-range-query(xs:QName("date"), "<=",
      xs:date("2000-01-01") ) )
```

このクエリは次のノードを返します（このレンジクエリを満たすノードは 1 つしかありませんでした)。

```
<entry>
    <date>1971-12-23</date>
    <info>Some different information.</info>
</entry>
```

次のクエリでは、`cts:and-query` を使用して、2006 年 1 月 1 日よりも後、また 2008 年 1 月 1 日よりも前という 2 つの日付範囲を組み合わせています。

```
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") ) )) )
```

このクエリにより、次の 2 つのノードが返されます。

```
<entry>
    <date>2007-01-01</date>
    <info>Some information.</info>
</entry>

<entry>
    <date>2006-06-23</date>
    <info>Some other information.</info>
</entry>
```

11.0 集計関数の使用

この章では、集計（aggregate）のビルトイン関数とユーザー定義関数（UDF）を使用して、レキシコンおよびレンジインデックスの値を分析する方法について説明します。

この章は、以下のセクションで構成されています。

- 集計関数の概要
- ビルトイン集計関数の使用
- 集計ユーザー定義関数の使用

11.1 集計関数の概要

集計関数は、1 つあるいは複数のレンジインデックスの値を対象にして演算を実行します。例えば、要素、属性、またはフィールドのレンジインデックスについて合計や度数を算出します。集計関数は、少数の結果を生成する分析に最適です（複数のレンジインデックスの値から 1 つの値を算出するなど）。

集計関数はインデータベース MapReduce を使用しており、次のような理由でパフォーマンスが大幅に改善されています。

- クラスタ内のホスト間、さらには各ホスト内のデータベースフォレスト間で分析が並列化されます。
- 分析はデータの近くで実行されます。

MarkLogic サーバーは、一般的な算術演算や統計処理用にビルトイン集計関数を提供します。集計用の UDF（ユーザー定義関数）画面を使用して、独自の集計関数も実装できます。詳細については、『Application Developer's Guide』の「Implementing an Aggregate User-Defined Function」を参照してください。

11.2 ビルトイン集計関数の使用

MarkLogic サーバーは、XQuery、REST、および Java の API からアクセス可能な、次のビルトイン集計関数を提供します。

XQuery 関数	REST および Java の集計名
cts:avg-aggregate	avg
cts:correlation	correlation
cts:count-aggregate	count
cts:covariance	covariance
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

下の表は、XQuery、REST、および Java の API で直接、集計関数を呼び出す方法をまとめたものです。

インターフェイス	手法	例

XQuery	XQuery コードから直接、ビルトイン関数を呼び出します。	`cts:sum-aggregate(`
		`  cts:element-reference(`
		`  xs:QName("Amount")`
		`  )`
		`)`

インターフェイス	手法	例
REST	GET */version*/values/*{name}* リクエストを送信します。関数名は、aggregates リクエストパラメータで指定します。詳細については、『REST Application Developer's Guide』の「Analyzing Lexicons and Range Indexes With Aggregate Functions」を参照してください。	GET /v1/values/amount?options=my-index-defns&**aggregate=sum**

Java	ValuesDefinition.setAggregate で集計名を指定して、ValuesDefinition を QueryManager.values または QueryManager.tuples に渡します。詳細については、『Java Application Developer's Guide』を参照してください。	QueryManager qm = ...;
		ValuesDefinition vdef =
		  qm.newValuesDefinition(...);
		vdef.setAggregate("sum");
		
		TuplesHandle t =
		  qm.tuples(vdef,
		  new TuplesHandle());

Node.js	valuesBuilder.aggregates で集計名を指定して、DatabaseClient.values.read を使用して計算を実行します。詳細については、『Node.js Application Developer's Guide』の「Analyzing Lexicons and Range Indexes with Aggregate Functions」を参照してください。	var vb = marklogic.valuesBuilder;
		
		db.values.read(
		  vb.fromIndexes('Amount')
		  .aggregates('sum')
		  .slice(0)
		)...

クエリオプションの <values/> 要素または <tuples/> 要素でも、集計関数を指定できます。例：

```
<options xmlns="http://marklogic.com/appservices/search">
  <values name="my-values">
    <aggregate apply="sum" />
    ...
  </values>
</options>
```

詳細については、「search API：概要と使い方」（ 22 ページ）、『Java Application Developer's Guide』、『REST Application Developer's Guide』、または『Node.js Application Developer's Guide』を参照してください。

11.3 集計ユーザー定義関数の使用

集計ユーザー定義関数（UDF）を作成して、1 つあるいは複数のレンジインデックスの値を分析できます。集計 UDF を使用するには、これを事前にインストールしておく必要があります。集計 UDF を作成してインストールする方法については、『Application Developer's Guide』の「User-Defined Functions」を参照してください。

集計 UDF は、処理対象のレンジインデックス値やドキュメントの個数に比例して結果の量が大きくなるような分析ではなく、少量の結果を生成する分析に最適です（複数のレンジインデックスの値から 1 つの値を算出するなど）。

UDF は、相対パスと関数名で識別されます。このパスは、このプラグインのインストール先のパスです。具体的には、*scope*/*plugin-id* です。*scope* はこのプラグインのインストール時に `plugin:install-from-zip` に渡されるスコープ、*plugin-id* はこのプラグインのマニフェストの `<id/>` で指定されている ID です。詳細については、『Application Developer's Guide』の「Installing a Native Plugin」を参照してください。

次の例では、パス `native/sampleplugin` にインストールされたプラグインが提供する「myAvg」という集計 UDF を使用します。

```
cts:aggregate("native/sampleplugin", "myAvg", ...)
```

下の表は、XQuery、Java、および RESTful なアプリケーションで集計 UDF を呼び出す方法をまとめたものです。

注： 集計 UDF に追加のパラメータを渡すことができるのは、XQuery だけです。

インターフェイス	手法	例

XQuery	集計と集計名を実装するネイティブプラグインのパスを指定して、cts:aggregate を呼び出します。4 番目の引数で集計固有のパラメータを渡します。	cts:aggregate(
		  "native/samplePlugin",
		  "myAvg",
		  cts:element-reference(
		  xs:QName("Amount")),
		  (plugin-arg1, plugin-arg2)
		)

REST	ネイティブプラグインのパスを aggregatePath に、また関数名を aggregate にそれぞれ指定して、GET リクエストを /values/{name} サービスに送信します。詳細については、『REST Application Developer's Guide』の「Analyzing Lexicons and Range Indexes With Aggregate Functions」を参照してください。	GET /v1/values/amount?options=myoptions& **aggregatePath=native/samplePlugin&aggregate=myAvg**

Java	集計名とパスを ValuesDefinition で設定してから、ValuesDefinition を QueryManager.values または QueryManager.tuples に渡します。詳細については、『Java Application Developer's Guide』を参照してください。	QueryManager qm = ...;
		ValuesDefinition vdef =
		  qm.newValuesDefinition(...);
		vdef.setAggregate("myAvg");
		vdef.setAggregatePath(
		  "native/samplePlugin");
		
		TuplesHandle t =
		  qm.values(vdef,
		  new ValuesHandle());

Node.js	valuesBuilder.udf と valuesBuilder.aggregates で集計名とパスを設定してから、DatabaseClient.values.read を使用して計算を実行します。詳細については、『Node.js Application Developer's Guide』の「Analyzing Lexicons and Range Indexes with Aggregate Functions」を参照してください。	var vb =
		marklogic.valuesBuilder;
		
		db.values.read(
		  vb.fromIndexes('Amount')
		  .aggregates(
		  vb.udf(
		  '/native/samplePlugin,
		  'myAvg'))
		  .slice(0)
		)...

クエリオプションの <values/> 要素または <tuples/> 要素でも、集計 UDF を指定できます。例：

```
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
```

詳細については、「search API：概要と使い方」（ 22 ページ）、『Java Application Developer's Guide』、または『REST Application Developer's Guide』を参照してください。

12.0 検索タームのマッチ箇所の強調表示

この章では、cts:highlight を使用して、検索クエリにマッチしたタームに任意のマークアップを適用する方法について説明します。以下のセクションで構成されています。

- cts:highlight の概要
- 一般的な検索と置換の関数
- cts:highlight のビルトイン変数
- cts:highlight を使用して、スニペットを作成する
- cts:walk と cts:highlight の比較
- 一般的な使用上の注意

cts:highlight のシンタックスについては、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

12.1 cts:highlight の概要

MarkLogic サーバーで検索を実行すると、一連のノードが返されます。それぞれのノードには、検索クエリにマッチしたテキストが含まれています。検索結果においてマッチしたタームを太字や別の色で強調表示することは、アプリケーションの要件としては一般的です。次のような目的のために設計された cts:highlight 関数を使用すると、こういった強調表示の要件を満たすことができます。

- 検索のマッチ箇所を容易に強調表示します。
- テキストを強調表示させるクエリのパフォーマンスを向上させます。
- 単純なテキストの強調表示だけでなく、複雑なアクションを実行できます。

cts:highlight 関数は、検索タームのマッチ箇所を容易に強調表示する目的で設計されていますが、汎用的な関数として実装されています。この関数は、検索のマッチ箇所を、3 番目の引数として指定された XQuery 式に置き換えます。検索タームのマッチ箇所を任意の XQuery 式に置き換えることができるので、クエリにマッチしたタームに対して、あらゆる種類の検索や置換を実行できます。cts:highlight は MarkLogic サーバーに組み込まれているので、このような検索と置換の操作はうまくいきます。

12.1.1 マッチするすべてのターム（語幹や大文字を含む）

fn:replace や fn:contains など、標準的な XQuery 文字列関数を使用してマッチ箇所を検索する場合、マッチの条件の文字列を正確に（完全に同じに）指定する必要があります。cts:search クエリにマッチした語を強調表示する場合、文字列の完全一致では、クエリにマッチする語の一部が検出されない場合があります。ただし、cts:highlight クエリでは、cts:highlight の 2 番目の引数として指定された cts:query にマッチするすべてのものが対象になります。

語幹処理による（stemmed）検索を有効にした場合、テキストの完全一致以外のものもマッチされます。例えば、run、running、および ran はすべて、run という条件のクエリとマッチします。ステミング（語幹処理）の詳細については、「ステミング検索の概要と使い方」（385 ページ）を参照してください。

同様に、クエリの実際の検索タームと大文字小文字が異なる場合もマッチとみなすことができます。さらに、ワイルドカードインデックスが有効になっている場合は、ワイルドカードを使用してさまざまなクエリでマッチできます。cts:highlight を使用するクエリは、このようなマッチ箇所をすべて検出して、指定されている式の評価結果に置き換えます。

12.2 一般的な検索と置換の関数

cts:highlight は強調表示を容易にするという目的で設計されていますが、一般的な検索と置換にも使用できます。例えば、content database というタームのインスタンスをすべて contentbase に置き換える場合、次のようなクエリを発行できます。

```
for $x in cts:search(//mynode, "content database")
return
cts:highlight($x, "content database", "contentbase")
```

このクエリはたまたま、cts:highlight と同じ検索クエリを cts:search で使用しています。このような方法は必須ではありませんが、テキストを強調表示する場合には一般的です。例えば、次のクエリは、foo という語を含むすべてのノードを検索して、そのノードの bar という語を baz という語に置き換えます。

```
for $x in cts:search(fn:doc(), "foo")
return
cts:highlight($x, "bar", "baz")
```

任意の XQuery 式を置換に使用できるので、かなり複雑な検索と置換も、比較的少ない量のコードで実行できます。

12.3 cts:highlight のビルトイン変数

cts:highlight 関数には、置換で使用できる 3 つのビルトイン変数があります。この式は、クエリのマッチごとに評価されます。したがって、各変数はクエリの一連のマッチ箇所にバインドされます。変数の値は、各反復におけるクエリのマッチ箇所の値になります。このセクションでは、この 3 つの変数について説明するとともに、その使用方法を以下のように説明します。

- $cts:text 変数を使用して、マッチしたテキストにアクセスする
- $cts:node 変数を使用して、マッチのコンテキストにアクセスする
- $cts:queries 変数を使用して、クエリに基づくロジックを提供する
- $cts:start を使用して、文字数による位置を取得する
- $cts:action を使用して、強調表示を停止する

12.3.1 $cts:text 変数を使用して、マッチしたテキストにアクセスする

$cts:text 変数は、クエリのマッチ箇所を表す文字列を保持します。例えば、データベース内に URI が test.xml である次のドキュメントが存在し、このデータベースではステミングが有効になっていると想定します。

```
<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>
```

次のように、run という語を検索するクエリのマッチ結果からテキストを強調表示できます。

```
for $x in cts:search(doc("test.xml")/root/p, "run")
return
cts:highlight($x, "run", <b>{$cts:text}</b>)
```

式 {$cts:text} はクエリのマッチ箇所ごとに評価され、クエリのマッチ箇所をその評価結果に置き換えます。run、running、および ran はすべて、run を対象とする cts:query とマッチするので、結果ではそれらの語が次のように強調表示されます。

```
<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>
```

12.3.2 $cts:node 変数を使用して、マッチのコンテキストにアクセスする

$cts:node 変数は、マッチが見つかったテキストノードへのアクセスを提供します。ノードにアクセスすることで、そのノードのコンテキストにおいて処理を実行する式を作成できます。例えば、XML において book、chapter、section、および paragraph という要素が階層構造になっている場合、強調表示式内にコードを記述し、それぞれのマッチが見つかったセクションを表示できます。次のコードは、タームが強調表示されるテキストノードの上にある、chapter という最初の要素を返す XPath ステートメントです。

```
$cts:node/ancestor::chapter[1]
```

この情報を使用して、この章（chapter）を表示するためのリンクの追加や、同一章内の他の語の検索など、いろいろな処理を実行できます。繰り返しますが、cts:highlight は検索クエリのマッチ箇所ごとに任意の XQuery 式を評価するので、その利用方法はほぼ無限です。

次の例は、直接の親が p 要素である場合は強調表示するタームを青に、それ以外の場合は赤にするという、$cts:node 変数の利用方法を示しています。

```
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> ) )
            )
```

このクエリは、次の結果を返します。

```
<root>
 <p>This is <font color="blue">blue</font>.</p>
 <p><i>This is <font color="red">red</font>italic.</i></p>
</root>
```

12.3.3 $cts:queries 変数を使用して、クエリに基づくロジックを提供する

$cts:queries 変数は、クエリのマッチ条件を満たしている cts:query へのアクセスを提供します。この情報に基づいて、クエリごとに異なる方法で強調表示するロジックを実行できます。

例えば、データベース内に URI が hellogoodbye.xml の次のドキュメントが存在すると想定します。

```
<root>
  <a>It starts with hello and ends with goodbye.</a>
</root>
```

このとき、次のクエリを実行できます。このクエリは、hello のクエリを青、goodbye のクエリを赤で表示するというシンプルなロジックに基づいています。

```
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>
```

12.3.4 $cts:start を使用して、文字数による位置を取得する

$cts:start 変数は、処理するテキストノードの文字数（$cts:node）に基づいて、マッチするテキストの開始位置（$cts:text）を返します。

12.3.5 $cts:action を使用して、強調表示を停止する

xdmp:set を使用して、$cts:action の値を変更するとともに、マッチ箇所の処理後に実行するアクションを指定します。この変数では強調表示を制御できます。通常これは、アプリケーション内でコード化されている何らかの条件（見つかったマッチの件数など）に基づきます。continue（デフォルト）で強調表示の続行、skip で現在のテキストノード内の 2 回め以降のマッチ箇所について強調表示の省略、break で残りの入力における強調表示の停止をそれぞれ指定できます。

12.4 cts:highlight を使用して、スニペットを作成する

検索においては、検索結果を強調表示する際に、ドキュメント内の該当部分だけを表示したいことがよくあります。ドキュメント内の検索条件とマッチした部分は、スニペットと呼ばれます。このセクションでは、シンプルな例を使って、cts:highlight を使用してスニペットを作成する際の基本的なデザインパターンについて説明します。この例は検索のマッチ箇所の親要素を出力するだけのものですが、同様な方法で役に立つスニペットを作成できます。典型的なスニペットでは、マッチした結果を太字で表示し、その前後の数語が表示されます。

スニペット作成の基本的なデザインパターンとしてでは、最初に cts:search でマッチを特定してから、これのマッチ箇所ごとに cts:highlight を実行してマークアップします。最後に、強調表示されているマッチ箇所に対して再帰的な変換などの処理を行い、関心があるドキュメントの対象部分を書き出します。再帰的な変換の詳細については、『Application Developer's Guide』の「Transforming XML Structures With a Recursive typeswitch Expression」を参照してください。

次の例では、シェークスピアのデータベースの検索でシンプルなスニペットを作成します。ここでは、検索とマッチしたテキストノードの親要素を返すだけです。cts:highlight を使用して、検索のマッチ箇所を含んだ要素を囲む一時的な要素（HIGHLIGHTME）を作成してから、この一時的な要素の名前を使用して、変換でマッチする要素を検索します。

```
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>
:)
```

この例は、マッチが見つかった要素（この例では 1 つだけ）と、戯曲のタイトルである TITLE 要素を返すだけです。必要に応じて任意のロジックを追加して、アプリケーションに応じたスニペットを作成できます。例えば、検索結果が含まれる幕や場のタイトルを出力する場合や、検索結果の行数を計算する場合などが考えられます。ドキュメント全体を変換できるので、コンテンツに対して興味深いさまざまな処理を簡単に実行できます。

注： 検索が返すノードの各部分（例えば、戯曲のタイトル、幕の番号、場の名前など）を処理したい場合、再帰 typeswitch が便利です。検索条件とマッチした要素を返すだけであれば、もっとシンプルな処理で対応できます。例えば、強調表示式に XPath を使用することで、次のようにシンプルにできます。

```
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()
```

12.5 cts:walk と cts:highlight の比較

関数 cts:walk は cts:highlight と似ていますが、指定された変更を適用したノードのコピーを返す代わりに、cts:walk 呼び出しで指定されたテキストノードのマッチに対する式の評価だけを返します。cts:walk はノードのコピーを作成しないので、cts:highlight よりも高速です。式の評価だけが必要な場合、cts:walk の方が cts:highlight よりも効率的です。

12.6 一般的な使用上の注意

このセクションでは、cts:highlight の使用における一般的な注意事項について説明します。以下の内容を取り上げます。

- 入力はノード 1 つだけであること
- xdmp:set 使用時の cts:highlight による副作用
- cts:similar-query または cts:element-attribute-*-query では強調表示はできない

12.6.1 入力はノード 1 つだけであること

cts:highlight への入力は、単一のノードであることが求められます。つまり、複数のノードを返す cts:search によるクエリのマッチ箇所を強調表示する場合は、次の例のように、cts:search の結果を変数にバインドする（例えば for ループなどを使用）必要があります。

```
for $x in cts:search(fn:doc(), "MarkLogic")
return
cts:highlight($x, "MarkLogic", <b>{$cts:text}</b>)
```

このクエリは、データベース内の MarkLogic を含むすべてのドキュメントを返しますが、マッチ箇所を b タグで囲みます。

> **注：** cts:highlight の入力ノードは必ずドキュメントノードまたは要素ノードである必要があり、テキストノードは使用できません。

12.6.2 xdmp:set 使用時の cts:highlight による副作用

タームを強調表示したまま、一部のインスタンスを他とは異なる方法で扱う場合、まず変数を定義します。その後 xdmp:set 関数を使用して、強調表示されたタームの処理に応じてこの変数の値を変更します。この機能の一般的な使用例は、次のとおりです。

- タームの 1 回目のインスタンスだけを強調表示します。
- 1 回目のタームはそれ以降のタームとは別の色で強調表示します。
- クエリとマッチしたターム数をカウントします。

状態を変更できるので（つまり「副作用」があります)、マッチしたタームに対して好きなようにさまざまな処理ができます。

次の例は、クエリの 1 回目のマッチを太字タグで強調表示して、それ以降のマッチ箇所についてはマッチしたテキストを返すだけのクエリです。

データベース内に URI が /docs/test.xml の次のドキュメントが存在すると想定します。

```
<html>
  <p>hello hello hello hello</p>
</html>
```

この場合、次のクエリを実行して、1 回目のマッチだけを強調表示できます。

```
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>
```

式はクエリのマッチごとに評価され、xdmp:set の呼び出しによってそれぞれのクエリのマッチ箇所の状態は変更され、条件がクエリのマッチ箇所ごとに異なる方法で評価されるという副作用があります。

12.6.3 cts:similar-query または cts:element-attribute-*-query では強調表示はできない

`cts:similar-query` や、いずれかの `cts:element-attribute-*-query` 関数を含むクエリによる結果では、`cts:highlight` を使用しても強調表示できません。これらのクエリで `cts:highlight` を使用すると、強調表示なしでノードが返されます。

13.0 位置検索アプリケーション

この章では、位置情報関数を使用する方法について説明するとともに、これらの関数を使用できるアプリケーションの種類について説明します。この章は、次のセクションから構成されます。

- MarkLogic サーバーの位置データの概要
- 位置情報の座標と領域の概要
- 位置インデックス
- API の使用
- シンプルな位置検索の例

13.1 MarkLogic サーバーの位置データの概要

位置データの最も基本的な形式は、緯度と経度の座標のペアです。MarkLogic サーバーの位置データは、XML の要素や属性、JSON プロパティによってマークアップされます。XML における位置データの表現方法はさまざまあり、MarkLogic サーバーは、複数の表現方法をサポートしています。このセクションでは、MarkLogic サーバーにおける位置データとクエリの仕組みについて説明します。このセクションは、次のパートから構成されます。

- 用語
- 座標系
- 位置クエリのタイプ
- 位置クエリ用の XQuery プリミティブ型とコンストラクタ
- WKT（Well-Known Text）マークアップ言語

13.1.1 用語

MarkLogic サーバーの位置情報機能の説明には、次の用語を使用します。

- 座標系

  位置情報の*座標系*は、地球上の位置を数値の組に対応付けます。縦軸は緯度座標、横軸は経度座標でそれぞれ表します。この両方を組み合わせることで、地球上の場所を指定する座標系が構成されます。詳細については、「MarkLogic サーバーの緯度座標と経度座標」（352 ページ）を参照してください。

- ポイント

  位置情報の*ポイント*は位置情報の座標系における点であり、特定の緯度と経度の交点を表します。詳細については、「MarkLogic サーバーのポイント」（352 ページ）を参照してください。

- 近接度

  検索結果の*近接度（proximity）*は、検索結果がドキュメント内で互いにどの程度近いかを示します。近接度は、位置検索タームを含む、任意の種類の検索タームに適用できます。例えば、特定の郵便番号エリア内の地点（ポイント）を表す語から 10 語以内にある「dog」という検索タームを検索できます。

- 距離

  2 つの位置情報オブジェクト間の*距離*は、その位置情報オブジェクト同士の地理的な近さを指しています。

13.1.2 座標系

MarkLogic サーバーは、次の 2 種類の位置データの座標系をサポートしています。

- WGS84
- Raw

MarkLogic サーバーは、デフォルトで WGS84（World Geodetic System）をジオコーディングの基礎として採用しています。WGS84 は、地球の単一の地図投影法を想定した座標系を使用します。WGS84 は、地球上の位置を表すのに幅広く使用されており、多くの衛星サービス（特に GPS）や Google マップなど、広範なサービスで採用されています。ジオコーディングの体系は他にも存在していますが、WGS84 と比較すると利点もあれば、欠点もあります（特定地域に限定すればより正確なもの、あるいはあまり使われていないものなど）。MarkLogic サーバーは、地球上の位置表現の標準として幅広く受け入れられている WGS84 を採用しています。WGS84 の詳細については、http://en.wikipedia.org/wiki/World_Geodetic_System を参照してください。

地球の形状にではなく、平面上にポイントを示す場合には、Raw 座標系を使用できます。

13.1.3 位置クエリのタイプ

MarkLogic サーバーでは、次のタイプの位置クエリがサポートされています。

- ポイントクエリー単一のポイント（点）とマッチ
- ボックスクエリー長方形内の任意のポイント
- 半径クエリーあるポイントを中心として指定された距離内の、任意のポイント
- 多角形クエリー指定された *n* 角形内の任意のポイント

位置情報の `cts:query` コンストラクタは、他の `cts:query` コンストラクタの場合と同様に作成できます。`cts:query` コンストラクタの作成の詳細については、「cts:query 式の構成」（239 ページ）を参照してください。

位置クエリコンストラクタに加えて、ビルトイン関数も用意されています（「位置情報の操作」（362 ページ）に列挙されています）。これで位置データの演算（距離の計算など）を実行します。

> **注：** 位置クエリコンストラクタを使用するには、有効な位置情報ライセンス（geospatial license）のキーが必要です。有効なライセンスキーがない場合、位置クエリを含む検索を実行すると、例外がスローされます。

13.1.4 位置クエリ用の XQuery プリミティブ型とコンストラクタ

位置クエリ用に、MarkLogic サーバーには、次の XQuery プリミティブ型が用意されています。

- `cts:box`
- `cts:circle`
- `cts:complex-polygon`
- `cts:linestring`
- `cts:point`
- `cts:polygon`

これらのプリミティブ型は、位置情報の `cts:query` コンストラクタ（例えば、`cts:element-geospatial-query`、`cts:element-attribute-pair-geospatial-query`、`cts:element-pair-geospatial-query`、`cts:json-property-geospatial-query` など）で使用します。それぞれの XQuery プリミティブ型（`cts:box`、`cts:circle`、`cts:complex-polygon`、`cts:linestring`、`cts:point`、`cts:polygon`）は、`cts:region` ベース型のインスタンスです。これらの型は領域を定義します。検索のコンテキストにおいてマッチするデータが領域に含まれている場合、クエリは true を返します。

また、各プリミティブ型にコンストラクタがあります。コンストラクタは、データに対応する型を生成します。データから生成できない場合は、例外がスローされます。MarkLogic サーバーは、データをパースしてポイントを抽出し、該当する型を生成します。例えば次の例は、文字列から、スペース区切りのポイントを含む cts:polygon を生成します。

```
cts:polygon("38,-10 40,-10 39, -15")
```

次の例は、数値で表された座標から cts:point を組み立てます。

```
cts:point(38.7, -10.3)
```

13.1.5 WKT（Well-Known Text）マークアップ言語

MarkLogic は、位置データの表現用に WKT（Well-Known Text）マークアップ言語をサポートしています。MarkLogic で使用できる WKT オブジェクトは、POINT、POLYGON、LINESTRING、TRIANGLE、MULTIPOINT、MULTILINESTRING、MULTIPOLYGON、および GEOMETRYCOLLECTION です。WKT を位置クエリで使用するには、cts:parse-wkt 関数を使用して、WKT を cts:region 項目のシーケンスに変換します。これで、cts:region 項目を任意の位置情報の cts:query コンストラクタで使用できます。

例えば、次のコードは、外側境界と内側境界を含む cts:complex-polygon を返します。

```
cts:parse-wkt("
 POLYGON(
  (0 0, 0 10, 10 10, 10 0, 0 0),
  (0 5, 0 7, 5 7, 5 5, 0 5) )" )
```

次の表に、WKT の種類と MarkLogic 位置情報の種類の対応を示します。

WKT の種類	MarkLogic 位置情報の種類
POINT	cts:point
POINT EMPTY	cts:point (flagged as empty)
POLYGON	cts:complex-polygon | cts:polygon
POLYGON EMPTY	cts:complex-polygon (flagged as empty)
LINESTRING	cts:linestring
LINESTRING EMPTY	cts:linestring (flagged as empty)
TRIANGLE	cts:polygon
TRIANGLE EMPTY	cts:complex-polygon (flagged as empty)

WKT の種類	MarkLogic 位置情報の種類
MULTIPOINT	cts:point*
MULTIPOINT EMPTY	()
MULTILINESTRING	cts:linestring*
MULTILINESTRING EMPTY	()
MULTIPOLYGON	(cts:polygon | cts:complex-polygon)*
MULTIPOLYGON EMPTY	()
GEOMETRYCOLLECTION	cts:region*
GEOMETRYCOLLECTION EMPTY	()
その他	throws XDMP-BADWKT

同様に、cts:to-wkt 関数で cts:region を WKT に変換できます。例えば、次の例は、WKT POINT を返します。

```
cts:to-wkt(cts:point(1, 2))
```

cts:circle または cts:box は、WKT に変換できません。WKT の詳細については、http://en.wikipedia.org/wiki/Well-known_text を参照してください。

13.2 位置情報の座標と領域の概要

このセクションでは、位置情報の座標のルールとさまざまな領域（cts:box、cts:circle、cts:complex-polygon、cts:linestring、cts:point、cts:polygon）について説明します。このセクションは、次のパートから構成されます。

- 基本的な座標とポイントの概要
- 位置情報のボックスの概要
- 位置情報の多角形の概要：多角形、複雑な多角形、ラインストリング
- 位置情報の円の概要
- ラインストリング

13.2.1 基本的な座標とポイントの概要

位置情報の領域が MarkLogic サーバーでどのように定義されるのかを理解するには、まず座標とポイント（地点）の基本について理解しておく必要があります。このセクションでは、次の内容について説明します。

- MarkLogic サーバーの緯度座標と経度座標
- MarkLogic サーバーのポイント

13.2.1.1 MarkLogic サーバーの緯度座標と経度座標

緯度は、南北の座標を示します。赤道は 0 度、北極点が 90 度、南極点が -90 度になります。90 度を上回る座標や、-90 度を下回る座標を指定した場合、それぞれは 90 または -90 として扱われます。

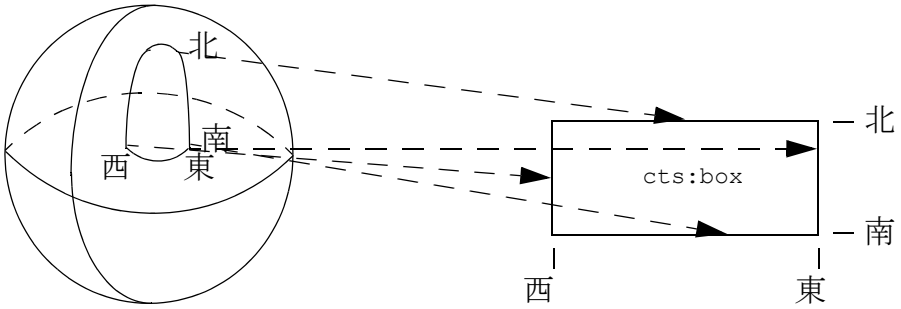
経度は、東西の座標を示します。本初子午線（グリニッジ子午線）の 0 度から始まり、地球の周囲を東に向かって 180 度、西に向かって -180 度まであります。360 度移動すると、本初子午線に戻ります。本初子午線から西へ向かうと、数値はマイナスになります。例えば、ニューヨーク市は本初子午線の西に位置し、その経度は -73.99 度です。経度座標に 360 の倍数を足したり引いたりしても、座標は同じになります。

13.2.1.2 MarkLogic サーバーのポイント

ポイントとは、緯度座標と経度座標のペアです。これら座標の交点は、地球上の位置に対応しています。例えば、カリフォルニア州サンフランシスコの座標は、緯度 37.655983 および経度 -122.425525 のペアです。MarkLogic サーバーでは、ポイントの定義に `cts:point` 型を使用します。`cts:point` コンストラクタを使用して、一連の座標からポイントを生成します。またポイントは、MarkLogic サーバーの他の領域（`cts:box`、`cts:polygon`、および `cts:circle`）の定義にも使用します。

13.2.2 位置情報のボックスの概要

位置情報のボックスを使用すると、4 つの座標で定義される領域を作成できます。4 つの座標で、*位置情報のボックス*を定義します。このボックスが平面に射影されると、長方形になります。ポイントがこのボックスの境界内に存在する場合、そのポイントはこの位置情報のボックス内に存在することになります。ボックスを定義する 4 つの座標は、ボックスの東西南北の境界を表します。ボックスは二次元であり、地球を三次元から平面に射影したものです。地球の表面上は、ボックスのエッジは弧になります。しかしこれらの弧は平面に射影されると二次元の緯線と経線になり、次の図に示すように、これらの線によって定義される領域は長方形になります（`cts:box` による表現）。



地球(球面)からボックス(平面)への座標の射影

次に示すのは、位置ボックスに関連する前提と制限事項です。

- ボックスの 4 つのポイントは、それぞれ順番に南、西、北、東を表します。
- 地球から二次元の平面への射影で考えると、ボックスの領域は、まず南西の端から南東の端に行き（日付変更線を越えても同様)、北に向かって北東の端（両極点の境界）まで、西に向かって北西の端まで、さらに南に向かって最初の南西の端に戻ることで決定されます。
- ボックスの東西の境界を決定するときには、常に西の経度から東の経度に向かいます。西のポイントが日付変更線の東に位置し、かつ東のポイントが日付変更線の西に位置している場合、元の東のポイントに戻るまで、地球の周囲を東に向かうことになります。
- 同様に、ボックスの南北の側面を決定するときには、常に南の緯度から北の緯度に向かいます。ただし、北のポイントを南のポイントの南側に配置できません(極点は越えられません)。極点を越えた場合、そのボックスを使用した検索では、XDMP-BADBOX ランタイムエラーがスローされます（北極点より北へは進めないため)。このエラーは、ボックスの作成時ではなく、検索時に発生することに注意してください。
- 東の座標が西の座標と同じ場合、経度だけが考慮されます。同様に、北の座標が南の座標と同じ場合、その緯度だけが考慮されます。これにより、次のようになります。
    - 西の座標と東の座標が同じ場合、ボックスは縦方向の線になります（その経度の、南の座標と北の座標間の線)。
    - 南の座標と北の座標が同じ場合、ボックスは横方向の線になります（その緯度の、西の座標と東の座標間の線)。
    - 西の座標と東の座標が同じで、かつ南の座標と北の座標が同じ場合、ボックスはその座標によって示されたポイント（地点）になります。
- ボックスの境界をボックスに含む / 含まないのかは、クエリオプションによって異なります。この動作を制御するため、位置情報の cts:query コンストラクタには、さまざまな境界オプションが用意されています。

13.2.3 位置情報の多角形の概要：多角形、複雑な多角形、ラインストリング

位置情報の多角形を使用すると、位置クエリ用に n 角形の境界の領域を作成できます。この境界によって、地球上のあらゆる領域を表現できます（次に説明しているように例外もありますが）。例えば多角形を作成して、国や地理的領域を表現できます。
MarkLogic でこういった位置情報の領域を作成する場合、多角形、複雑な多角形、ラインストリングという 3 種類の方法があります。このセクションでは、多角形のいくつかの特性について説明します。このセクションは、次のパートから構成されます。

- 多角形の概要
- 多角形
- 複雑な多角形
- ラインストリング

13.2.3.1 多角形の概要

多角形は、円やボックスよりも柔軟性に優れています。この柔軟性と引き換えに、位置情報の多角形は、位置情報のボックスほど高速でも正確でもありません。多角形の効率は、多角形の辺の数に比例します。例えば、一般的な 10 角形は一般的な 1000 角形よりも高速で処理されます。この処理速度は、多角形の位置や位置データの特性など、さまざまな要因によって異なります。

次に示すのは、位置情報の多角形に関連する前提と制限事項です。

- 地球は球体であり、その中心を通る大圏（最大の円）によって分割した場合、その 1 つは本初子午線（グリニッジ子午線）を通る経度によって分割されるもの、もう 1 つは赤道の位置での緯度によって分割されるものです。
- 多角形の各辺は、ある点から球面である地球表面への半球の射影に相当します。したがって、これらの線は単一の平面内にはなく、地球の曲面（ほぼ球体です）に従います。
- 同一の多角形内に、極点を 2 つとも含めることはできません。したがって、極点を 2 つとも境界として設定することはできません（境界が多角形に含まれるかどうかにかかわらず）。つまり、緯度を 180 度対象とすることはできないということになります。
- 多角形のエッジは、180 度未満であることが求められます。つまり、多角形の隣接する 2 つのポイントは、地球の経度または緯度の半分未満となっている必要があります。180 度以上に広がる多角形は可能ですが、3 つ以上のポイントを使用する必要があります。したがって、隣接する頂点は、経度を 180 度を超えて離すことはできません。この結果、いずれかのエッジに沿っている場合を除いて、多角形に極点を含めることはできません。また、多角形のエッジを形成する 2 つのポイントが 180 度を超えて離れている場合、MarkLogic サーバーは常に 180 度未満の方向を選択します。

- 位置クエリは、cts:query コンストラクタで命名された要素および属性に制約されます。単一のクエリで複数の形式に対応するには、cts:or-query を使用します。
- 多角形が座標系に対して有効ではない場合、検索時に、ランタイム例外がスローされることがあります（座標系は、cts:polygon の作成時ではなく、検索時に指定されます）。
- 多角形の境界を多角形に含む / 含まないのかは、クエリオプションによって異なります。この動作を制御するため、位置情報の cts:query コンストラクタには、さまざまな境界オプションが用意されています。
- 地球は球体であると想定し、ポイントは float で表現されるので、結果は必ずしも正確ではありません。多角形による表現は、地球を球体としてモデル化しているため、他の方法ほど正確ではありません。ちょっとわかりにくいですが、地球上のポイントを float で表現している理由は、double を使っても精度が大きく向上しないからです（つまり、地球はそれほど大きくないということです）。

13.2.3.2 多角形

cts:polygon は、多角形の頂点となるポイントを指定して作成します。これにより作成された領域に囲まれているポイントはすべて、領域内に含まれていると定義されます。

13.2.3.3 複雑な多角形

cts:complex-polygon は、ゼロ個以上の他の多角形の内部に多角形を作成することで作成できます。作成される複雑な多角形は、外側の多角形と内側の多角形の間の部分になります。また、穴のない（内側の多角形が存在しない）cts:complex-polygon で cts:polygon を型抜きできます。内側の多角形を複数指定する場合は、相互に重ならないようにする必要があります。

13.2.3.4 ラインストリング

ラインストリングは、一連の弧をつなげたものです。これらの弧は、多角形のような閉ループを必ず形成するわけではありません（閉ループを形成することも可能です）。「直線（ライン）」は地球表面に射影された場合、実際には弧になります。ラインストリングは、等しい / 等しくないの判断に使用できます。それぞれの頂点がすべて等しい場合（または両方に頂点がない場合）、2 つのラインストリングは等しいことになります。

cts:linestring で cts:polygon を型抜きすると、同じ一連のラインストリングをたどって多角形を閉じる「平らな」多角形を作成できます。

13.2.4 位置情報の円の概要

位置情報の円を使用すると、ポイント（中心点）と半径（単位：マイル）による境界を持つ領域を定義できます。ポイントと半径によって円が定義され、円の内部に存在するものはすべて、`cts:circle` の境界内に存在することになります。また、円の境界を円に含む / 含まないかは、クエリオプションによって異なります。この動作を制御するため、位置情報の `cts:query` コンストラクタには、さまざまな境界オプションが用意されています。

13.3 位置インデックス

位置データは、ドキュメント内で XML マークアップとして格納されるので、位置情報の XML マークアップでコンテンツを制約するクエリを実行できます。位置インデックスがあると、位置クエリが高速化されるとともに、位置情報レキシコンクエリが可能になります。これでコンテンツ内でマークアップされた位置データをフル活用できるようになります。このセクションでは、さまざまな種類の位置インデックスについて説明します。このセクションは、次のパートから構成されます。

- さまざまな種類の位置インデックス
- 位置インデックスの位置
- 位置情報レキシコン

13.3.1 さまざまな種類の位置インデックス

Admin 画面を使用して、必要なインデックスを作成します（[Database] > [*database_name*] > [Geospatial Indexes]）。次のセクションでは、位置インデックスのタイプごとの位置データの構造について説明するとともに、各インデックスで使用可能な位置オプションについても説明します。

- 位置情報要素のインデックス
- 位置情報子要素のインデックス
- 位置情報要素ペアのインデックス
- 位置情報属性ペアのインデックス
- 位置情報パスのレンジインデックス
- 位置情報 JSON プロパティのインデックス
- 位置情報子 JSON プロパティのインデックス
- 位置情報 JSON プロパティペアのインデックス

注： 位置インデックスを作成または使用するには、有効な位置情報ライセンス（geospatial license）のキーが必要です。

13.3.1.1 位置情報要素のインデックス

位置情報の要素のインデックスでは、位置データはスペースまたは句読点（+、-、. を除く）で区切られた要素コンテンツによって表現されます。

```
<element-name>37.52  -122.25</element-name>
```

point 形式の場合、最初のエントリーは緯度、2 番目は経度を表します。long-lat-point 形式の場合、最初のエントリーは経度、2 番目は緯度を表します。その他のエントリーも持つことができますが、無視されます。例えば、KML にはこれ以外に高度がありますが、これは無視されます。

13.3.1.2 位置情報子要素のインデックス

位置情報の子要素のインデックスでは、位置データはスペースまたは句読点（+、-、または . を除く）で区切られた要素コンテンツから得られますが、対象は特定の要素の子に該当する要素に限定されます。

```
<element-name1>
  <element-name2>37.52  -122.25</element-name2>
</element-name1>
```

point 形式の場合、最初のエントリーは緯度、2 番目は経度を表します。long-lat-point 形式の場合、最初のエントリーは経度、2 番目は緯度を表します。

13.3.1.3 位置情報要素ペアのインデックス

位置情報の要素ペアのインデックスでは、位置データは特定の要素の子要素の特定のペアから得られます。

```
<element-name>
  <latitude>37.52</latitude>
  <longitude>-122.25</longitude>
</element-name>
```

13.3.1.4 位置情報属性ペアのインデックス

位置情報の属性ペアのインデックスでは、位置データは特定の要素に含まれる特定の属性のペアから得られます。

```
<element-name latitude="37.52" longitude="-122.25"/>
```

13.3.1.5 位置情報パスのレンジインデックス

位置情報のパスのレンジインデックスでは、位置データは位置情報の要素のインデックスと同じ方法で表現されます。インデックスの XML 要素、XML 属性、および JSON プロパティはパス式によって定義されます。

例えば、次の XML データは、

```
<a:data>
  <a:geo>37.52  -122.25</a:geo>
</a:data>
```

次のパス式でインデックスが設定されます。

```
/a:data/a:geo
```

この位置データは XML 属性としても表現できます。次の XML データは、

```
<a:data>
  <a:geo data="37.52  -122.25"/>
</a:data>
```

次のパス式でインデックスが設定されます。

```
/a:data/a:geo/@data
```

同様に、次の JSON データは、

```
{ "geometry" : {
    "type": "Point",
    "coordinates": [37.52, -122.25]
  }
}
```

次のパス式でインデックスが設定されます。

```
/geometry[type="Point"]/array-node("coordinates")
```

注： Admin 画面で位置情報のパスのレンジインデックスを作成した後では、パス式は変更できません。変更したい場合は、既存の位置情報のパスのレンジインデックスを削除して、更新したパス式で新しいものを作成する必要があります。

13.3.1.6 位置情報 JSON プロパティのインデックス

ポイントの座標が単一の JSON プロパティに含まれている場合、位置情報の要素のインデックスを使用して、JSON ドキュメント内の位置データにインデックスを設定します。位置データは、スペースや句読点で区切られた文字列の値、または値の配列として、プロパティ値で表現する必要があります。例：

```
"prop-name": "37.52 -122.25"

"prop-name": [37.52, -122.25]
```

point 形式の場合、最初のエントリーは緯度、2 番目は経度を表します。long-lat-point 形式の場合、最初のエントリーは経度、2 番目は緯度を表します。値にはその他のエントリーも含めることができますが、無視されます。例えば、KML に用意されている追加の高度座標も使用できますが、無視されます。

13.3.1.7 位置情報子 JSON プロパティのインデックス

特定のプロパティに含まれている座標プロパティにインデックス付けを制限する場合、位置情報の子要素のインデックスを使用して、JSON の位置データにインデックスを設定します。位置データは、スペース、句読点で区切られた文字列の値、または値の配列として、子プロパティ値で表現する必要があります。

例えば、次のいずれかのデータの場合、「theParent」を親要素（プロパティ）、「theChild」を子要素（プロパティ）としてそれぞれ指定することで、位置情報の子要素のインデックスを作成できます。

```
"theParent": {
  "theChild": "37.52 -122.25"
}

"theParent": {
  "theChild": [37.52, -122.25]
}
```

point 形式の場合、最初のエントリーは緯度、2 番目は経度を表します。long-lat-point 形式の場合、最初のエントリーは経度、2 番目は緯度を表します。

13.3.1.8 位置情報 JSON プロパティペアのインデックス

ポイントの座標が兄弟 JSON プロパティに含まれている場合、位置情報の子要素のインデックスを使用して、JSON の位置データにインデックスを設定します。例：

```
"theParent" : {
  "latitude": 37.52,
  "longitude": -122.25
}
```

13.3.2 位置インデックスの位置

それぞれの位置インデックスには、位置（positions）オプションがあります。位置オプションによって、ドキュメント内の位置データ間での距離によって制限することでクエリが高速化されます（例えば、`cts:near-query` の使用時）。また、要素の positions がデータベースで有効になっている場合、位置クエリを含む XML 要素や JSON プロパティのクエリでインデックス解決が改善（推定精度が向上）します（位置クエリにおいて、位置データに対して positions が有効になっている位置インデックスの場合）。

13.3.3 位置情報レキシコン

位置インデックスによって、位置情報レキシコンの検索が有効になります。レキシコンのルックアップによって、位置情報値の取得が大幅に高速化されます。位置情報レキシコンの詳細については、「位置情報レキシコン」（321 ページ）を参照してください。

13.4 API の使用

このセクションでは、位置情報 API の概要について説明します。このセクションは、次のパートから構成されます。

- 位置クエリを実行する基本的な手順
- 位置情報の cts:query コンストラクタ
- 領域の位置情報値コンストラクタ
- 位置情報の形式変換関数
- 位置情報の操作

13.4.1 位置クエリを実行する基本的な手順

位置情報 API の使用は、`cts:query` を `cts:search` の 2 番目のパラメータ（または 2 番目のパラメータの構成要素）として使用する他の `cts:query` コンストラクタを使用する場合と同様です。基本的な手順は、次のとおりです。

1. 位置データをデータベースに読み込みます。
2. 位置インデックスを作成します（高速化のためのオプション）。
3. 位置情報の `cts:query` コンストラクタで使用するプリミティブ型を作成します。
4. 位置情報のプリミティブ型を使用して位置クエリを作成します。
5. 位置クエリを `cts:search` 操作で使用します。

位置データがデータベース内に存在するかどうかに関係なく、cts:contains 操作で位置クエリも使用できます。

13.4.2 位置情報の cts:query コンストラクタ

次の位置情報の cts:query コンストラクタを使用できます。

- cts:element-attribute-pair-geospatial-query
- cts:element-pair-geospatial-query
- cts:element-geospatial-query
- cts:element-child-geospatial-query
- cts:json-property-child-geospatial-query
- cts:json-property-geospatial-query
- cts:json-property-pair-geospatial-query
- cts:path-geospatial-query

13.4.3 領域の位置情報値コンストラクタ

次の API を使用して、位置情報の領域を作成できます。これらの関数を上記の位置情報の cts:query コンストラクタとともに使用して、cts:queries を作成します。

- cts:box
- cts:circle
- cts:complex-polygon
- cts:linestring
- cts:point
- cts:polygon

これらの関数の詳細については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。これらの関数は、同名の型コンストラクタを補完するものです。詳細については、「位置クエリ用の XQuery プリミティブ型とコンストラクタ」（349 ページ）を参照してください。

13.4.4 位置情報の形式変換関数

Metacarta、GML、KML、および GeoRSS の形式を cts:box、cts:circle、cts:point、および cts:polygon の形式に変換する XQuery のライブラリモジュールが用意されています。各ライブラリの関数は各形式の位置データから、cts:region プリミティブ型を作成するように設計されています。このプリミティブ型が位置情報の cts:query コンストラクタに渡されて、適切なクエリが作成されます。各関数のシグネチャについては、『MarkLogic XQuery and XSLT Function Reference』の XQuery ライブラリモジュールに関するセクションを参照してください。

13.4.5 位置情報の操作

次の API を使用して、位置データに対するさまざまな操作や計算を実行します。

- `cts:box-intersects`
- `cts:circle-intersects`
- `cts:polygon-intersects`
- `cts:complex-polygon-intersects`
- `cts:polygon-contains`
- `cts:complex-polygon-contains`
- `cts:distance`
- `cts:shortest-distance`
- `cts:destination`

それぞれのシグネチャとこれらの関数の詳細については、『MarkLogic XQuery and XSLT Function Reference』の XQuery ライブラリモジュールに関するセクションを参照してください。

13.5 シンプルな位置検索の例

このセクションでは、位置クエリを使用する `cts:search` の例を示します。

次のような形式になっている URI が `/geo/zip_labels.xml` のドキュメントを想定します。

```
<labels>
  <label id="1">96044</label>
  ...
  <label id="589">95616</label>
  <label id="712">95616</label>
  <label id="715">95616</label>
  ...
</labels>
```

URI が /geo/zip.xml のドキュメントには、次のような形式で多角形データがあると想定します。

```
<polygon id="712">
       0.383337584506173E+02,       -0.121659014798844E+03
       0.383133840000000E+02,       -0.121656011000000E+03
       0.383135090000000E+02,       -0.121666647000000E+03
       0.383135090000000E+02,       -0.121666647000000E+03
       0.383135120000000E+02,       -0.121666875000000E+03
       0.383349030000000E+02,       -0.121667035000000E+03
       0.383353510000000E+02,       -0.121657355000000E+03
       0.383496550000000E+02,       -0.121656811000000E+03
       0.383495590000000E+02,       -0.121646955000000E+03
       0.383494950000000E+02,       -0.121645323000000E+03
       0.383473190000000E+02,       -0.121645691000000E+03
       0.383370790000000E+02,       -0.121650187000000E+03
       0.383133840000000E+02,       -0.121656011000000E+03
</polygon>
```

この場合、cts:polygon コンストラクタを使用して、多角形要素のコンテンツで cts:polygon をキャスト（型抜き）できます。例えば、次の例は上記データの cts:polygon を返します。

```
cts:polygon(fn:data(fn:doc("/geo/zip.xml")//polygon[@id eq "712"]))
```

さらに、次のような形式のコンテンツを想定します。

```
<feature id="1703188" class="School">
  <name>Ralph Waldo Emerson Junior High School</name>
  <state id="06">CA</state>
  <county id="113">Yolo</county>
  <lat dms="383306N">38.5515731</lat>
  <long dms="1214639W">-121.7774624</long>
  <elevation>17</elevation>
  <map>Merritt</map>
</feature>
```

今度は、次のような XQuery について考えます。

```
let $searchterms := ("school", "junior")
let $zip := "95616"
let $ziplabel :=
fn:doc("/geo/zip_labels.xml")//label[contains(.,$zip)]
let $polygons :=
   for $p in fn:doc("/geo/zip.xml")//polygon[@id=$ziplabel/@id]
   return cts:polygon(fn:data($p))
let $query :=
   cts:and-query((
       for $term in $searchterms return cts:word-query($term),
       cts:element-pair-geospatial-query(xs:QName("feature"),
             xs:QName("lat"), xs:QName("long"), $polygons) ))
return  (
<h2>{fn:concat("Places with the term '",
               fn:string-join($searchterms, "' and the term '"),
               "' in the zipcode ", fn:data($zip), ":")}</h2>,
  <ol>{for $feature in cts:search(//feature, $query)
  order by $feature/name
  return (
  <li><h3>{fn:data($feature/name)," "}
  <code>({fn:data($feature/lat)},{fn:data($feature/long)})</code></h3>
  <p>{fn:data($feature/@class)} in {fn:data($feature/county)},
  {fn:data($feature/state)} from map {fn:data($feature/map)}</p></li> )
  }</ol> )
```

この例は、次のような結果を返します（ブラウザでの表示例）。

Places with the term 'school' and the term 'junior' in the zipcode 95616:

1. **Emerson Junior High School** (38.5476843,-121.7452392)

   School in Yolo, CA from map Davis

2. **Oliver Wendell Holmes Junior High School** (38.556573,-121.7363502)

   School in Yolo, CA from map Davis

3. **Ralph Waldo Emerson Junior High School** (38.5515731,-121.7774624)

   School in Yolo, CA from map Merritt

14.0 エンティティエンリッチメントによるドキュメントのマークアップ

この章では、MarkLogic サーバーでエンティティエンリッチメントを使用して、人物や場所といったエンティティに XML マークアップを追加して、テキストを囲む方法について説明します。以下のセクションで構成されています。

- エンティティエンリッチメントの概要
- エンティティエンリッチメントパイプライン

14.1 エンティティエンリッチメントの概要

XML では、要素タグを追加してテキストを囲むことができます。要素タグを追加して特定の意味を持つテキストを囲んでおけば、このタグを検索することで必要なものを発見できます。通常、要素タグでマークアップする項目は人物や場所です。しかし、マークアップが便利な項目は他にもたくさんあります。多くの業界では、マークアップすることに意味があるようなその分野固有の項目があります。例えば、医療研究者にとっては、RX のようなタグで処方薬をマークアップできると便利でしょう。マークアップされる項目は、「*エンティティ*」と呼ばれます。このようなエンティティを見つけて、意味のある（かつ検索可能な）タグでマークアップすることで XML をさらに有用なものにするプロセスは、「*エンティティエンリッチメント*」と呼ばれています。

MarkLogic サーバーは、サードパーティのエンティティエンリッチメントサービスと統合できます。Content Processing Framework では、サードパーティツールを簡単に呼び出せます。「サードパーティ技術を使用するサンプルパイプライン」（366 ページ）で説明しているように、このプロセスを紹介するためのサンプルがいくつか用意されています。

14.2 エンティティエンリッチメントパイプライン

MarkLogic サーバーには、XML に対してエンティティエンリッチメントができる CPF（Content Processing Framework）アプリケーションがあります。サードパーティのエンティティ抽出技術用の CPF アプリケーションを使用することも、独自技術やその他のサードパーティ技術でカスタムアプリケーションを作成することもできます。このセクションは、以下のように構成されています。

- サードパーティ技術を使用するサンプルパイプライン
- カスタムエンティティエンリッチメントパイプライン

CPF アプリケーションを使用するには、Content Processing をデータベースにインストールする必要があります。ドメインやパイプラインについての情報を含む、CPF の詳細については、『Content Processing Framework Guide』を参照してください。

14.2.1 サードパーティ技術を使用するサンプルパイプライン

サードパーティのエンティティエンリッチメントツールに接続できるサンプルパイプラインと CPF アプリケーションが用意されています。サンプルパイプラインは、*<marklogic-dir>*/Installer/samples ディレクトリにインストールされます。次のエンティティエンリッチメントツールに対応するサンプルパイプラインが用意されています。

- TEMIS Luxid®
- Calais OpenCalais
- SRA NetOwl
- Janya
- Data Harmony

MarkLogic サーバーは、これらのツールに web サービス経由で接続します。サンプルコードは、「アズイズ」（そのまま）で提供されています。このサンプルコードは実稼働アプリケーション用ではなく、サポートの対象外です。セットアップの方法などの詳細については、*<marklogic-dir>*/Installer/samples ディレクトリ内にある README.txt ファイルと samples-license.txt ファイルを参照してください。

14.2.2 カスタムエンティティエンリッチメントパイプライン

カスタム CPF アプリケーションを作成すると、他のサードパーティのエンリッチメントアプリケーションを使用してドキュメントをエンリッチできます。カスタム CPF アプリケーションを作成するには、サードパーティアプリケーションや、そのアプリケーションへの接続手段（例えば、web サービス経由）が必要になります。また、前のセクションで説明したサンプルアプリケーションと同様に、XQuery コードとパイプラインファイルを記述する必要があります。

15.0 アラートアプリケーションの作成

この章では、MarkLogic サーバーでアラートアプリケーションを作成する方法について説明するとともに、アラートアプリケーションのコンポーネントについても説明します。この章は、次のセクションから構成されます。

- MarkLogic サーバーのアラートアプリケーションの概要
- cts:reverse-query コンストラクタ
- cts:query コンストラクタの XML シリアライズ
- アラートアプリケーションのセキュリティ上の考慮事項
- リバースクエリのインデックス
- アラート API
- サンプルアラートアプリケーション

15.1 MarkLogic サーバーのアラートアプリケーションの概要

*アラートアプリケーション*は、新たに利用可能になった（入力された）コンテンツが、事前定義（通常は格納）されているクエリとマッチした場合にユーザーに通知するものです。MarkLogic サーバーに含まれる複数のインフラストラクチャコンポーネントを使用することで、柔軟性に優れ、クエリが大量に格納されていてもパフォーマンスと拡張性を実現できるアラートアプリケーションを作成できます。

アラート API を使用したアラートアプリケーションのサンプルが、オープンソースプロジェクトとして、github（https://github.com/marklogic/alerting）に公開されています。このサンプルアプリケーションには、さまざまなエンタープライズ仕様のアラートアプリケーションで必要とされる、詳細なコンポーネントがすべて含まれています。ただし、このサンプルアプリケーションのパッケージのユーザーインターフェイスは、アラートアプリケーションの機能を紹介するためだけのものです。このため、実際のアプリケーションでは、多くの場合、まったく異なるより高性能なユーザーインターフェイスになるはずです。また、サンプルアプリケーションはデモンストレーションのみを目的としており、実稼働での使用は想定されていません。詳細については、`samples-license.txt` ファイルを参照してください。アラートアプリケーションの詳細なコンポーネントに興味がない場合は、この章のアラート API およびサンプルアプリケーションについてのセクション、「アラート API」（ 371 ページ）および「サンプルアラートアプリケーション」（ 378 ページ）に進んでもかまいません。

アラートアプリケーションのコンポーネントの核となるのは、リバースクエリを作成する機能です。リバースクエリ（cts:reverse-query）は、検索でクエリが実行された場合、リバースクエリに提供されたノードが、そのクエリとマッチした場合に true を返す cts:query です。cts:reverse-query の詳細については、「cts:reverse-query コンストラクタ」（368 ページ）を参照してください。

アラートアプリケーションは、リバースクエリ以外にもいくつかのコンポーネントを使用します。例えば、シリアライズされた cts:query コンストラクタ、リバースクエリのインデックス、MarkLogic サーバーセキュリティコンポーネント、アラート API、Content Processing Framework のドメインおよびパイプライン、トリガーなどです。

注： アラートアプリケーションには、有効なアラートライセンスキーが必要です。このライセンスキーは、リバースインデックスやアラート API を使用する場合に必要です。

15.2 cts:reverse-query コンストラクタ

cts:reverse-query コンストラクタは、cts:query 式で使用します。入力とマッチする cts:query ノードに対して true を返します。例えば、次のコードについて考えてみましょう。

```
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 :)
```

$query の cts:query は $node とマッチするので、このクエリは true を返します。コンセプト的には、cts:reverse-query コンストラクタは、他の cts:query コンストラクタとは逆です。具体的には、他の cts:query コンストラクタはドキュメントをクエリにマッチするのに対して、cts:reverse-query コンストラクタはクエリをドキュメントにマッチします。この機能がアラートアプリケーションの核となるのは、実行された場合、特定のノードとマッチするクエリをすべて返す検索を効率的に実行できるためです。

cts:reverse-query コンストラクタは、完全にコンポーザンブルです。つまり他の cts:query コンストラクタと同様に、cts:reverse-query コンストラクタを他のコンストラクタと組み合わせることができます。アラート API は、必要なリバースクエリを生成するときに、開発者に対して cts:reverse-query コンストラクタを抽象化します。cts:query コンストラクタの機能の詳細については、「cts:query 式の構成」（239 ページ）を参照してください。

15.3 cts:query コンストラクタの XML シリアライズ

cts:query 式は、検索対象の指定に使用されます。cts:query 式は、条件次第でシンプルにも複雑にもできます。cts:query 式を格納するため、MarkLogic サーバーには cts:query の XML 表現が用意されています。アラートアプリケーションは、シリアライズされた XML 表現や cts:query 式を格納して、これにリバースインデックスを設定します。この仕組みによって、「このドキュメントとマッチするクエリ」という条件の検索に対して、幅広い回答が短時間で返されます。cts:query の XML 表現をデータベースに格納することが、アラートアプリケーションのコンポーネントの 1 つです。アラート API は、開発者から XML シリアライズを抽象化します。cts:query の XML シリアライズの詳細については、「cts:query 式の構成」（ 239 ページ）章の「cts:query コンストラクタの XML シリアライゼーション」セクションを参照してください。

15.4 アラートアプリケーションのセキュリティ上の考慮事項

アラートアプリケーションでは通常、個々のユーザーが自分専用のアラート基準を作成できます。このためアラートアプリケーションには、これに起因するセキュリティ要件がいくつかあります。例えば、特定のユーザーのアラート基準が満たされた場合、すべてのユーザーに対してではなく、その特定のユーザーに対してのみアラートを通知したいはずです。このセクションでは、セキュリティ上の考慮事項について説明します。このセクションは、次のパートから構成されます。

- アラートユーザー、アラート管理者、アクセス制御
- 事前定義されたアラートアプリケーションのロール

15.4.1 アラートユーザー、アラート管理者、アクセス制御

アラートアプリケーションを管理する必要があるのと同時に、データベースに対する何らかのアクションの実行をアラートアプリケーションのユーザーに許可する必要があります。このためアラートアプリケーションでは、セキュリティの管理が求められます。場合によっては、アラートアプリケーションのユーザーが、実行の権限を与えられていないクエリを実行する必要があります。例えばユーザーが、ある統制された方法で設定情報を確認する必要があるような場合です。これに対応するには、アラートアプリケーションで amp を使用して、アラートアプリケーションに*限って*必要な権限を与えることで、権限のなかった操作の実行をユーザーに許可できます。amp と MarkLogic サーバーセキュリティモデルの詳細については、『Understanding and Using Security Guide』を参照してください。

アラート API は、ビルトインのロールである alert-admin および alert-user だけでなく、複雑なセキュリティロジックをすべて抽象化するので、手動で管理しなくてもセキュリティを適切に処理するアプリケーションを作成できます。

15.4.2 事前定義されたアラートアプリケーションのロール

アラート API で構築されたアラートアプリケーション使用の事前定義された 2 種類のロールに加えて、アラート API が内部で使用するロールもいくつか用意されています。

- Alert-Admin ロール
- Alert-User ロール
- 内部使用限定のロール

15.4.2.1 Alert-Admin ロール

`alert-admin` ロールは、アラートアプリケーションの管理者に対して、アラート API で設定（アラート設定）を作成するために必要なすべての権限を付与するものです。このロールには、ルールが設定されている任意のユーザーとしてコードを実行する権限など、さまざまな権限が含まれています。そのため、`alert-admin` ロールは、信頼できるユーザー（悪意を持っておらず、適切なトレーニングを受けており、正しい管理手続きに従うことが想定されるユーザー）だけに付与してください。`alert-admin` ロールは、アラートアプリケーションの管理者に割り当てます。

15.4.2.2 Alert-User ロール

`alert-user` ロールは、最低限の権限を持つロールです。このロールでは、通常のアラートユーザーが（つまり `alert-admin` ユーザーではない人）、アラート API でコードを実行できます。このようなコードでは、アラートアプリケーションで使用されるドキュメント（設定ファイルやルールなど）の読み取りや更新が必要な場合があります。このロールは、アラート API においてアラートアプリケーションのユーザーに、必要最低限のアクセス権だけを付与するものです。

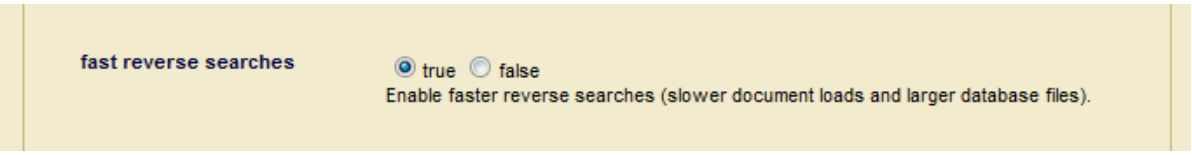
`alert-user` ロールには、アラート API で使用される権限だけが含まれています。アラート API のスコープ外の関数に対する実行権限はありません。アラート API では、上位の権限が必要とされる操作の管理を強化するメカニズムとして、`alert-user` ロールを使用しています。そのため、アラートアプリケーションを使用してもかまわないと考えられるユーザーに、このロールを割り当ててもそれほど危険ではありません。

15.4.2.3 内部使用限定のロール

上記以外にも、アラート API で使用される `alert-internal` と `alert-execution` という 2 種類のロールが存在します。これらのロールは、どのユーザーやロールにも明示的に付与しないでください。これらのロールは、アラート API に含まれる特定の関数のコンテキストの範囲内で、特別な権限を強化する目的で使用されます。これらのロールをユーザーに付与すると、本来、付与すべきではないシステム上の権限が付与されることになります。したがって、これらのロールは、どのユーザーにも付与しないでください。

15.5 リバースクエリのインデックス

リバースクエリインデックスを有効または無効にするには、データベース設定で、`fast reverse searches` インデックス設定を `true` に設定します。



`fast reverse searches` インデックスは、`cts:reverse-query` を使用する検索を高速化します。アラートアプリケーションが大量のルールを扱えるように拡張するには、fast reverse searches を有効にする必要があります。

> **注：** アラートアプリケーションには、有効なアラートライセンスキーが必要です。このライセンスキーは、リバースインデックスやアラート API を使用する場合に必要です。

15.6 アラート API

アラート API は、堅牢なアラートアプリケーションの構築を支援する目的で設計されています。この API は、アプリケーションのセキュリティに詳細に対応するとともに、アラートアプリケーションのあらゆるコンポーネントを設定するメカニズムを提供します。これはトリガーや CPF を使用して、ドキュメントの状態に応じてアラートを発生しやすくすることを目的としています。このセクションでは、アラート API について説明します。このセクションは、次のパートから構成されます。

- アラート API の概念
- アラート API の使用
- CPF とアラートアプリケーションの使用

アラート API は、XQuery ライブラリモジュールとして実装されています。個々の関数のシグネチャと説明については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

15.6.1 アラート API の概念

アラート API を使用する際には、次の 3 つの主要な概念について理解しておく必要があります。

- アラート設定
- アラート発行時に実行するアクション

- アラート発行のルール

15.6.1.1 アラート設定

アラート*設定*は、アラートアプリケーションに対するアラート設定の XML 表現です。通常、アラートアプリケーションに必要なアラート設定は 1 つだけですが、必要に応じて複数のアラート設定を用意できます。アラート API で、アラート設定の XML 表現を定義します。この XML 表現は、`alert:make-config` 関数から返されます。この設定は、`alert:config-insert` 関数を使用してデータベースに保存します。アラート API には、アラート設定を操作する setter 関数と getter 関数も用意されています。アラート設定は、アラートアプリケーションの管理者が作成および更新すると想定されています。したがって、アラート設定を操作するユーザーには、`alert-admin` ロールが必要です。

15.6.1.2 アラート発行時に実行するアクション

アクション（*action*）は、アラートが発生したときに実行される XQuery コードです。アクションとしては、例えば、データベース内のドキュメントの更新やメールの送信など、アプリケーションに適切な処理を行います。アクションは XQuery のメインモジュールであり、アラート API でアクションの XML 表現を定義します。この XML 表現は、`alert:make-action` 関数から返されます。アクションの XML 表現は、そのアクションを実行する XQuery のメインモジュールを参照します。このアラートアクションの XML 表現については、`alert:action-insert` 関数を使用してデータベースに保存します。アラート API には、アラートアクションを操作する setter 関数と getter 関数も用意されています。アラートアクションは、アラートアプリケーションの管理者が作成および更新すると想定されています。したがって、アラートアクションを操作するユーザーには、`alert-admin` ロールが必要です。

アラートアクションは、`alert:invoke-matching-actions` または `alert:spawn-matching-actions` によって起動および生成されます。アクションには、次の外部変数を指定できます。

```
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;
```

これらの外部変数は、必要に応じてアクションで使用できます。変数を使用するには、起動または生成するアクションモジュールのプロローグに、上記の変数宣言を含める必要があります。

15.6.1.3 アラート発行のルール

*ルール*はユーザーへのアラート通知の基準であり、その基準を満たした場合に実行するアクションへの参照と一緒に使用します。例えば、`jelly beans` という検索タームとマッチする新しいコンテンツや変更されたコンテンツに関心がある場合、`jelly beans` というタームを含むドキュメントの新規作成や変更が発生した時点でアラートを発行するというルールを定義できます。このルールは、次の `cts:query` のように表されます

```
cts:word-query("jelly beans")
```

ルールには、ドキュメントがクエリとマッチした場合に実行するアクションも関連付けます。アラートアプリケーションは、高速で拡張性に優れたパフォーマンスで多数のルールをサポートするように設計されています。また、各ルールの作業量は、それぞれのアクションによって異なります。例えば、アプリケーションに、ルールと一致したときにメールを送信するというアクションがあるとします。一致するルールが大量に存在する場合は、そのすべてのメールを送信する影響を考慮する必要があります。

アラート API で、ルールの XML 表現を定義します。この XML 表現は、`alert:make-rule` 関数から返されます。このルールは、`alert:rule-insert` 関数を使用してデータベースに保存します。ルールは、アラートアプリケーションの通常のユーザーが作成および更新すると想定されています。アラート API には、アラートルールを操作する setter 関数と getter 関数も用意されています。ルールを作成する通常のユーザーは、特定のタスク（特定のドキュメントの読み取りや更新）の実行に必要な権限やパーミッションが必要なので、ルールの挿入には最低限の権限が必要です。このため、アラートアプリケーションでルールを作成するユーザーには、最低限の権限を持つ `alert-user` ロールが必要です。

15.6.2 アラート API の使用

前のセクションで説明した概念について理解したら、実際にアラート API を使用するのは簡単です。このセクションでは、次のようなアラート API 使用時の詳細について説明します。

- 設定の作成（alert-admin ロールのユーザー）
- アクションの作成（alert-admin ロールのユーザー）
- ルールの作成（alert-user ロールのユーザー）
- コンテンツに対するルールの実行

15.6.2.1 設定の作成（alert-admin ロールのユーザー）

アラート API を使用する最初のステップは、アラート設定の作成です。アラート設定の詳細については、「アラート設定」（372 ページ）を参照してください。アラート設定は、アラートアプリケーション管理者（`alert-admin` ロールまたは `admin` ロールのユーザー）が作成する必要があります。次のサンプルコードでは、アラート設定を作成する方法を示します。

```
(: 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)
```

15.6.2.2 アクションの作成（alert-admin ロールのユーザー）

アラートアプリケーション管理者は、アラートの発生時に実行するアクションも作成する必要があります。アクションは XQuery のメインモジュールであり、条件次第でシンプルにも複雑にもなります。アラートアクションの詳細については、「アラート発行時に実行するアクション」（372 ページ）を参照してください。

実際にアクションを作成するには、アラートアプリケーションで実行しようとする処理の内容について、よく理解しておく必要があります。次に示すのは、`ErrorLog.txt` ファイルにログメッセージを送信する極めてシンプルなアクションです。このアクションを作成するには、次のコンテンツを含む XQuery ドキュメントをアプリケーションサーバーのルートに作成します。

```
xdmp:log(fn:concat(xdmp:get-current-user(), " was alerted"))
```

例えば、アラートアプリケーションを実行するアプリケーションサーバーのルートが `/space/alert` である場合は、この XQuery ドキュメントを `/space/alert/log.xqy` に配置します。

アラートアクションの他の例については、XQuery ファイル `<marklogic-dir>/Modules/alert/log.xqy` を参照してください。アラートアクションでは、XQuery で記述できる任意のアクションを実行できます。

次の例のように、アラート API でアクションを設定する必要もあります。この操作は alert-admin ロールのユーザーが実行します。

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

let $action := alert:make-action(
    "xdmp:log",
    "log to ErrorLog.txt",
    xdmp:modules-database(),
    xdmp:modules-root(),
    "/alert-action.xqy",
    <alert:options>put anything here</alert:options> )
return
alert:action-insert("my-alert-config-uri", $action)
```

15.6.2.3 ルールの作成（alert-user ロールのユーザー）

アラートアプリケーションを設定することで、アプリケーションの通常のユーザーがルールを作成できます。例えば、ユーザーによるルールの作成を支援するため、フォームを用意できます。次に示すのは、ルールの作成に必要な基本的なコードです。このコードにはユーザーインターフェイスが含まれていないので、実際のコードはおそらくもっと複雑になるはずです。このコードは、上記で説明したアクションを実行するルールを作成して、このルールに cts:query を関連付けているだけです。このコードは、alert-user ロールのユーザーが実行します。

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

let $rule := alert:make-rule(
    "simple",
    "hello world rule",
    0, (: equivalent to xdmp:user(xdmp:get-current-user()) :)
    cts:word-query("hello world"),
    "xdmp:log",
    <alert:options/> )
return
alert:rule-insert("my-alert-config-uri", $rule)
```

注： データベース内のドキュメントの読み取りや作成など、権限が必要な操作をアクションとして実行する場合は、アプリケーションを実行するユーザーに対して、適切な実行権限と URI 権限を追加する必要があります。

15.6.2.4 コンテンツに対するルールの実行

アプリケーションでアラートを発行（つまり、ルールによるアクションを実行）するには、何らかのコンテンツに対してルールを実行する必要があります。これには、アラート API でトリガーを設定する方法（alert:create-triggers）、CPF とアラートパイプラインを使用する方法、専用のメカニズムを作成してコンテンツに対してルールを実行する方法など、いくつかの方法があります。

ルールを手動で実行するには、alert:spawn-matching-actions または alert:invoke-matching-actions API を使用します。これらは、アプリケーションとしても、ルールを簡単にテストする方法としても、コンテキストに関係なく、アラートの実行時に有用です。alert:spawn-matching-actions は、タスクサーバーに対して多数のアクションを生成して非同期的に実行します。したがって、一度に多数のアラートが発行される可能性がある場合に適しています。alert:invoke-matching-actions API は、アクションを即座に実行します。したがって、多数のアクションが発生する可能性がある場合は、すべて同じコンテキストで実行されるので、使用に注意が必要です。これらの API は、任意のユーザーが実行できます。アクションが生成されるかどうかは、各ルールの所有者が持つ表示パーミッションによって異なります。次に示すのは、以前に作成したアラートを発行するだけのシンプルな例です。

```
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/>)
```

前のセクションの説明に従って、設定、アクション、およびルールを作成した場合、alert-user ロールが割り当てられ、ルールを作成した some-user という名前のユーザーとしてコードを実行すると、次の情報がログとして ErrorLog.txt ファイルに記録されます。

```
some-user was alerted
```

注： アラートの数が多く、ルールのアクションでリソースが大量に消費される場合、アクションの呼び出しや生成のときに、システムで大量のクエリアクティビティが発生する可能性があります。これこそがアラートアプリケーションの目的なので、このような処理自体は問題ありません。ただし、処理に対応できるリソースを確保しておく必要があります。

15.6.3 CPF とアラートアプリケーションの使用

アラート API を使用して構築したアラートアプリケーションは、CPF（Content Processing Framework）との使用に適しています。CPF はドキュメントの状態を維持するように設計されているので、CPF を使用すれば簡単に、特定のスコープのドキュメントが作成または更新された時点をトラッキングして、そのドキュメントに対して何らかのアクションを実行できます。アラートアプリケーションの場合、このアクションとしては、変更されたドキュメントに対するリバースクエリを実行し、マッチしたルールに応じてアラートを発行する処理などがあります。アラート API は、開発者からリバースクエリを抽象化します。

アラートアプリケーションでの CPF の使用をシンプルにするため、アラート用に事前構築されたパイプラインが用意されています。このパイプラインは、アラート API で構築されたアラートアプリケーションで使用するように設計されています。このアラート CPF アプリケーションは、アラートパイプラインがアタッチされている CPF ドメインのスコープ内で新規作成および変更されたすべてのコンテンツに対してアラートを実行します。

MarkLogic サーバーに含まれている他のパイプライン（変換パイプラインやモジュラードキュメントパイプラインなど）とともにアラートパイプラインを使用する場合、アラートパイプラインは、他のパイプラインの処理がすべて完了してから実行されるように優先度が定義されています。この方法では、パイプライン処理を経たコンテンツの最終的なビューに対してアラートが発生します。カスタムパイプラインをアラートパイプラインとともに使用する場合、想定どおりの順序でアラートが発生するように、カスタムパイプラインに優先度を追加したほうがよいでしょう。

アラートを使用する CPF アプリケーションを設定するには、次の手順に従います。

1. 「リバースクエリのインデックス」（371 ページ）の説明に従って、データベースのリバースインデックスを有効にします。

2. 「設定の作成（alert-admin ロールのユーザー）」（374 ページ）および「アクションの作成（alert-admin ロールのユーザー）」（374 ページ）の説明に従って、`alert-admin` ロールのユーザーがアラート設定とアラートアクションを作成します。

3. 「ルールの作成（alert-user ロールのユーザー）」（375 ページ）の説明に従って、`alert-user` ロールのユーザーがルールを定義するようにアプリケーションを設定します。

4. Content Processing がまだインストールされていない場合は、データベースにインストールします（［Databases］＞［*database_name*］＞［Content Processing］＞［Install］タブ）。

5. ドメインの `domain scope` を設定します。

6. アラートパイプラインと Status Change Handling パイプラインをドメインにアタッチします。必要に応じて、他のパイプライン（さまざまな変換パイプラインなど）もドメインに追加できます。

7. 使用するアラート設定をアラートアクション側で決定できるように、`alert:config-set-cpf-domain-names` 関数または `alert:config-set-cpf-domain-ids` 関数を使用して、ドメインのアラート設定を通知します。

   例えば、CPF ドメインの名前が「Default Documents」である場合、次のようにできます。

```
alert:config-insert(
   alert:config-set-cpf-domain-names(
      alert:config-get($config-uri),
      ("Default Documents")))
```

   注： アラート設定は、複数の CPF ドメインで使用できます。この場合、複数のドメイン名または ID のシーケンスを設定します。

ドメインのスコープ内でコンテンツが新規作成または更新されると、マッチしたすべてのルールに対応するアクションが開始されます。タスクサーバーに対して生成されるアラート数が多い場合、タスクサーバーをマシンに対して適切に設定してください。例えば、16 コアのマシンを使用する場合、タスクサーバーの `threads` 設定をデフォルトの 4 よりも大きい値に設定できます。実際の `threads` 設定は、マシンで実行されている他の処理によって異なります。

CPF の詳細については、『Content Processing Framework Guide』を参照してください。

15.7 サンプルアラートアプリケーション

サンプルアラートアプリケーションが developer.marklogic.com/code で公開されています。このサンプルアプリケーションはアラート API を使用しており、さまざまなエンタープライズ仕様のアラートアプリケーションで必要とされる、詳細なコンポーネントがすべて含まれています。ただし、このサンプルアプリケーションのパッケージのユーザーインターフェイスは、アラートアプリケーションの機能を紹介するためだけのものです。このため、実際のアプリケーションでは、多くの場合、まったく異なるより高性能なユーザーインターフェイスになるはずです。このサンプルコードは、「アズイズ」（そのまま）で提供されています。このサンプルコードは実稼動アプリケーション用ではなく、サポートの対象外です。セットアップの方法などの詳細については、developer.marklogic.com/code のコードを参照してください。

16.0 fn:count と xdmp:estimate の使い方の比較

この章では、fn:count 関数と xdmp:estimate 関数の違いをいくつか説明します。この章は、次のセクションから構成されます。

- fn:count は正確、xdmp:estimate は高速
- xdmp:estimate ビルトイン関数
- 検索サイズの推定のための cts:remainder の使い方
- xdmp:estimate を使用する場合

16.1 fn:count は正確、xdmp:estimate は高速

XQuery 言語では、fn:count 関数により、シーケンス内の項目を数えるための一般的なサポートを提供しています。ただし、fn:count は汎用的なため、最適化が困難です。数えるシーケンスとしては、データベースに格納されているシーケンス、動的に構築されるシーケンス、取得または構築後にフィルタリングされるシーケンスなどのシーケンスを任意に複雑に組み合わせることができます。ほとんどの場合、MarkLogic サーバーでは、シーケンスを数えるためにシーケンスを処理する必要があります。このため、パフォーマンスに影響を与えるほどの著しい I/O 要求が生じることがあります。

MarkLogic サーバーでは、fn:count を概算する効率的な方法として、xdmp:estimate XQuery ビルトイン関数が提供されています。データを直接検査することによって答えを頻繁に処理する必要がある fn:count（したがって I/O 読み込みの負荷が高くなります）とは異なり、xdmp:estimate ではインデックスから答えを直接計算します。状況によっては、インデックスによって生成される値は fn:count によって返される値と同一になります。また別の状況では、これらの値は、指定されたシーケンスとデータに応じてさまざまな程度で異なります。xdmp:estimate は、高速な推定値を返すことができない場合、エラーをスローします。したがって、fn:count の正確性と同様に、xdmp:estimate の高速性は信頼できます。

xdmp:estimate によって開発者は、インデックスを使って数えることを最適化できます。

16.2 xdmp:estimate ビルトイン関数

xdmp:estimate では、検索可能な XPath 式をパラメータとして利用でき、シーケンス内の項目の個数の概算値が返されます。

```
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 は、必ずしも fn:count と同じ値を返すわけではありません。fn:count 関数は、パラメータとして提供されたシーケンス内の項目の正確な数を返します。対照的に、xdmp:estimate は次の規則に基づいて答えを提供します。

1. xdmp:estimate に渡されたパラメータが検索可能な XPath 式である場合、xdmp:estimate では、事後フィルタリングのためにデータベースから選択されるフラグメントの個数が返されます。この数値は、非常に高いパフォーマンスでインデックスから直接計算されます。ただし、(a) 単一のフラグメント内にマッチする項目が複数存在する場合、または (b) インデックスによって暫定的に選択されたフラグメントで、実際にはマッチ項目を含んでいないものが存在する場合のいずれかで、指定されたシーケンスの実際の fn:count の値とは異なることがあります。

2. xdmp:estimate に渡されたパラメータが検索可能な XPath 式ではない場合（つまり、doc、collection()、または input() 関数、あるいは / ステップまたは // ステップにルートを持つ XPath ではない場合）、xdmp:estimate はエラーをスローします。

xdmp:estimate は、こうした方法によって fn:count 関数と明確に異なるように定義されています。xdmp:estimate は常に、高速に実行されます。fn:count は常に、「正しい」答えを返します。今後 MarkLogic がサーバーの最適化機能を向上させるにつれ、fn:count が正確かつ高速性となる場合も増えてくるでしょう。ただし当面は、どのアプローチを採用するかは開発者の判断に任せられます。

16.3 検索サイズの推定のための cts:remainder の使い方

同一のクエリステートメントにおいて、検索サイズの推定値を取得し、かつ検索を実際に実行する必要がある場合、検索サイズの推定のために cts:remainder 関数を使用できます。cts:remainder による検索は、xdmp:estimate よりも効率的です。推定値のみが必要で、実際の検索結果が必要ではない場合、xdmp:estimate がより効率的です。

cts:remainder は、検索結果セットの特定ノードからの残りのノードの数を返します。最初のノードで実行した場合、検索時の xdmp:estimate と同じ結果が返されます。また、cts:remainder には、検索での任意の項目から始まる検索の推定結果数（例えば、500 番目の検索項目の後に残る結果の数など）を返す柔軟性もあり、これを効率的に行います。

xdmp:estimate と同様、cts:remainder は、フィルタリングされていない結果に基づいて、概算の結果数を求めるためにインデックスを使用します。フィルタリングされていない結果については、『Query Performance and Tuning Guide』の「Using Unfiltered Searches for Fast Pagination」を参照してください。cts:remainder のシンタックスと例については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

16.4 xdmp:estimate を使用する場合

MarkLogic サーバーでは、指定された XPath と一致するコンストラクトを含んでいる可能性がある XML フラグメントの識別を概算するためにインデックスを使用します。次に、このフラグメントの集まりは、さらに処理するために必要なノードを正確に把握する目的でフィルタリングされます。

検索可能な XPath 式の場合、`xdmp:estimate` は、前述の最初の概算ステップで選択されたフラグメントの数を返します。この演算は、インデックスから直接実行されるので、ほとんど瞬時に完了します。ただし、この概算値が `fn:count` によって返される結果と一致しない場合が 2 つ考えられます。

1. フラグメントに、指定された XPath に対して複数のマッチ項目が含まれている場合、`xdmp:estimate` ではこれらの項目すべてをまとめて 1 つとして数える（少なく数えてしまう）のに対して、`fn:count` では個別の項目として数えます。
2. さらに、多く数えてしまうこともありえます。インデックスの最適化では、マッチ項目の見落としがないようにするために、過剰に選択してしまう場合があります。一般的なクエリ処理においては、余分に選択されたこれらのフラグメントは、第 2 ステージのフィルタリングプロセスで破棄されます。ただし、`xdmp:estimate` ではこれらのフラグメントはマッチ項目として数えます（`fn:count` では除外されます）。

次のサンプルクエリについて考えてみましょう。前述の最適化アルゴリズムの最初のステップは、クエリの後の `xdmp:query-trace` の出力に示されています。

クエリ：

```
/MedlineCitationSet/MedlineCitation//Author[LastName="Smith"])
```

クエリトレース出力：

```
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
```

このシナリオでは、指定された XPath に `fn:count` を適用すると、データベース内に姓が「Smith」の著者が 271 人いることがわかります。`xdmp:estimate` を使用すると、答えとして 263 が生成されます。この例では、データベース内で「Smith」という著者が複数含まれるフラグメントがあり、`xdmp:estimate` ではフラグメントの数のみが数えられるために、少なく数えられています。

こうした状況が特定のデータベースやデータセットで発生する状況について理解するには、オプティマイザに関する深い理解が必要です。サーバーがバージョンアップするたびにオプティマイザも強化されるので、これはなかなか難しいです。

次の 3 つのガイドライン集は、`xdmp:estimate` を使用すべき場合と方法を理解するうえで役立ちます。

- 推定値が十分に有益である場合
- XPath が適切な条件を満たす場合
- 経験的テストによって正確性が示される場合

16.4.1 推定値が十分に有益である場合

状況によっては、推定だけで十分です（正確な値ではなく）。今日の多くの検索エンジンではこのアプローチを使用しています。最初の 20 件の結果を表示しながら、総ヒット（マッチ）数を推定しています。正確な件数が重要でない場合、`xdmp:estimate` の使用が妥当です。

16.4.2 XPath が適切な条件を満たす場合

単なる概算ではなく正確な答えが必要な場合で、パフォーマンス面のメリットから `xdmp:estimate` を使用するときは、シンプルながら留意すべき条件があります。

1. フラグメントルートまたはドキュメントルートのいずれかであるノードを数えると、常に、正確な結果が返されます。

   例：

   ```
   xdmp:estimate(/node-name) is equivalent to count(/node-name)

   xdmp:estimate(//MedlineCitation) is equivalent to count(//MedlineCitation)
   ```

   `MedlineCitation` がフラグメントルートである場合。例えば、この制約は、サンプルの Medline アプリケーションが `http://support.marklogic.com` 上のサンプルコードでどのように設定されているかを示します。

2. 1 つのフラグメントに、述語とマッチする要素が複数含まれていることがある場合、少なく数えてしまう可能性があります。下記のサンプルデータが 1 つのフラグメントに存在する場合を考えます。

   ```
   <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>
   ```

   この場合、`fn:doc()//author[last-name = "Smith"]` を指定する XPath では、*少なく数えます*。つまり、上記のサンプルデータで 2 件マッチしているのに 1 件の項目のみを数えます。

3. XPath に述語が複数含まれている場合、多く数えてしまう可能性があります。上記のサンプルデータを使用した場合、`fn:doc()//author[last-name = "Smith"][first-name = "David"]` を指定する XPath ではマッチがありません。ただし、上記のフラグメントには述語 `[last-name = "Smith"]` および `[first-name = "David"]` を個別に満たす著者要素が含まれているので、事後フィルタリング用に選択されます。この場合、`xdmp:estimate` は上記のフラグメントをマッチと見なし、多く数えてしまいます。

16.4.3 経験的テストによって正確性が示される場合

最後のステップとして、xdmp:estimate によって返される値について理解するために、次の 2 つの手法を使用できます。

1. 開発時に、同じシーケンスを xdmp:estimate と fn:count を使用して数え、実稼働データセットで想定されるすべての構造的バリエーションを示すデータセットで結果が異なるかどうかを確認します。

2. xdmp:query-trace をオンにし、xdmp:estimate で使用する XPath シーケンスを評価し、ログファイル内の query-trace 出力を検査します。この出力により、XPath で検索可能な割合、選択されたフラグメントの数（xdmp:estimate が示す答え）、および最終的なマッチの数（fn:count が示す答え）がわかります。

17.0 ステミング検索の概要と使い方

この章では、MarkLogic サーバーのステミング（語幹処理）検索機能の使用方法について説明します。次のセクションから構成されます。

- MarkLogic サーバーでのステミング
- ステミングをオンにする
- ステミング検索とワード検索の比較
- 語（ワード）がクエリと一致するかどうかを確認するために cts:highlight または cts:contains を使用する
- ワイルドカード検索との関係

17.1 MarkLogic サーバーでのステミング

MarkLogic サーバーでは、英語などの言語でのステミングをサポートしています。ステミングがサポートされている言語のリストについては、「サポートされている言語」（478 ページ）を参照してください。

MarkLogic サーバーでは、データベース設定でステミング検索がオンになっている場合、クエリで指定された文字列そのものだけでなく、クエリで指定された語と同じ語幹に由来する語も自動的に検索されます。語の*ステミング*検索では、まったくそのままのタームに加えて、検索タームと同じ意味、同じ品詞から派生するタームも検索されます。語幹はスペリングに基づきません。例えば、`cardiac` のスペリングが `card` から始まっていても、`card` と `cardiac` は語幹は異なっています。それに対して、`running` と `ran` は、スペリングがまったく異なっていても、語幹は同じ（`run`）です。（前述の `card` と `cardiac` の例のように）部分的なパターンマッチングに基づいて語を検索する場合は、ワイルドカード検索（「ワイルドカード検索の概要と使い方」（405 ページ）を参照）を使用します。

ステミングにより、それ以外の方法では見落とされる非常に関連性の高いマッチを検索で返すことができます。例えば、ターム `run`、`running`、および `ran` は、すべて語幹が同じです。したがって、ターム `run` のステミング検索では、指定されたターム `run` に加えて、ターム `running`、`ran`、および `runs` を含んでいる要素とマッチする結果が返されます。

MarkLogic サーバーでサポートされているステミングでは、異なる品詞間は対象になりません。例えば、`conserve`（動詞）と `conservation`（名詞）は、品詞が異なるために語幹が同じとは見なされません。したがって、ステミング検索をオンにして `conserve` を検索すると、結果に `conserve` と `conserves` を含んでいるドキュメントは含まれますが、`conservation` を含んでいるドキュメントは（`conserve` または `conserves` も出現しない限り）含まれません。

ステミングは言語固有です。つまり、各語は指定された言語であるとして扱われます。言語は、`xml:lang` 属性または他のいくつかの方法によって指定できます。ある言語のタームは、別の言語の同じタームに対するステミング検索ではマッチしません。言語がクエリに与える影響の詳細については、「言語別のドキュメントのクエリ実行」（475 ページ）を参照してください。

17.2 ステミングをオンにする

検索でステミングを使用するには、データベース設定でステミングをオンにする必要があります。MarkLogic サーバーで作成された新しいデータベースではすべて、デフォルトでステミングがオンになっています。ステミング検索では、読み込み時、更新時、または再インデックス付け時に作成されたインデックスが必要です。これまでステミングをオンにしたことがない既存のデータベースでステミングをオンにする場合は、ステミングされた結果を検索で得られるようにするには、データベースを再読み込みするか、再インデックス付けする必要があります。ステミングをオンにする場合は、ソースコンテンツのサイズの 2 倍程度のディスク領域がさらに必要です。

MarkLogic サーバーのステミングでは、基本、高度、複合語の分割の 3 つのタイプが利用可能です。次の表では、Admin 画面のデータベース設定ページで利用可能なステミングオプションについて説明します。

ステミングオプション	説明
OFF（オフ）	ステミング用に語にインデックス付けすることはありません。
Basic（基本）	デフォルトです。各語は、単一の語幹でインデックス付けされます。
Advanced（高度）	各語は、1 つあるいは複数の語幹にインデックス付けされます。単語の中には、2 つ以上の意味を持っているものもあるため、複数の語幹があることもあります。例えば、語 `further` の語幹には、`further`（*he attended the party to further his career* のような場合）および `far`（*she was further along in her studies than he* のような場合）があります。
Decompounding（複合語の分割）	個々のすべての語幹にインデックス付けされます。大きな複合語を構成する小さな構成語にもインデックス付けされます。多くは、複合語を使用するドイツ語のような言語で使用されます。

データベースに対してステミングがオンになっている場合は、さらにクエリレベルでステミングの使用を制御できます。ステミングは、`cts:word-query`、`cts:element-word-query`、`cts:element-value-query` など、どの MarkLogic `cts:query` コンストラクタ関数でも使用できます。`cts:query` コンストラクタへのオプションパラメータには、ステミングオプション「`stemmed`」または「`unstemmed`」を指定できます。これらの関数の詳細については、『MarkLogic XQuery and XSLT Function Reference』（http://docs.marklogic.com）を参照してください。

ワイルドカードを含んでいるクエリタームはステミングされません。ステミングオプションを指定しないままにした場合、ワイルドカードを含んでいないあらゆる語についてステミング検索が実行されます。したがって、データベースでステミングがオンになっている限り、クエリでステミングを明示的にオンにする必要はありません。

データベースでステミングをオフにし、クエリでステミングを明示的に指定した場合、クエリでエラーがスローされます。

17.3 ステミング検索とワード検索の比較

ステミング検索インデックスとワード検索（ステミングなし）インデックスには重複した機能があり、ステミング検索インデックスのみをオンにすることで（つまり、ワード検索インデックスをオフのままにしても）、目的の結果が得られる可能性はかなり高いです。

ステミング検索では、検索する語と、検索する語と同じ語幹の語について、関連性でランク付けされた結果が返されます。したがって、ワード検索の場合と同じ結果に加えて、同じ語幹の語を含んでいる項目についての結果も得られます。ほとんどの検索アプリケーションで、これは望ましい動作です。

ワード検索インデックスもオンにする必要がある唯一の場合は、アプリケーションで完全一致の結果*のみ*を返す（つまり、ステミングに基づいた結果を*返さない*）正確なワード検索を必要とするときです。

さらに、ステミング検索インデックスは、ワード検索（非ステミング）インデックスよりも少ないディスク領域で済みます。したがって、データベース設定でステミング検索オン、ワード検索オフというデフォルト設定を使用すると、ディスク領域を節約し、読み込み時間を削減できます。どのインデックスにおいても、使用ディスク領域および読み込み時間の増大というコストが生じます。アプリケーションにとって追加インデックスの作成コストは意味があるかどうか、また一部のインデックスについて使用しなくても同じ要件を満たすことができるかどうか、アプリケーションの要件に基づいて決定する必要があります。

ステミング検索インデックスのみをオンにしている場合（つまり、データベース設定でワード検索をオフにしている場合）にワード検索（ステミングなし検索）を実行する必要があるときは、「ステミングなし検索」（476 ページ）の説明に従って、ステミング検索を行ってから、ステミングなしの `cts:query` で結果をフィルタリングする必要があります。

17.4 語（ワード）がクエリと一致するかどうかを確認するために cts:highlight または cts:contains を使用する

ステミングでは、同じスペリングではないタームでクエリをマッチできるため、実際にクエリのマッチを引き起こした語を見つけることが難しい場合があります。cts:highlight を使用して、実際にクエリとマッチした語のテストや強調表示を行うことができます。cts:highlight の詳細については、『MarkLogic XQuery and XSLT Function Reference』の「検索タームのマッチ箇所の強調表示」（338 ページ）を参照してください。

cts:contains を使用して、語がクエリとマッチするかどうかをテストすることもできます。cts:contains 関数は、マッチがある場合に true、マッチがない場合に false を返します。例えば、次の関数を使用して、ある語が別の語と同じ語幹を持っているかどうかをテストできます。

```
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")
```

17.5 ワイルドカード検索との関係

ステミング検索とワイルドカード検索の関係については、「他の検索機能との関係」（410 ページ）を参照してください。

18.0 トークン化とステミングのためのカスタム辞書

カスタム辞書は、言語ごとに語のステミングおよびトークン化の方法をカスタマイズするために使用します。この章では、カスタム辞書について説明します。次のセクションを含んでいます。

- MarkLogic サーバーのカスタム辞書
- 辞書とエントリーのスキーマ
- カスタム辞書の関数
- 使用例

18.1 MarkLogic サーバーのカスタム辞書

MarkLogic でサポートおよびライセンスされた任意の言語においてデフォルトのステミングまたはトークン化の動作を変更したい場合、カスタム辞書を作成または修正できます。カスタム辞書のエントリーは、ビルトインのステミングデータよりも優先されます。また、MarkLogic で使用する新しいステミングデータを追加します。例えば、「meeting」がビルトイン辞書内の語幹「meet」にマッピングされるとします。しかし、カレンダで使用される「meeting」を、「meet」とマッチさせたくはありません。この場合、「meeting」を語幹として持っている語「meeting」のエントリーをカスタム辞書に追加することによってこれを防止できます。

カスタム辞書には、次の 3 つの用途があります。

- 日本語（`ja`）、簡体字中国語（`zh`）、および繁体字中国語（`zh_hant`）ではすべて、語を区切るためにスペースを使用しないので、テキストをトークン（語と句読点）に分割する、言語のトークン化機能を使用します。トークン化機能の動作は、言語のカスタム辞書にエントリーを追加することによって変更できます。
- スペースと句読点に基づいてトークン化されるすべての言語ならびに日本語では、辞書によって語の活用形が辞書形（原形）にマッピングされます。例えば、英語の辞書では、「views」、「viewed」、「viewing」はすべて、元の共通の語幹である「view」にマッピングされます。辞書エントリーを変更、削除、または追加して、どの語がどの語幹にマッピングされるかを修正できます。ステミングとトークン化の詳細については、「ステミング検索の概要と使い方」（385 ページ）を参照してください。
- スペリングのバリエーションと専門語彙の取り扱い（「aluminum」、「aluminium」のような語）。辞書のエントリーにより、これらの 2 つのスペリングは、ステミングに基づいて、サーバー内のあらゆる要素について実質的に同じものとして扱われます。

MarkLogic でライセンスおよびサポートされる各言語につき、カスタム辞書は 1 つのみ存在します。言語のカスタム辞書を取得するには、まず使用中のシステムでライセンスされているすべての言語のリストを取得してください。このリストを反復処理することにより、任意のカスタム辞書を取得できます。カスタム辞書はデータディレクトリに格納されるので、MarkLogic サーバーをアップグレードしても削除されません。

カスタム辞書の形式および API は、名前空間を利用しており、ランタイムエラーの回避のため、辞書書き込み中に検証されます。ただし、MarkLogic では、必要とされない限りエラーの原因とならない重複エントリーについては、チェックしません。また、英語のカスタム辞書内では非ラテン文字についてチェックしません（しかしそのような文字を含むエントリーは無効になり、処理されません）。カスタム辞書内のエラーを修正した際、再起動は必要ありません。

注意：カスタム辞書の変更によってドキュメントが影響を受ける場合、再インデックス付けが必要です。辞書を変更する前に影響を受けるドキュメントがある場合はそれを特定するために、変更によって追加または削除される語のワード検索（ステミング検索ではありません）を行ってください。例えば、語幹「view」に「viewer」を追加する一方「viewing」を削除する場合は、「viewer」と「viewing」が辞書の変更によって影響を受けるので、両方の語を検索します。「view」自体は影響を受けません。これらの語を含んでいるドキュメントは、辞書の変更後に再インデックス付けが必要になります。

18.2 辞書とエントリーのスキーマ

カスタム辞書は、XML ドキュメントとして格納され、専用の名前空間と共通のエントリー形式を使用します。次に、単一のエントリーを含むカスタム辞書の例を示します。

```
<dictionary xmlns="http://marklogic.com/xdmp/custom-dictionary"
            xml:lang="en">
  <entry>
    <word>servlets</word>
    <stem>servlet</stem>
    <pos>Nn</pos>
  </entry>
</dictionary>
```

ルートの `<dictionary>` 要素には、辞書に関連付けられている言語（この場合は英語）を指定する `xml:lang` 属性があることに注意してください。

辞書には、任意の数の `<entry>` 要素を含めることができます。各 <entry> 要素には、次の 3 つのサブ要素が含まれます。

- `<word>`：文字列。ステミングの対象の語を入れます。
- `<stem>`：文字列。文字列 `<word>` の語幹です。

- <pos>：中国語や日本語など、語がスペースで区切られない言語の場合に、品詞を指定します。他の言語の場合、無視されます。省略された場合、Nn-Prop（固有名詞）がデフォルト値です。よく使用される値は次のとおりです。

値	品詞
Adj	形容詞
Adv	副詞
Interj	間投詞
Nn	名詞
NN-Prop	固有名詞（デフォルト）
Verb	動詞

18.3 カスタム辞書の関数

手持ちの XQuery モジュールで custom-dictionary.xqy モジュールを使用するには、XQuery プロローグに次の行を含めます。

```
import module namespace cdict =
  "http://marklogic.com/xdmp/custom-dictionary" at
  "/MarkLogic/custom-dictionary.xqy";
```

これにより、モジュールがインポートされ、モジュールの名前空間がカスタム辞書モジュールのプレフィックス cdict にバインドされます。

次に示すように、カスタム辞書に関連する 4 つの関数があります。カスタム辞書はそのままでは編集できないことに注意してください。この場合、まず辞書を取得する必要があります。次に、そのエントリーの追加、編集、または削除を行います。最後に、新しいバージョンを書き出すことによって古いバージョンを上書きします。

- ライセンスされたすべての言語の取得（cdict:get-languages）
- カスタム辞書の取得（cdict:dictionary-read）
- カスタム辞書の追加 / 書き込み（cdict:dictionary-write）
- カスタム辞書の削除（cdict:dictionary-delete）

これらの機能の詳細については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

18.3.1 ライセンスされたすべての言語の取得

```
cdict:get-languages()
  as xs:string*
```

http://marklogic.com/xdmp/privileges/custom-dictionary-user 権限が必要です。サーバーにライセンスされたすべての言語を表す 2 文字の ISO 言語コードを返します。コードとそれに関連付けられている言語のリストについては、http://www.loc.gov/standards/iso639-2/php/code_list.php を参照してください。MarkLogic では、zh の zh_Hant バリエーションも含め、2 文字の ISO 639-1 コードのみを使用していることに注意してください。

18.3.2 カスタム辞書の取得

```
cdict:dictionary-read(
  $lang as xs:string
) as element(cdict:dictionary)
```

http://marklogic.com/xdmp/privileges/custom-dictionary-user 権限が必要です。$lang は ISO 言語コードです。$lang が、カスタム辞書があるライセンス済み言語とマッチした場合、ローカルホストにより、そのカスタム辞書（クラスタ全体にわたって同じです）が返されます。関連付けられている言語を示す辞書の xml:lang 属性も返されます。$lang が、ライセンス済み言語ではない場合、XDMP-LANG エラーを生成します。

18.3.3 カスタム辞書の追加 / 書き込み

```
cdict:dictionary-write(
  $lang as xs:string,
  $dict as element(cdict:dictionary)
) as empty-sequence()
```

http://marklogic.com/xdmp/privileges/custom-dictionary-admin 権限が必要です。$lang は ISO 言語コードです。$dict はカスタム辞書です。$lang がライセンス済み言語とマッチし、$dict が検証された場合、クラスタにより、$dict がインストールされ、辞書の保存元のホスト ID が返されます。$lang が、ライセンス済み言語ではない場合、XDMP-LANG エラーを生成します。検証が失敗した場合、検証エラーが生成されます。dictionary-write では、xml:lang 属性は無視されます。

18.3.4 カスタム辞書の削除

```
cdict:dictionary-delete(
  $lang as xs:string
) as empty-sequence()
```

http://marklogic.com/xdmp/privileges/custom-dictionary-admin 権限が必要です。$lang は ISO 言語コードです。$lang がカスタム辞書を持つライセンス済み言語とマッチした場合、この辞書が削除され、辞書が削除されたホストの ID が返されます。$lang がライセンス済み言語ではない場合、XDMP-LANG エラーを生成します。

18.4 使用例

次のコードは、カスタム辞書の検索と取得、エントリーの追加と編集、修正した辞書の書き出し、そして最後にカスタム辞書の削除の方法を示しています。各コードサンプルの後に、レスポンスの例を示します。

まず、サポートおよびライセンス済み言語のシーケンスを取得します。

```
xquery version "1.0-ml";
import module namespace cdict =
"http://marklogic.com/xdmp/custom-dictionary"
                 at "/MarkLogic/custom-dictionary.xqy";

cdict:get-languages()

==> ("en", "ja", "zh", "zh_Hant")
```

次に、特定の言語、この場合は英語（en）の辞書コンテンツを取得します。

```
xquery version "1.0-ml";
import module namespace cdict =
"http://marklogic.com/xdmp/custom-dictionary"
                at "/MarkLogic/custom-dictionary.xqy";

cdict:dictionary-read("en")

=> <cdict:dictionary
        xmlns:cdict="http://marklogic.com/xdmp/custom-dictionary"
        xml:lang="en">
     <cdict:entry>
        <cdict:word>Furbies</cdict:word>
        <cdict:stem>Furby</cdict:stem>
     </cdict:entry>
     <cdict:entry>
        <cdict:word>servlets</cdict:word>
        <cdict:stem>servlet</cdict:stem>
     </cdict:entry>
    </cdict:dictionary>
```

ファイル（例えば /var/tmp/cdict-en.xml など）にコンテンツを入れた後、XQuery でファイルを編集または修正して、新しいエントリーの追加や、既存のエントリーの削除または修正を行います。

次に、修正した辞書を MarkLogic にインストールします。戻り値は、空のシーケンスです。辞書をどこに置くかを指定する必要はなく、関連付ける言語のみを指定することに注意してください。サーバーが配置先を認識しています。1 つの言語に対してカスタム辞書は 1 つしか存在しないので、このコマンドにより、言語の引数に関連付けられている既存のカスタム辞書が上書きされます。

```
xquery version "1.0-ml";
import module namespace dict =
"http://marklogic.com/xdmp/custom-dictionary"
                at "/MarkLogic/custom-dictionary.xqy";
let $dict := xdmp:document-get("/var/tmp/cdict-en.xml")/*
return
  cdict:dictionary-write("en",$dict)
=> empty sequence
```

最後に、英語のカスタム辞書を削除する場合は、次のようにします。これにより、空のシーケンスが返されます。

```
xquery version "1.0-ml";
import module namespace cdict =
"http://marklogic.com/xdmp/custom-dictionary"
                at "/MarkLogic/custom-dictionary.xqy";
cdict:dictionary-delete("en")
=> empty sequence

xquery version "1.0-ml";
import module namespace cdict =
"http://marklogic.com/xdmp/custom-dictionary"
                at "/MarkLogic/custom-dictionary.xqy";
cdict:dictionary-read("en");
```

前に示した例では、いくつかのクエリを使用しており、各クエリではカスタム辞書の読み取り、保存、修正のために個別のトランザクションを実行しています。次の例では、単一のトランザクションと同様に処理が実現され、XQuery プログラムでカスタム辞書の読み取り、そして修正が行われます。

```
(: Add an entry to the English custom dictionary :)
xquery version "1.0-ml";
import module namespace cdict =
         "http://marklogic.com/xdmp/custom-dictionary"
              at "/MarkLogic/custom-dictionary.xqy";
  let $language := "en"
(: Get the English custom dictionary :)
  let $dictionary := cdict:dictionary-read($language)
(: Specify the new entry element as XML (not as a string) :)
  let $entry := <cdict:entry>
                  <cdict:word>views</cdict:word>
                  <cdict:stem>view</cdict:stem>
                </cdict:entry>
(: First, check if there are already any dictionary entries :)
  return if (fn:empty($dictionary))
(: If no entries, then we have to create a cdict:dictionary
   element and insert our new entry before writing it out :)
      then cdict:dictionary-write($language,
                                  element cdict:dictionary {
                                      attribute xml:lang {
                                          $language },
                                      $entry})

(: If there are entries, just insert the new entry as a node and
   write out the dictionary :)
      else cdict:dictionary-write($language,
                                  xdmp:node-insert-child(
                                     $dictionary/dictionary,$entry))

(: Finally, test the new mapping; this should return "view":)
cts:stem("views", "en")
```

19.0 バイナリドキュメントからのメタデータとテキストの抽出

この章では、バイナリドキュメントからメタデータやテキストを抽出する方法について説明します。次のセクションを含んでいます。

- メタデータとテキスト抽出の概要
- 使用例
- サポートされているバイナリ形式

19.1 メタデータとテキスト抽出の概要

バイナリドキュメントには多くの場合、さまざまなメタデータが関連付けられています。例えば、カメラの JPEG 画像には、カメラのタイプとモデル番号、撮影時のタイムスタンプなどのメタデータが含まれていることが多いです。

MarkLogic サーバーでは、バイナリドキュメントメタデータにアクセスした後、それを XML としてプロパティドキュメントに格納できます。その後、MarkLogic サーバーのリッチ XML 検索機能を使用して、メタデータを検索および取得できます。さらに、Microsoft Word 形式のバイナリドキュメントなど、テキストベースのバイナリドキュメントに対して、MarkLogic ではテキストコンテンツの抽出とインデックス付けができます。

バイナリドキュメントからメタデータを抽出して関連付けるには、一般的に方法が 2 つあります。Information Studio での「Filter Documents」変換と、ビルトインの XQuery 関数 `xdmp:document-filter` です。

「Filter Documents」トランスフォーマーは、ある種のドキュメントにおいてテキストとメタデータを自動的に抽出し、これをプロパティとしてドキュメントに付与します。詳細については、『Information Studio Developer's Guide』の「Extracting Document Properties」を参照してください。

`xdmp:document-filter` 関数は、バイナリドキュメントからメタデータとテキストを XHTML として抽出します。この出力は、ドキュメントのプロパティとして使用できます。`xdmp:document-filter` によって抽出されたテキストには、若干のフォーマット情報（構造）が含まれているため、検索や分類といったテキスト処理に最適です。

19.2 使用例

以降のセクションでは、xdmp:document-filter がさまざまなファイルタイプでどのように機能するかを示します。「Microsoft Word」セクションではまた、メタデータとテキストを組み合わせた結果からメタデータ要素のみを抽出するためのコードを示します。

- Microsoft Word

- ファイルアーカイブ
- PowerPoint

19.2.1 Microsoft Word

次のクエリと結果は、テキスト「This is a test」のみを含んでいる Microsoft Word ドキュメントを対象としたものです。

```
xquery version "1.0-ml";
  xdmp:document-filter(doc("/documents/test.docx"))
```

返される内容：

```
<?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>
```

このドキュメントでは、語「test」に斜体と太字の両方が適用されています。`xdmp:document-filter` では、このようなテキスト書式を返しません。

前に示した例を拡張した次のコードでは、`xdmp:document-filter` を使用して、同じ Microsoft Word ドキュメントからメタデータのみを抽出します。

```
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)}
)
```

これで、プロパティドキュメントは次のようになります。

```
xdmp:document-properties("/documents/test.docx")
```

返される内容：

```
<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>
```

19.2.2　ファイルアーカイブ

個別の処理のために `zip` アーカイブからファイルを抽出する必要がある場合は、`xdmp:zip-manifest` と `xdmp:zip-get` を使用してください。アーカイブからすべてのテキストのみが必要である場合は、`xdmp:document-filter` を使用してください。これは、ファイルに埋め込まれた構造が保持されず、ドキュメントのテキストすべてが含まれるためです。これは、検索結果の元の位置を見つける場合に便利です。「Elvis」を検索し、さまざまなファイルで `xdmp:document-filter` を使用した場合、結果には、zip アーカイブ、Word ドキュメント、写真のいずれであっても、「Elvis」を含んでいるすべてのバイナリが含まれます。

この例では、`xdmp:document-filter` は 2 つの Word ファイルと 1 つの JPEG ファイルから成るファイルアーカイブ `test.zip` に対して実行されます。

```
xquery version "1.0-ml";
  xdmp:document-filter(doc("/documents/test.zip"))
```

返される内容

```
<?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>
```

この例で返される HTML body テキストの各文は異なるファイルからのものであり、各テキストがどのファイルから取得されたかを識別する方法はありません。同様に、返されるサブファイルメタデータは、ファイルの順序で返されることが保証されない（例えば、`name="a"` と `name="b"` は、アーカイブ内の異なるドキュメントからのものである可能性があります）ので、個々のサブファイルに正しく関連付けることもできません。

また、アーカイブ内の個々のサブファイルは、必ず識別できるというわけではありません。前述の例では、この出力からはアーカイブ内にあるファイルの数、またファイルのタイプはわかりません。アーカイブに対して `xdmp:document-filter` を使用する場合は、アーカイブをサブファイルの組み合わせではなく、単一のファイルと見なす必要があります。1 つのアーカイブファイルに含まれているメタデータとテキストはすべて取り戻すことはできますが、返された情報をその取得元の個々のサブファイルに関連付ける方法はありません。

19.2.3 PowerPoint

次のクエリと結果は、PowerPoint ドキュメントの 2 つのスライドを対象としたものです。各スライドには、タイトルと個別のコンテンツがあります。

```
xquery version "1.0-ml";
  xdmp:document-filter(doc("/documents/test.pptx"))
```

返される内容：

```
<?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>
```

同様に、テキスト書式、およびスライドでのテキストの役割（タイトル、本文など）を示すインジケーターは返されず、どのテキストがどのスライドに属するかを知る方法もありません。

19.3 サポートされているバイナリ形式

以降のセクションでは、`xdmp:document-filter` がメタデータを抽出できるバイナリファイル形式とファイル拡張子を示します。形式によっては、元のテキストを示します。多くの形式が存在するため、一般的なアプリケーションで分類し（データベース、マルチメディアなど）、次に分類に該当する形式と拡張子の一覧を示します。

一部の形式は、`xdmp:document-filter` によって特定できますが、実行可能ファイルのように、抽出するテキストやメタデータがありません。このような場合、返された `<meta name="content-type" content=.../>` によってファイルの形式を特定します。

- アーカイブ
- データベース
- 電子メールとメッセージング
- マルチメディア
- その他
- プレゼンテーション

- ラスター画像
- 表計算
- テキストとマークアップ
- ベクトル画像
- ワードプロセッサおよびオフィス全般

19.3.1 アーカイブ

形式：7-Zip、ACE、Apple ディスクイメージ、ARJ、Bzip2、ISO ディスクイメージ、Java アーカイブ、LZH、Microsoft Cabinet、Microsoft Office Binder、RedHat Package Manager、Roshal Archive、自己展開 .exe、StuffIt、StuffIt Self Extracting Archive、SuffIt X、GNU Zip、UNIX cpio、UNIX Tar、Zip、PKZip、WinZip

拡張子：.7Z、.ACE、.ARJ、.BZ2、.CAB、.CPIO、.DMG、.EXE、.GZ、.ISO、JAR、LZH、.ORD、.RAR、.RPM、.SIT、.SEA、.SITX、.TAR、.TBZ2、.ZIP

19.3.2 データベース

形式：dBase、dBase III、Microsoft Access、Paradox Database

拡張子：.DB、.DBF、.DB3、.MDB

19.3.3 電子メールとメッセージング

形式：MHT、Multipart Alternative、Multipart Digest、Multipart Mixed、Multipart Newsgroup、Multipart Signed、TNEF のいずれかの形式でエンコードされたメールメッセージ。また、Eudora、Microsoft Outlook、Microsoft Outlook3、Microsoft Outlook Express3 の個々の形式。Microsoft Outlook フォームテンプレート、Sendmail「mbox」、Thunderbird

拡張子：.EML、.MBOX、.MBX、.MHT、.MSG、.OFT、.PST

19.3.4 マルチメディア

形式：3GP、Adobe Flash、Adobe Flash ビデオ、AVI（Audio Video Interleave）、DVD Information File、DVD Video Object、Microsoft Windows Movie Maker、MIDI（Musical Instrument Digital Interface）、MPEG ビデオ、MPEG-1 Audio Layer 3、MPEG-4 ビデオ、MPEG-2 Audio Layer 3、OGG Flac Audio、OGG Vorbis Audio、QuickTime、Real Media、WAVE（Waveform Audio File Format）、Window Media Audio、Windows Media Video。

拡張子：.3GP、.AIFF、.AVI、.BUP、.FLAC、.FLV、.IFO、.MID、.MIDI、.MOV、.MP3、.MP4、.MPG、.MSWMM、.OGG、.RM、.SMF、.SWF、.VOB、.WAV、.WMA、.WMV

19.3.5　その他

形式：Apple 実行可能ファイル、BIN HEX Encoded、BitTorrent Metafile、Linux Executable and Linkable Format、Log File、Microsoft Project、Microsoft Windows DLL、Microsoft Windows 実行可能ファイル、Microsoft Windows インストーラー、Microsoft Windows、ショートカット、OAII（Open Access II）、VCard、Uniplex

拡張子：.BIN、.COM、.DLL、.ELF、.EXE、.HBX、.HEX、.HQXX、.LNK、.LOG、.MPP、.MPX、.MSI、.SYS、.TORRENT、.VCF

19.3.6　プレゼンテーション

形式：IBM Lotus Symphony プレゼンテーション、LibreOffice プレゼンテーション、Microsoft PowerPoint for Windows または Macintosh、OpenOffice Impress、StarOffice Impress

拡張子：.ODP、.ODS、.PPT、.PPTX、.SDI、.SDP、.SXI

19.3.7　ラスター画像

形式：Encapsulated PostScript、GIF（Grapics Interchange Format）、JPEG（Joint Photographic Experts Group）、Microsoft Document Imaging、Microsoft Windows Bitmap、PCX、PNG（Portable Network Graphic）、Progressive JPEG、TIFF（Tagged Image Format File）

拡張子：.BMP、.EPS、.GFA、.GIF、.GIFF、.JIF、.JPEG、.JPG、.JPE、.MDI、.PCX、.PNG、.TIF、.TIFF

19.3.8　表計算

形式：カンマ区切り値、Franeword Spreadsheet、IBM Lotus Symphony、LibreOffice Spreadsheet、Lotus 1-2-3、Microsoft Excel for Windows または Mac、Microsoft Works SS for DOS または Windows、OpenOffice Calc、StarOffice Calc

拡張子：.CSV、.FW3、.ODS、.SX、.SXC、.SXS、.XLS、XLSB、.XLSX、.WK.、.WK3、.WK4、.WKS、.WPS

19.3.9　テキストとマークアップ

形式：ASCII テキスト（7 ビットおよび 8 ビット）、ANSI テキスト（7 ビットおよび 8 ビット）、HTML（テキストのみ、公開コード、メタデータのみ）、IBM DCA、Microsoft HTML ヘルプ、Microsoft OneNote、リッチテキスト形式、SGML テキスト、ソース、Transcript、Unicode UTF8/UTF16/UCS2、XML、Windows 拡張メタファイル、Windows メタファイル

拡張子：.CHM、.DCA、.EMF、.HTM、.HTML、.ONE、.RFT、.RTF、.SGML、.TXT、.XML、.WMF

19.3.10　ベクトル画像

形式：Adobe Illustrator、Adobe InDesign、Adobe Photoshop、AutoCAD Drawing、AutoCAD drawing Exchange Format、Corel Draw 画像、Intergraph-Microstation CAD、MathCAD、Microsoft XPS、Microsoft Visio

拡張子：.AI、.CDR、.DGN、.DWG、.DXF、.INDD、.MCD、.OXPS、.PSD、.VSD、.XMCD、.XPS

19.3.11　ワードプロセッサおよびオフィス全般

形式：Adobe PDF、Adobe PostScript、Ami Pro for Windows、Apple iWork、Framework WP、Hangul、IBM DCA/FFT、IBM DisplayWrite、IBM Lotus Symphony Document、JustSystems 一太郎、LibreOfffice Document、Lotus Manuscript、Lotus Notes、Mass 11、Microsoft Publisher、Microsoft Word for DOS/Windows/Macintosh、QuarkXpress、MultiMate、MultiMate Advantage、OpenOffice Writer、Professional Write for DOS、Professional Write Plus for Windows、Q&A Write、QuickBooks Backup、QuickBooks for Windows、StarOffice Writer、TrueType Font、Wang IWP、Wang WP Plus、Windows Write、WinWord、WordPerfect for DOS/Macintosh/Windows、Wordstar for DOS/Windows、Wordstar 2000 for DOS、XYwrite

拡張子：.AMI、.DCA、DOC、.DOCX、.DOX、.DW4、.FFT、.FW3、.IWP、.JTD、.JBW、.JTT、.KEY、.M11、.MAN、.MANU、.MNU、.NSF、.NUMBERS、.ODT、PAGES、.PDF、.PS、.PUT、.QCx、.QXx、.PW、.PW1、.PW2、.QA、.QA3、.QBB、.QBW、.RFT、.SAM、.SXW、.SDW、.TTF、.WPD、WRI、.WS、.WS2、.WSD、.XY

20.0 ワイルドカード検索の概要と使い方

この章では、MarkLogic サーバーのワイルドカード検索について説明します。次のセクションから構成されます。

- MarkLogic サーバーでのワイルドカード
- ワイルドカード検索をオンにする
- 他の検索機能との関係

20.1 MarkLogic サーバーでのワイルドカード

ワイルドカード検索により、MarkLogic サーバーは、文字とワイルドカードの組み合わせとマッチした結果を返すことができます。ワイルドカード検索は、文字列との完全一致だけでなく文字パターンマッチング（クエリ内で指定された文字と、その文字パターンを含むドキュメント内の語の関係性を扱う）に基づきます。このセクションでは、ワイルドカードについて説明します。次のトピックから構成されます。

- ワイルドカード文字
- ワイルドカード検索の規則

20.1.1 ワイルドカード文字

MarkLogic サーバーでは、`*` と `?` の 2 つのワイルドカード文字がサポートされています。

- `*` は、0 個以上の非スペース文字とマッチされます。
- `?` は、正確に 1 個の非スペース文字とマッチされます。

例えば、`he*` は、`he`、`her`、`help`、`hello`、`helicopter` など、`he` で始まるあらゆる語とマッチされます。一方、`he?` は、`hem`、`hen` など、`he` で始まる 3 文字の語とのみマッチされます。

20.1.2 ワイルドカード検索の規則

次に、MarkLogic サーバーのワイルドカード検索の基本的な規則を示します。

- 単一の検索タームまたは検索語句に複数のワイルドカードを含めることができ、2 種のワイルドカード文字を組み合わせて使用できます。例えば、`m*??` は、`m` で始まり、3 文字以上の語とマッチされます。
- スペースは、語の区切りとして使用されます。また、ワイルドカードマッチは、1 語の範囲内でのみ機能します。例えば、`m*th*` は `method` とマッチされますが、`meet there` とはマッチされません。

- 値クエリ（例えば、cts:element-value-query、cts:element-value-match など）でワイルドカード * を単独で指定した場合、すべてとマッチされます（語の区切りを越えてマッチされます）。例えば、* は、値 meet me there とマッチされます。
- ワイルドカード * を非ワイルドカード文字とともに指定した場合、値レキシコンクエリ（例えば、cts:element-value-match など）ではマッチされますが、値クエリ（例えば、cts:element-value-query など）ではマッチされません。例えば m* は、値レキシコン検索（例えば cts:element-value-match など）で値 meet me there とマッチされますが、値クエリ検索（例えば、cts:element-value-query など）ではこの値とマッチされません。値クエリは、1 つの語のみとマッチされるためです。m* * の値検索では、この値とマッチされます（m* が最初の語とマッチされ、* がその後のすべてとマッチされるため）。
- cts:query 式で「wildcarded」を明示的に指定した場合は、検索がワイルドカード検索として実行されます。
- 「wildcarded」と「unwildcarded」のいずれも cts:query 式で指定されていない場合には、データベース設定とクエリテキストによってワイルドカード機能の有無が決定されます。データベースでいずれかのワイルドカードインデックスがオンである場合（three character searches、two character searches、one character searches、または trailing wildcard searches）、およびクエリテキストにワイルドカード文字 ? または * のいずれかが含まれている場合、ワイルドカード文字はワイルドカードとして扱われ、検索が「wildcarded」として実行されます。いずれのワイルドカードインデックスもオンではない場合、ワイルドカード文字は句読点として扱われ、検索が unwildcarded として実行されます（cts:query 式で「wildcarded」が指定されていない場合）。
- クエリに punctuation-sensitive オプションがある場合、ワイルドカード検索では句読点は文字として扱われます。例えば、d*benz に対する punctuation-sensitive ワイルドカード検索は、daimler-benz がマッチします。
- クエリに whitespace-sensitive オプションがある場合、スペースは文字として扱われます。これは、ワイルドカードありの値クエリでスペースとマッチさせるときに役立ちます。ワイルドカードありのワードクエリでは whitespace-sensitive オプションも使用できますが、意図したものよりも多くマッチするので、あまり役に立たないかもしれません。

20.2 ワイルドカード検索をオンにする

ワイルドカード検索では、パフォーマンスを高速化するために文字インデックス、レキシコン、および末尾ワイルドカードインデックスを使用します。ワイルドカード検索を高速に行うには、データベース設定画面である Admin 画面で少なくとも 1 つのワイルドカードインデックス（three character searches、trailing wildcard searches、two character searches、または one character searches）と fast element character searches（特定要素内で高速な検索が必要な場合）をオンにする必要があります。デフォルトでは、ワイルドカードはオフです。文字インデックスをオンにする場合は、ソースコンテンツのサイズの 3 倍程度の追加ディスク領域を割り当てることを計画する必要があります。

このセクションでは、次のトピックについて説明します。

- クエリでのワイルドカードの指定
- 推奨されるワイルドカードインデックス設定
- さまざまなワイルドカードインデックスについて

20.2.1 クエリでのワイルドカードの指定

データベースに対していずれかのワイルドカードインデックスがオンになっている場合は、さらにクエリレベルでワイルドカードの使用を制御できます。ワイルドカードは、`cts:word-query`、`cts:element-word-query`、`cts:element-value-query` など、どの MarkLogic `cts:query` リーフレベル関数でも使用できます。`cts:query` 関数の詳細については、「cts:query 式の構成」（239 ページ）を参照してください。`cts:query` コンストラクタ関数で「`wildcarded`」および「`unwildcarded`」クエリオプションを使用して、ワイルドカード機能を明示的にオンまたはオフにできます。詳細については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

ワイルドカードオプションを指定しないまま、いずれかのワイルドカードインデックスをオンにした場合、MarkLogic サーバーでは、クエリに `*` または `?` が存在するときにワイルドカードクエリが実行されます。例えば、検索関数

```
cts:search(fn:doc(), cts:word-query("he*"))
```

では、ワイルドカード検索が実行されます。したがって、データベース内でいずれかのワイルドカードインデックスをオンにすれば、ワイルドカード検索を実行するためにワイルドカード機能を明示的にオンにする必要はありません。

また、データベース内でワイルドカードインデックス付けがオンになっている場合、ほとんどのクエリ式で、`fn:contains`、`fn:matches`、`fn:starts-with`、および `fn:ends-with` のパフォーマンスも向上します。

> **注：** データベース内で文字インデックス、レキシコン、末尾（trailing）ワイルドカードインデックスがすべてオフになっていて、（リーフレベルの `cts:query` コンストラクタへの「`wildcarded`」オプションを指定して）クエリで明示的にワイルドカード機能をオンにした場合、クエリは実行されますが、必要な処理が大きくなる可能性が高いです。このようなクエリは、対象を選択して比較的少量のコンテンツに対してのみワイルドカード検索を実行する必要がある場合は高速ですが、実際に大量のコンテンツから結果をフィルタリングで除外する必要がある場合は長い時間がかかることがあります。

20.2.2 推奨されるワイルドカードインデックス設定

MarkLogic では、パフォーマンスとデータベースサイズのバランスが最善となるようにあらゆる種類のワイルドカードクエリ機能が使えるよう、次のインデックス設定を推奨しています。

- word searches
- three character word searches
- word positions
- コードポイントコレーション内の word lexicon
- three character word positions

これらにより、先頭（leading）および末尾ワイルドカード付き検索も含む、さまざまなワイルドカード検索で正確かつ高速なワイルドカードクエリを作成できます。trailing wildcard searches インデックスを追加した場合、末尾ワイルドカード検索の効率がわずかに向上しますが、データベースサイズが増大します。次のセクションでは、さまざまなワイルドカードインデックスについて説明します。

20.2.3 さまざまなワイルドカードインデックスについて

Admin 画面を使用して、データベースレベルでインデックス設定を行います。データベース設定およびその他のテキストインデックスの詳細については、『Administrator's Guide』の「Databases」の章および「Text Indexes」の章を参照してください。

次の表では、ワイルドカード検索に適用されるインデックス付けオプションについて説明します。

インデックス	説明
ワードレキシコン	ワイルドカード検索を高速化します。検索のインデックス解決とパフォーマンスの向上のために、他に利用可能なワイルドカードインデックスと組み合わせて機能します。three character search インデックスと組み合わせて使用した場合、ワイルドカードのインデックス解決を向上させ、ワイルドカード検索を高速化します。データベースに対して three character search およびワードレキシコンをオンにした場合、one character インデックスまたは two character search インデックスのいずれも必要ありません。最適なパフォーマンスを得るには、ワードレキシコンがコードポイントコレーション（http://marklogic.com/collation/codepoint）内にあることが必要です。さらに、データベース内で word searches をオンにすると、ワイルドカードのインデックス解決の精度が向上します（xdmp:estimate クエリがより正確になります）。

インデックス	説明
trailing wildcard searches	末尾にワイルドカード文字を含んでいる検索パターン（例えば、abc* など）で、ワイルドカード検索を高速化します。末尾ワイルドカードとマッチするワイルドカード検索を高速化する場合は、このインデックスをオンにしてください。trailing wildcard search インデックスは、three character search インデックスと同様の容量のスペースを使用しますが、一般に、末尾ワイルドカードクエリの場合に効率が向上します。
three character searches	3 文字以上の連続する非ワイルドカード文字（例えば、*abc など）を含んでいる検索パターンで、ワイルドカード検索を高速化します。ワイルドカード式内のあらゆる場所で 3 文字以上とマッチするワイルドカード検索を高速化する場合は、このインデックスをオンにしてください。コードポイントワードレキシコンと組み合わされると、（連続する非ワイルドカード文字が 3 文字よりも少ない検索も含めて）あらゆるワイルドカード検索のパフォーマンスを高速化します。MarkLogic では、three character search インデックスをコードポイントコレーションのワードレキシコンと組み合わせることを推奨しています。
fast element character searches	特定の要素内でのワイルドカード検索を高速化します。また、要素ベースのワイルドカード検索も高速化します。特定の要素に対してクエリを実行するワイルドカード検索のパフォーマンスを向上させる場合は、このインデックスをオンにしてください。
two character searches	2 文字以上の連続する非ワイルドカード文字を含んでいる検索パターンで、ワイルドカード検索を高速化します。2 文字以上とマッチするワイルドカード検索（例えば、ab* など）を高速化する場合は、このインデックスをオンにしてください。three character searches とワードレキシコンがある場合、このインデックスは必要ありません。
要素、属性、およびフィールドワードレキシコン	cts:element-value-query 式および cts:element-attribute-value-query 式のワイルドカード検索を高速化します。検索のインデックス解決とパフォーマンスの向上のために、他に利用可能なワイルドカードインデックスと組み合わせて機能します。three character search インデックスと組み合わせて使用した場合、ワイルドカードのインデックス解決を向上させ、ワイルドカード検索を高速化します。

インデックス	説明
one character searches	非ワイルドカード文字 1 文字のみを含む検索パターンで、ワイルドカード検索を高速化します。1 つあるいは複数の文字とマッチするワイルドカード検索（例えば、a* など）を高速化する場合は、このインデックスをオンにしてください。three character searches とワードレキシコンがある場合、このインデックスは必要ありません。

すべてのインデックス付けの場合と同様、必要なインデックスの選択はトレードオフの関係にあります。インデックスを追加すると、クエリのパフォーマンスが向上しますが、使用されるディスク領域が増大し、読み込みおよび再インデックス付けが遅くなります。MarkLogic では、ワイルドカード検索が必要なほとんどの環境について、three character searches とコードポイントコレーションのワードレキシコンをオンにし、one character searches と two character searches をオフにすることを推奨しています。

ワードレキシコンと組み合わせた three character searches インデックスにより、ほとんどのクエリで最適なパフォーマンスが得られます。また、要素クエリを送信する場合、fast element character searches インデックスが役立ちます。1 文字または 2 文字のみのマッチを試みるワイルドカード検索を行う場合で、ワードレキシコンと three character searches インデックスを一緒に使わないときは、one character searches インデックスと two character searches インデックスのみが使用されます。one character searches と two character searches では一般に多数のマッチが返されるので、ディスク領域や読み込み時間とのトレードオフを正当化できないことが考えられます。

> **注：** three character searches インデックスがオンであり、2 文字インデックスと 1 文字インデックスがオフである場合、およびワードレキシコンがない場合でも、2 文字または 1 文字の語幹（例えば ab* または a*）を検索するワイルドカードクエリを発行することはできます。こうした検索は許可されますが、高速にはなりません。このようなクエリを入力できる検索ユーザーインターフェイスがある場合は、2 文字または 1 文字のワイルドカード検索パターンをチェックしてエラーを発生させてみるといいでしょう。これは対応するインデックスのないこのような検索は低速であり、リソースを大量に消費する可能性があるためです。代わりに、データベースにコードポイントコレーションのワードレキシコンを追加することもできます。

20.3 他の検索機能との関係

このセクションでは、MarkLogic サーバーのワイルドカード、ステミング、およびその他の検索機能の関係について説明します。次のトピックから構成されます。

- ワイルドカード機能とステミング
- ワイルドカード機能と句読点による区別

20.3.1 ワイルドカード機能とステミング

ワイルドカード検索は、ステミングと組み合わせて使用できます（ステミングの詳細については、「ステミング検索の概要と使い方」（385 ページ）を参照）。つまり、クエリでは、ステミング検索とワイルドカード検索を同時に実行できます。ただし、ワイルドカードがある語に対しては、ステミング検索は実行されません。例えば、検索語句が `running car*` であるとします。ターム `running` は、語幹に基づいてマッチされます。一方、`car*` は、ワイルドカード検索に基づいてマッチされ、`car`、`cars`、`carriage`、`carpenter` などとマッチします。ワイルドカードとマッチする語に対しては、ステミングされた語のマッチは返されません。

20.3.2 ワイルドカード機能と句読点による区別

ステミングと句読点による区別（punctuation sensitivity）は、互いに独立して実行されます。ただし、ワイルドカード機能と句読点による区別の間には関係があります。このセクションでは、この関係について説明します。次のパートから構成されます。

- 句読点の暗黙的および明示的な指定
- 句読点とワイルドカード機能の相互作用に関する規則
- ワイルドカードと句読点の関係の例

20.3.2.1 句読点の暗黙的および明示的な指定

MarkLogic サーバーでは、クエリで句読点による区別を行うかどうか、およびワイルドカードを使用するかどうかについて、明示的に指定できます。これは、次の例に示すように、クエリに対するオプションで指定します。

```
cts:search(fn:doc(), cts:word-query("hello!", "punctuation-sensitive") )
```

句読点による区別がある検索にワイルドカード文字を含めた場合、ワイルドカードは句読点として扱われます。例えば、次のクエリは、`hello*` とマッチしますが、`hellothere` とはマッチしません。

```
cts:search(fn:doc(), cts:word-query("hello*", "punctuation-sensitive") )
```

句読点による区別のオプションが未指定のままである場合、クエリタームに何らかの非ワイルドカードの句読点があるときに、句読点による区別がある（punctuation sensitive）検索が実行されます。例えば、句読点を指定しなかった場合、クエリ

```
cts:search(fn:doc(), cts:word-query("hello!") )
```

では、句読点による区別がある検索が実行されます。また、クエリ

```
cts:search(fn:doc(), cts:word-query("hello") )
```

では、句読点による区別がない検索が実行されます。

（暗黙的または明示的であるかを問わず）検索で句読点による区別がある場合、MarkLogic サーバーでは、検索タームに加えて句読点ともマッチされます。句読点は語の一部とは見なされないことに注意してください。例えば、mark! では、感嘆符の隣にある mark が語であると見なされます。検索で句読点による区別がない場合、句読点はスペースとマッチされます。

20.3.2.2 句読点とワイルドカード機能の相互作用に関する規則

文字 ? と * は、データベースに読み込まれたドキュメント内では句読点と見なされます。しかし、? と * は、クエリではワイルドカード文字としても扱われます。これにより、ワイルドカード機能と句読点による区別の間で、興味深い（わかりにくい）相互作用が生じます。

次に、句読点とワイルドカード機能の相互作用に関する規則を示します。これは、句読点とワイルドカード文字の間に相互作用の関係がある際、システムの動作を判断するうえで役立ちます。

1. データベース内でワイルドカードインデックスがオフである場合、すべてのクエリがデフォルトで「unwildcarded」になり、ワイルドカード文字は句読点として扱われます。クエリ内で「wildcarded」を指定した場合、このクエリはワイルドカードクエリになり、ワイルドカード文字はワイルドカードとして扱われます。

2. ワイルドカード機能は、句読点による区別よりも優先されます。つまり、クエリ内に文字 * や ? が存在する場合、ワイルドカードがオフにされていない限り、* および ? はワイルドカードとして扱われ、句読点としては扱われません。クエリ内でワイルドカード機能がオフにされている場合（「unwildcarded」）、これらの文字は句読点として扱われます。

3. ワイルドカード機能と句読点による区別がともに明示的にオフになっていて、クエリ内に句読点（*、? を含む）がある場合、これらの文字はスペースとして扱われます。

4. ワイルドカード機能と句読点による区別は同時にオンにできます。この場合、ドキュメント内の句読点は文字として扱われ、またクエリ内のワイルドカードは句読点文字も含む、クエリ内のあらゆる文字とマッチします。したがって、次のクエリは hello* と hellothere の両方とマッチされます。

```
cts:search(fn:doc(),
           cts:word-query("hello*",
                          ("punctuation-sensitive", "wildcarded") )
           )
```

20.3.2.3 ワイルドカードと句読点の関係の例

このセクションは、次のカテゴリのクエリの出力の例を含んでいます。

- ワイルドカード機能と句読点による区別を指定しない場合（ワイルドカードインデックスはオン）
- ワイルドカード機能を明示的にオフにし、句読点による区別を指定しない
- ワイルドカード機能を指定せず、句読点による区別を明示的にオンにした場合（ワイルドカードインデックスがオン）

ワイルドカード機能と句読点による区別を指定しない場合（ワイルドカードインデックスはオン）

以降の例では、少なくとも 1 つのワイルドカードインデックスがオンであり、`cts:word-query` でオプションが明示的に設定されていない場合に実行されるクエリを示しています。

- **クエリの例：**`cts:word-query("hello world")`

  実際の動作：ワイルドカード機能オフ、句読点による区別なし

  マッチする語句：`hello world`、`hello ?! world`、`hello? world!` など

- **クエリの例：**`cts:word-query("hello?world")`

  実際の動作：ワイルドカード機能オン、句読点による区別なし

  マッチする語句：`helloaworld`

  マッチしない語句：`hello world`、`hello!world`

- **クエリの例：**`cts:word-query("hello*world")`

  実際の動作：ワイルドカード機能オン、句読点による区別なし

  マッチする語句：`helloabcworld`

  マッチしない語句：`hello to world`、`hello-to-world`

- **クエリの例：**`cts:word-query("hello * world")`

  実際の動作：ワイルドカード機能オン、句読点による区別なし

  マッチする語句：`hello to world`、`hello-to-world`

  マッチしない語句：`helloaworld`、`hello world`、`hello ! world`

注： 連続する複数のスペースは、サーバーで文字列を比較する際には 1 つのスペースとして扱われます（圧縮されます）。クエリ語句 `hello * world` では、アスタリスクの両側の 2 つのスペースは互いに隣接していないので、比較の際に圧縮されません。このため、`hello world` では、`hello` と `world` の間にスペースは 1 つだけあり、一方 `hello * world` ではスペースが 2 つある（これらのスペースは圧縮されない）ので、これらはマッチしません。また、語句 `hello ! world` もマッチしません。これは、`!` がスペースとして扱われ（句読点による区別なし）、連続する 3 つのスペースすべてが文字列の比較前に 1 つのスペースに圧縮されるためです。

- **クエリの例：** `cts:word-query("hello! world")`

  実際の動作： ワイルドカード機能オフ、句読点による区別あり

  マッチする語句： `hello! world`

  マッチしない語句： `hello world`、`hello; world`

- **クエリの例：** `cts:word-query("hey! world?")`

  実際の動作： ワイルドカード機能オン、句読点による区別あり

  マッチする語句： `hey! world?`、`hey! world!`、`hey! worlds`

  マッチしない語句： `hey. world`

ワイルドカード機能を明示的にオフにし、句読点による区別を指定しない

以降の例では、「`unwildcarded`」を指定し、句読点による区別については何も指定しないクエリに対するマッチを示しています。

- **クエリの例：** `cts:word-query("hello?world", "unwildcarded")`

  実際の動作： ワイルドカード機能オフ、句読点による区別あり

  マッチする語句： `hello?world`

  マッチしない語句： `hello world`、`hello;world`

- **クエリの例：** `cts:word-query("hello*world", "unwildcarded")`

  実際の動作： ワイルドカード機能オフ、句読点による区別あり

  マッチする語句： `hello*world`

  マッチしない語句： `helloabcworld`

ワイルドカード機能を指定せず、句読点による区別を明示的にオンにした場合（ワイルドカードインデックスがオン）

以降の例では、少なくとも 1 つのワイルドカードインデックスがオンであり、`cts:word-query` で「`punctuation-sensitive`」オプションが明示的に設定されている場合に実行されるクエリを示しています。

- **クエリの例：**`cts:word-query("hello?world", "punctuation-sensitive")`

  実際の動作：ワイルドカード機能オン、句読点による区別あり

  マッチする語句：`hello?world`、`hello.world`、`hello*world`

  マッチしない語句：`hello world`、`hello ! world`

- **クエリの例：**`cts:word-query("hello * world", "punctuation-sensitive")`

  実際の動作：ワイルドカード機能オン、句読点による区別あり

  マッチする語句：`hello abc world`、`hello ! world`

  マッチしない語句：`hello-!- world`

- **クエリの例：**`cts:word-query("hello? world", "punctuation-sensitive")`

  実際の動作：ワイルドカード機能オン、句読点による区別あり

  マッチする語句：`hello! world`、`(hello) world`

  注： `(hello) world` は、`?` が `)` とマッチし、`(` が語 `hello` の一部とは見なされないため、マッチします。

  マッチしない語句：`ahello) world`、`hello to world`

21.0 コレクション

MarkLogic サーバーには、ドキュメントのグループであるコレクションが用意されています。コレクションとして MarkLogic データベース内のコンテンツをサブセットにまとめることで、効率的にクエリを実行できます。

コレクションは W3C XQuery 規格の一部として記載されていますが、その実装は定義されていません。MarkLogic は、クエリの処理対象となるドキュメントを選択するための、強力かつ高パフォーマンスなメカニズムとして、コレクションを活用しています。この章では、`collection()` 関数を紹介するとともに、コレクションを定義してアクセスする方法と、開発者が知っておくべき基本的なパフォーマンス特性について説明します。この章は、以下のセクションで構成されています。

- collection() 関数
- コレクションとディレクトリの比較
- コレクションの定義
- コレクションのメンバーシップ
- コレクションとセキュリティ
- パフォーマンス特性

21.1 collection() 関数

`collection()` 関数は、`doc()` 関数または `input()` 関数を使用する XQuery 内の任意の場所で使用できます。`collection()` 関数は、次のシグネチャを持ちます。

```
fn:collection($URI as xs:string*) as node*
```

注： MarkLogic サーバーにおける `collection()` 関数の実装は、URI のシーケンスを取ります。したがって、1 つあるいは複数のコレクションに対して `collection()` 関数を呼び出すことができます。W3C XQuery ドキュメントにおける関数のシグネチャでは、文字列を 1 つだけ受け取ります。また、MarkLogic サーバーに `fn` 名前空間が組み込まれているので、名前空間をプレフィックスとして関数に付加する必要はありません。

`collection()` 関数の用途を説明するため、次の 2 つの XPath 式について考えます。

```
fn:doc()//sonnet/line[cts:contains(., "flower")]

collection("english-lit/shakespeare")//sonnet/
                                      line[cts:contains(., "flower")]
```

最初の式は、line ノードのシーケンスを返します。それぞれの line ノードは、sonnet ノードの子になっています。さらに、それぞれの sonnet ノードは、大文字 / 小文字の区別なしでターム *flower* を含む必要があります。

2 番目の式も同じ line ノードのシーケンスを返しますが、対象は `english-lit/shakespeare` コレクションのメンバーであるドキュメント内に含まれている line ノードだけです。MarkLogic サーバーは、この式を最適化します。`collection()` 関数による演算は、この XPath 式の残りの部分と同様に、一連のインデックスの参照によって極めて効率的に実行されます。

前述のように、`collection()` 関数は単一のコレクション（上記の例を参照）も、コレクションのシーケンス（下記の例を参照）も取ることができます。

```
collection(("english-lit/shakespeare",
            "american-lit/poetry"))//sonnet/
                                line[cts:contains(., "flower")]
```

上記のクエリは、`english-lit/shakespeare` コレクションまたは `american-lit/poetry` コレクションのどちらか一方または両方のメンバーである、指定の述語と一致した line ノードのシーケンスを返します。このような変更を `collection()` 関数に加えた場合、その形式は、同様に URI のシーケンスを受け取る `doc()` 関数形式と類似します。複雑なブール型のメンバーシップ条件に関する XPath レベルでのサポートは、現在のところ存在しません（例えば、複数のコレクションへのメンバーシップの要求（and）、特定のコレクションに属しているドキュメントの除外（not）、厳密に二者択一のメンバーシップの要求（exclusive or）など）。ただしこのような条件は、周囲の FLWOR 式に `where` 句を含めることで実現できます（例については、「コレクションのメンバーシップ」（420 ページ）を参照）。

21.2 コレクションとディレクトリの比較

コレクションは、データベース内のドキュメントの整理に使用します。ディレクトリを使用しても、データベース内のドキュメントを整理できます。ドキュメントの整理におけるコレクションとディレクトリの違いは以下のようになります。

- コレクションでは、所属するドキュメントが特定の URI パターンを満たす必要はありません。コレクションは階層型ではありません（ディレクトリは階層型です）。ドキュメントは任意のコレクションに所属できます。また複数のコレクションに所属できます。
- 1 つのコレクション内のドキュメントをすべて削除するには、`xdmp:collection-delete` 関数を使用します。同様に 1 つのディレクトリ内のドキュメントをすべて削除するには、`xdmp:directory-delete` 関数を使用します（そのディレクトリに含まれるディレクトリとそれに含まれるドキュメントもすべて削除されます）。
- コレクションにはプロパティを設定できませんが、ディレクトリにはできます。

ドキュメントの整理にはコレクションとディレクトリの両方が利用できます。しかしコレクションとディレクトリには関係がないこと（全く別の物だということ）に注意してください。ディレクトリの詳細については、『Application Developer's Guide』の「Properties Documents and Directories」を参照してください。

21.3 コレクションの定義

各ドキュメントにおけるコレクションのメンバーシップは暗黙的に定義されます。MarkLogic サーバーでは、コレクションをトップダウンで規定（特定のコレクションに属するドキュメントのリストを指定）するのではなく、自らをこのコレクションのメンバーであると規定しているドキュメントをまとめることで、メンバーシップがボトムアップで決定されます。MarkLogic サーバーのセキュリティスキーマを使用して、コレクションのメンバーシップに関連するポリシーを管理できます。

コレクションの名前は、URI となっています。任意の URI をコレクションの正式な名前として使用できます。URI は、データベースの一連のコレクション（保護されているコレクションと保護されていないコレクションの両方）において一意である必要があります。

コレクション名の URI は、サーバーにとっての識別子の意味しかありません。つまり実際には、コレクションがファイルシステムのディレクトリのようにモデル化されているわけでは*ありません*。コレクションは階層としてではなく、集合（セット）として解釈されます。例えば、コレクション `english-lit/poetry/sonnets` に属しているドキュメントが、コレクション `english-lit/poetry` に属す必要はありません。実際に、URI `english-lit/poetry/sonnets` のコレクションが存在するからといって、URI `english-lit/poetry` や URI `english-lit` のコレクションが存在するわけではありません。

MarkLogic サーバーでサポートされているコレクションには、保護されていないコレクションと保護されているコレクションという 2 つのタイプが存在します。この 2 つのタイプは、`collection()` 関数のシンタックスに関しては同じです。ただし、コレクションにアクセスできるユーザーや、コレクションのドキュメントを変更、追加、および削除できるユーザーの定義方法が違います。次のサブセクションでは、この 2 つの方法でコレクションを定義する場合について説明します。

- 保護されていないコレクションの暗黙的な定義
- 保護されているコレクションの明示的な定義

21.3.1 保護されていないコレクションの暗黙的な定義

保護されていないコレクションは、*暗黙*的に作成されています。

ドキュメントを初めてシステムに読み込む際、XQuery または XDBC の読み込みディレクティブで、ドキュメントの所属コレクションを任意に指定できます。このコレクションのリストにおいて、以前に使用されたことがないコレクション URI を指定するだけで、保護されていないコレクションが新規作成されます。

読み込みディレクティブでコレクションを指定しなかった場合、ドキュメントは、現在のユーザーに定義されているデフォルトのコレクションのメンバーシップでデータベースに追加されます。デフォルトのコレクションは、セキュリティモデルに従って、現在のユーザーのロールから継承されます。このようなデフォルト設定を使って、保護されていないコレクションを新規作成することもできます。読み込みディレクティブでコレクションを指定せず、かつ現在のユーザーにデフォルトのコレクションが定義されていない場合、ドキュメントはコレクションにまったく属していない状態で、データベースに追加されます。

さらに、ドキュメントがデータベースに読み込まれた後で、次のいずれかのビルトイン XQuery 関数によって、コレクションのメンバーシップを変更できます（対象のドキュメントを変更できる適切なパーミッションがある場合）。

- `xdmp:document-add-collections`
- `xdmp:document-remove-collections`
- `xdmp:document-set-collections`

データベースで使用されていないコレクション URI をパラメータとして `xdmp:document-add-collections` や `xdmp:document-set-collections` に渡した場合、保護されていないコレクションが新規作成されます。

保護されていないコレクションは、メンバーであるドキュメントがデータベース内に存在しなくなると、消去されます。したがって、`xdmp:document-remove-collections`、`xdmp:document-set-collections`、または `xdmp:document-delete` を使用すると、保護されていないコレクションが消去される可能性があります。

`xdmp:collection-delete` 関数は、データベース内で特定のコレクションに属しているドキュメントをすべて削除します（ドキュメントごとに必要なパーミッションを現在のユーザーが持っている場合）。この関数を使用すると、指定された保護されていないコレクションは常に消去されます。

> **注：** `xdmp:collection-delete` 関数は、他のコレクションでのメンバーシップに関係なく、コレクション内のドキュメントをすべて削除します。

21.3.2 保護されているコレクションの明示的な定義

保護されているコレクションは、*明示*的に作成します。

保護されているコレクションは、保護されていないコレクションでは得られない、一定のセキュリティ保護を提供します（「コレクションとセキュリティ」（421 ページ）を参照）。したがって、上記で説明した暗黙的なモデルとは異なり、保護されているコレクションは、ドキュメントを含める*前に*、Admin 画面で明示的に定義しておく必要があります。

保護されているコレクションとそのセキュリティポリシーを定義したら、上記で説明した保護されていないコレクションの場合と同様に、ドキュメントをそのコレクションに追加できます。ただし、ドキュメントを変更するパーミッションに加えて、保護されているコレクションを変更するパーミッションも、ユーザーが保持している必要があります。

保護されているコレクションは明示的に作成されるのと同様に、データベースの状態が変化して、保護されているコレクションに属しているドキュメントが存在しなくなっても、このコレクションは消去されません。保護されているコレクションをデータベースから削除するには、Admin 画面を使用して、コレクションの定義を削除する必要があります。

21.4 コレクションのメンバーシップ

上述したように、特定のドキュメントの所属コレクション（保護されていないコレクションと保護されているコレクション）は、読み込み時に指定できます。また、データベースへのドキュメントの読み込み後に変更できます。ドキュメントは、同時に複数のコレクションに所属できます。

読み込み時に特定のコレクションが定義されていない場合、コレクションのメンバーシップはサーバーによって自動的に割り当てられます。これはユーザーやユーザーのロールのデフォルトのコレクションメンバーシップ設定に基づきます。コレクションにまったく属さないドキュメントを読み込むには、コレクションのパラメータとして空のシーケンスを明示的に指定します。

コレクションのメンバーシップは、`collection()` 関数、`doc()` 関数、または `input()` 関数を使用する任意の XPath 式で利用できます。さらに、特定のドキュメントやノードのコレクションのメンバーシップは、ビルトインの `xdmp:document-get-collections` によるクエリが可能です。

例えば、次の式は、line ノードのシーケンスを返します。各 line ノードはそれぞれ、sonnet ノードの子であるとともに、大文字 / 小文字の区別なくターム *flower* を含む必要があり、`english-lit/shakespeare` コレクションと `american-lit/poetry` コレクションの一方または両方に属しています。

```
collection(("english-lit/shakespeare",
            "american-lit/poetry"))//sonnet/
                                line[cts:contains(., "flower")]
```

これに対して、次の式は同様の line ノードのシーケンスを返しますが、その line ノードは `english-lit/shakespeare` コレクションには属しておらず、`english-lit/poetry` コレクションと `american-lit/poetry` コレクションの一方または両方に属しているという点が異なります。

```
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
```

21.5 コレクションとセキュリティ

コレクションと MarkLogic サーバーのセキュリティモデルの基本的な関係として、次の 3 つが挙げられます。

- すべてのユーザーおよびロールが、オプションとしてデフォルトのコレクションを指定できます。読み込みの際にコレクションが明示的に指定されていない場合、このデフォルトのコレクションに、新しく挿入されたドキュメントが追加されます。
- 読み込みの際ならびに読み込んだ後に、ドキュメントをコレクションに追加するには、対象ドキュメントを挿入または更新するパーミッションが必要です。
- コレクションからドキュメントを削除する処理や、`xdmp:collection-delete` を使用する処理も同様に、対象ドキュメントを更新するパーミッションが必要です。

さらに、保護されているコレクションと MarkLogic サーバーのセキュリティモデルの関係として、次の 3 つが挙げられます。

- 保護されているコレクションは、Admin 画面のセキュリティモジュールで設定する必要があります。
- 保護されているコレクションには、保護されているコレクションに対する読み取り、挿入、および更新のパーミッションを持つロールを指定します。
- 保護されているコレクションに対してドキュメントを追加または削除する処理では、ドキュメントに対する適切なパーミッションだけでなく、関連するコレクションに対する適切なパーミッションも必要になります。

21.5.1 保護されていないコレクション

1 つあるいは複数の保護されていないコレクションに属する新しいドキュメントをデータベースに追加するには、ドキュメントの追加のパーミッションが（直接または間接的に）必要です。つまり、ユーザーは admin ロールを保持しているか、次の両方を保持している必要があります。

- `xdmp:document-load` 関数を実行する権限（この関数をドキュメント挿入ディレクティブとして使用する場合）。
- `unprotected-uri` 権限、`any-uri` 権限、または挿入するドキュメントの特定のパスに対する適切な URI 権限。例えば、挿入するドキュメントの URI が `/docs/poetry/love.xml` である場合、適切な URI 権限は `/`、`/docs`、`/docs/poetry` です。

ドキュメントが所属する一連のコレクションを変更するには、admin ロールか、ドキュメントの更新権限が必要です。

保護されていないコレクションに XPath 式でアクセスする場合、特別なパーミッションは不要です。指定されたコレクションに属している個々のドキュメントへのアクセスは、個々のドキュメントの読み取り権限によって管理されます。

21.5.2 保護されているコレクション

1 つあるいは複数の保護されているコレクションに属する新しいドキュメントをデータベースに追加するには、ドキュメントの追加のパーミッションとともに、保護されているコレクションへの追加のパーミッションが（直接または間接的に）必要です。つまり、ユーザーは admin ロールを保持しているか、次のすべてを保持している必要があります。

- 保護されているコレクションに対する挿入パーミッション。
- `xdmp:document-load` 関数を実行する権限（この関数をドキュメント挿入ディレクティブとして使用する場合）。
- `unprotected-uri` 権限、`any-uri` 権限、または挿入するドキュメントの特定のパスに対する適切な URI 権限。例えば、挿入するドキュメントの URI が `/docs/poetry/love.xml` である場合、適切な URI 権限は `/`、`/docs`、`/docs/poetry` です。

ドキュメントが所属する一連の保護されているコレクションを変更するには、admin ロールか、次のパーミッションが必要です。

- コレクションに対する更新パーミッション
- ドキュメントに対する更新パーミッション

保護されているコレクションに XPath 式でアクセスするには、コレクションに対する読み取りパーミッションが必要です。さらに、指定されたコレクションに属している個々のドキュメントへのアクセスは、個々のドキュメントの読み取り権限によって管理されます。ドキュメント自体へのアクセスは、コレクションのメンバーシップとは異なり、現在のユーザーのロールと、それぞれのドキュメントに関連付けられているパーミッションによって管理されることに注意してください。

セキュリティ関数ライブラリモジュール `sec:protect-collection` を使用すると、保護されていないコレクションを保護されているコレクションに変換できます。このライブラリモジュールにアクセスするには、`protect-collection` 権限が必要です。

セキュリティ関数ライブラリモジュール `sec:unprotect-collection` を使用すると、保護されているコレクションを保護されていないコレクションに変換できます。このライブラリモジュールにアクセスするには、`unprotect-collection` 権限と、保護されているコレクションに対する更新パーミッションが必要です。

21.6 パフォーマンス特性

MarkLogic のコレクションは、大量ドキュメントに対するクエリパフォーマンスを最適化するように設計されています。他の場合と同様に、これにはある程度のトレードオフが伴います。このセクションでは、コレクションのパフォーマンス特性の概要について簡単に説明します。このセクションは、次のパートから構成されます。

- ドキュメントが所属するコレクションの数
- コレクションへの既存のドキュメントの追加と削除

21.6.1 ドキュメントが所属するコレクションの数

ドキュメントの読み込み時に、コレクションの情報がドキュメントに埋め込まれて、データベースに格納されます。

この設計により、MarkLogic データベースは数百万個のコレクションでも問題なく処理できます。また、`collection()` 関数自体の大幅な効率化も実現し、インデックス操作 1 つだけで、大規模なデータセットをコレクション別にサブセット化できます。`collection()` 関数で複数のコレクションを指定する場合、指定するコレクションごとに、追加のインデックス操作が必要です。クエリで同様のコレクションを対象にする場合、このようなインデックス操作はキャッシュ内で解決され、極めて高いパフォーマンスが実現します。

この設計に伴うトレードオフの 1 つとして、単一のドキュメントが所属できるコレクションの数に実用上の制約があることが挙げられます。アーキテクチャによる制限はありませんが、各ドキュメントのコレクションの平均数が増えると、データベースのサイズは大きくなります。このようなデータベースの拡大は、個々のドキュメントフラグメントのサイズの拡大に起因しています。フラグメントのサイズが拡大するのは、ドキュメントが所属する各コレクションが、少量の情報をフラグメントに埋め込むからです。フラグメントが大きくなると、ストレージの I/O 時間も増加して、パフォーマンスが低下します。重要なポイントは、ドキュメントごとのコレクションの平均数による影響は、インデックス解決の時間としてではなく、ストレージからコンテンツ（フラグメント）を取得する時間にだけ現れるということです。

実際的なガイドラインとしては、フラグメントのサイズが平均 50K であるドキュメントでは、所属するコレクションの数を 100 個以下にすることをお勧めします。これにより、平均的なフラグメントサイズの拡大を 10% 未満に抑えられます。

21.6.2 コレクションへの既存のドキュメントの追加と削除

MarkLogic のコレクションのもう 1 つのトレードオフは、ドキュメントがすでにデータベースに存在している場合、コレクションへのドキュメントの追加や削除において消費されるリソースが相対的に大きいことです。ドキュメントが所属するコレクションを変更するには、ドキュメントのフラグメントをすべて書き換える必要があります。サイズの大きいドキュメントの場合、この処理には、CPU と I/O のリソースの両方が必要になります。コレクションのメンバーシップがアプリケーションで動的に変化する場合、代わりにドキュメント自体に含まれる要素でメンバーシップを指定したほうがいいでしょう。

22.0 シソーラス関数の使用

MarkLogic サーバーには、アプリケーションでシソーラス機能を利用するための関数が用意されています。シソーラスアプリケーションとは、シソーラス（類義語）ドキュメントを利用して、ユーザーが入力した単語に意味が似た単語を検索するものです。一般的なアプリケーションの例としては、ユーザーが入力した単語と似た意味を持つ単語も対象とするように検索を拡張します。例えば、*car* の類義語として車のブランドを列挙したシソーラスドキュメントを使用した場合、car を検索すると、*car* という単語の結果に加えて、*Alfa Romeo*、*Ford*、*Hyundai* などの結果が返されます。

この章では、シソーラス関数を使用する方法について説明します。この章は、次のセクションから構成されます。

- シソーラスモジュール
- 関数リファレンス
- シソーラススキーマ
- 大文字 / 小文字の区別
- シソーラスドキュメントの管理
- シソーラスによる検索の拡張

22.1 シソーラスモジュール

シソーラス関数は、次の XQuery モジュールファイルにインストールされています。

- *install_dir*/Modules/MarkLogic/thesaurus.xqy

ここで、*install_dir* は MarkLogic サーバーのインストール先のディレクトリです。シソーラスモジュールの関数は、thsr: という名前空間プレフィックスを使用します。この名前空間プレフィックスを XQuery プログラムで指定するか、独自の名前空間を指定します。関数を使用するには、次のように、XQuery プログラムのプロローグにモジュールと名前空間の宣言を含めます。

```
import module namespace thsr="http://marklogic.com/xdmp/thesaurus"
                            at "/MarkLogic/thesaurus.xqy";
```

22.2 関数リファレンス

シソーラスモジュール関数の参考情報は、developer.marklogic.com で入手できる『MarkLogic XQuery and XSLT Function Reference』に記載されています。

22.3 シソーラススキーマ

MarkLogic サーバーに読み込むシソーラスドキュメントは、次のファイルにインストールされているシソーラススキーマに準拠している必要があります。

- *install_dir*/Config/thesaurus.xsd

ここで、*install_dir* は MarkLogic サーバーのインストール先のディレクトリです。

22.4 大文字 / 小文字の区別

シソーラスドキュメントとシソーラス関数では、大文字と小文字が区別されます(case-sensitive)。したがって、シソーラスでは *Car* というタームは *car* というタームとは異なるものとして扱われます。このようなタームの参照では大文字と小文字が区別されます。

アプリケーションにおいて大文字と小文字を区別しない(つまり *Car* というタームで、シソーラスの *Car* と *car* という両方のエントリーを返す)ようにするには、アプリケーション側で、参照するタームの大文字と小文字に対処する必要があります。大文字と小文字に対処する方法はいくつかあります。例えばシソーラスドキュメントのエントリーをすべて小文字にしておき、参照するタームを小文字にしてから、シソーラスを参照します。シソーラスドキュメントのタームを小文字にする例については、「シソーラスドキュメントの挿入時のタームの小文字化」(427 ページ)を参照してください。

22.5 シソーラスドキュメントの管理

データベースに用意できるシソーラスドキュメントの数はいくつでも構いません。既存のシソーラスドキュメントのエントリーの追加や変更もできます。このセクションでは、シソーラスドキュメントを読み込んで更新する方法について説明します。このセクションは、次のトピックから構成されます。

- シソーラスドキュメントの読み込み
- シソーラスドキュメントの挿入時のタームの小文字化
- WordNet シソーラスの XML バージョンの読み込み
- シソーラスドキュメントの更新
- シソーラスドキュメントのセキュリティ上の考慮事項
- シソーラス管理関数を使用するクエリの例

22.5.1 シソーラスドキュメントの読み込み

クエリでシソーラスを使用するには、thsr:load 関数または thsr:insert 関数を使用して、ドキュメントをシソーラスとして読み込みます。例えば、URI が /myThsrDocs/wordnet.xml となるシソーラスドキュメントを読み込むには、次のようなクエリを実行します。

```
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")
```

この XQuery は、c:\thesaurus\wordnet.xml ファイルから、すべての <entry> 要素を URI が /myThsrDocs/wordnet.xml のシソーラスに追加します。このドキュメントがすでに存在している場合、指定したファイルの新しいコンテンツで上書きされます。

> **注：** 使用するシソーラスドキュメントのサイズが大きすぎてインメモリリストに収まらない場合、シソーラスを複数のドキュメントに分割できます。この場合、URI をパラメータとして取るシソーラス API で、すべてのシソーラスドキュメントを指定します。また、異なるシソーラスドキュメント間で重複するエントリーが存在しないことを確認します。

22.5.2 シソーラスドキュメントの挿入時のタームの小文字化

シソーラスドキュメントとして挿入する前に、thsr:insert 関数でドキュメントを変換できます。次の例は、xdmp:get 関数でドキュメントをメモリに読み込んでから、そのインメモリドキュメントを操作して、タームが小文字である新しいドキュメントを作成する方法を示しています。

```
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>
                    )
```

22.5.3 WordNet シソーラスの XML バージョンの読み込み

MarkLogic 開発者サイト（developer.marklogic.com）から *WordNet* の XML バージョンをダウンロードできます。シソーラスファイルをダウンロードしたら、`thsr:load` 関数を使用して、シソーラスドキュメントとして読み込むことができます。

次の手順に従って、WordNet シソーラスをダウンロードして読み込みます。

1. developer.marklogic.com の［*Workshop*］ページに移動します。

   ```
   http://developer.marklogic.com/code/default.xqy
   ```

2. シソーラスドキュメントのセクションに移動して、`thesaurus.xml` ドキュメントを検索します。

3. `thesaurus.xml` をファイル（例えば、`c:\thesaurus\thesaurus.xml`）に保存します。

4. 次のようなコマンドで、シソーラスを読み込みます。

   ```
   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")
   ```

この結果、URI が `/myThsrDocs/wordnet.xml` のシソーラスが読み込まれます。これで、この URI をシソーラスモジュール関数で使用できます。

22.5.4 シソーラスドキュメントの更新

次のシソーラス関数を使用して、既存のシソーラスドキュメントを修正します。

- `thsr:set-entry`
- `thsr:add-synonym`
- `thsr:remove-entry`
- `thsr:remove-term`
- `thsr:remove-synonym`

さらに、`thsr:insert` 関数を使用して、既存のシソーラスドキュメントにエントリーを追加できます。指定する URI にシソーラスドキュメントが存在しない場合は新規作成されます。

注： MarkLogic サーバーのトランザクション単位はクエリです。このため、同じシソーラスドキュメントに対して複数の更新を実行する場合は、それぞれの更新を個別のクエリとして実行してください。複数の更新ステートメントの間にセミコロンを挿入すると、新しいクエリ（つまり、新しいトランザクション）を開始できます。セミコロンを使用して、シソーラス関数を使用する新しいクエリを開始する場合、それぞれのクエリのプロローグに `import` ステートメントを含めて、シソーラスの名前空間を解決する必要があります。

22.5.5 シソーラスドキュメントのセキュリティ上の考慮事項

シソーラスドキュメントは、XML 形式でデータベースに格納されます。したがって、他のドキュメントと同様にクエリの対象になります。セキュリティとシソーラスドキュメントについては、次の点に注意してください。

- デフォルトでは、シソーラスドキュメントは次のコレクションに読み込まれます。
  - http://marklogic.com/xdmp/documents
  - http://marklogic.com/xdmp/thesaurus
- シソーラスドキュメントは、読み込んだユーザーのデフォルトのパーミッションで読み込まれます。シソーラスドキュメントの読み込みを実行するユーザーには、適切な権限が必要です。権限がなかった場合、読み取りや更新に必要なパーミッションをドキュメントに設定できません。詳細については、『Loading Content Into MarkLogic Server Guide』の「Setting Document Permissions」を参照してください。
- ドキュメントが読み込まれたときのデフォルトのパーミッションの範囲を超えて、シソーラスドキュメントへのアクセス（読み書き）を制御する場合、`thsr:load` 操作の後で、`xdmp:document-set-permissions` を実行します。

22.5.6 シソーラス管理関数を使用するクエリの例

このセクションでは、以下の各例について説明します。

- 例：新しいシソーラスエントリーの追加
- 例：シソーラスエントリーの削除
- 例：シソーラスからのタームの削除
- 例：シソーラスエントリーへの類義語の追加
- 例：シソーラスからの類義語の削除

22.5.6.1 例：新しいシソーラスエントリーの追加

次の XQuery では、`thsr:set-entry` 関数を使用して、*Car* というエントリーを URI が `/myThsrDocs/wordnet.xml` のシソーラスに追加しています。

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

thsr:set-entry("/myThsrDocs/wordnet.xml",
<entry xmlns="http://marklogic.com/xdmp/thesaurus">
    <term>Car</term>
    <synonym>
      <term>Ford</term>
      <part-of-speech>noun</part-of-speech>
    </synonym>
    <synonym>
      <term>automobile</term>
      <part-of-speech>noun</part-of-speech>
    </synonym>
    <synonym>
      <term>Fiat</term>
      <part-of-speech>noun</part-of-speech>
    </synonym>
</entry>)
```

`/myThsrDocs/wordnet.xml` シソーラスに同じエントリーが含まれている場合、シソーラスは変更されません。シソーラスに *car* というエントリーが含まれていない場合や、同一ではない（つまり、ノードが異なる）*car* というエントリーが含まれている場合は、新しいエントリーが追加されます。この新しいエントリーは、シソーラスドキュメントの末尾に追加されます。

22.5.6.2 例：シソーラスエントリーの削除

次の XQuery では、`thsr:remove-entry` 関数を使用して、*Car* というエントリーを URI が `/myThsrDocs/wordnet.xml` のシソーラスから削除しています。

```
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])
```

この結果、*Car* の 2 番目のエントリーが、シソーラスドキュメント `/myThsrDocs/wordnet.xml` から削除されます。

22.5.6.3 例：シソーラスからのタームの削除

次の XQuery では、thsr:remove-term 関数を使用して、*Car* というタームのすべてのエントリーを、URI が /myThsrDocs/wordnet.xml のシソーラスから削除しています。

```
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")
```

この結果、*Car* というタームがすべて、シソーラスドキュメント /myThsrDocs/wordnet.xml から削除されます。シソーラスに *Car* というタームが 1 つだけ含まれている場合、thsr:remove-term 関数は thsr:remove-entry 関数と同じ処理を実行することになります。

22.5.6.4 例：シソーラスエントリーへの類義語の追加

次の XQuery では、URI /myThsrDocs/wordnet.xml のシソーラスで、*car* というエントリーに、*Alfa Romeo* という類義語を追加しています。

```
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>)
```

このクエリは、シソーラスの *car* というエントリーを参照すると、エントリーが 1 つ返されるという想定です。*car* というエントリーを参照すると複数のエントリーが返される場合、エントリーを 1 つ指定する必要があります。例えば、シソーラスの最初の *car* というエントリーに類義語を追加する場合、次のように最初の引数を指定します。

```
thsr:lookup("/myThsrDocs/wordnet.xml", "car")[1]
```

22.5.6.5 例：シソーラスからの類義語の削除

次の XQuery では、URI が /myThsrDocs/wordnet.xml のシソーラスで、*car* というエントリーから、*Fiat* という類義語を削除しています。

```
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>)
```

このクエリは、シソーラスの *car* というエントリーを参照すると、エントリーが 1 つ返されるという想定です。*car* というエントリーを参照すると複数のエントリーが返される場合、エントリーを 1 つ指定する必要があります。例えば、シソーラスの最初の *car* というエントリーから類義語を削除する場合、次のように最初の引数を指定します。

```
thsr:lookup("/myThsrDocs/wordnet.xml", "car")[1]
```

22.6 シソーラスによる検索の拡張

検索を拡張して、検索で入力されたタームに加えて、シソーラスのタームを対象に含めることができます。例えば、以下のクエリについて考えてみましょう。

```
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"),
  (),
  (),
  () )
)
```

このクエリは、シェークスピアの「ハムレット」の全行を検索して、*weary* という単語か、*weary* という単語の類義語をすべて見つけます。

ただし、シソーラスのエントリーには、多数の類義語が設定されている場合があります。その場合、検索を拡張する場合には、`thsr:expand` が返すリストにおいて目的の類義語をユーザーが指定できる、フォーム形式のユーザーインターフェイスをアプリケーション側で作成しなければなりません。検索の対象に含める類義語をユーザーが選択すると、アプリケーションがそのタームを検索対象に追加して、データベースに送ります。

23.0 スペリング修正関数の使用

MarkLogic サーバーには、アプリケーションでスペリング修正機能を利用できる関数が用意されています。スペリングアプリケーションは、辞書ドキュメントを使用して、ユーザーが入力する単語のスペルミスを見つけます。一般的なアプリケーションの例として、スペリングが間違っている可能性がある単語をユーザーに提示できます。例えば、ユーザーが *albetros* という単語を検索条件に入力した場合に、スペリング修正関数を使用するアプリケーションでは、本当は *albatross* と入力するつもりではなかったのかとユーザーに確認します。

この章では、スペリング修正関数を使用する方法について説明します。この章は、次のセクションから構成されます。

- スペリング修正の概要
- スペリング辞書管理モジュール関数
- 関数リファレンス
- 辞書ドキュメント
- 大文字 / 小文字の区別
- 辞書ドキュメントの管理
- 単語のスペルが正しいかどうかのテスト
- 単語のスペルが正しくない場合のスペルの候補の取得

23.1 スペリング修正の概要

スペリング修正関数を使用すると、単語のスペルが正しいかどうかをチェックするアプリケーションを作成できます。ここでは、データベースに読み込んである 1 つあるいは複数の辞書のうち、指定された辞書を使って単語をチェックできます。辞書にどのような単語を含めるかについては、完全に制御できます。関数としては、辞書、スペルチェック、およびスペルミス候補の単語表示の管理に対応するものが用意されています。

23.2 関数リファレンス

スペリングモジュール関数のリファレンス情報は、docs.marklogic.com で入手できる『MarkLogic XQuery and XSLT Function Reference』に記載されています。スペリング関数は、次のカテゴリに分類されます。

- スペリングビルトイン関数
- スペリング辞書管理モジュール関数

23.2.1 スペリングビルトイン関数

スペリング修正関数はビルトイン関数であり、XQuery プロローグに `import module` ステートメントを必要としません。スペリング修正関数には以下のものがあります。

- `spell:is-correct`
- `spell:suggest`
- `spell:suggest-detailed`
- `spell:double-metaphone`
- `spell:levenshtein-distance`

`spell:double-metaphone` 関数および `spell:levenshtein-distance` 関数は生の値を返し、この値から、`spell:suggest`、`spell:suggest-detailed`、および `spell:is-correct` の値が計算されます。

`spell:suggest` と `spell:suggest-detailed` の違いとしては、`spell:suggest-detailed` は候補の演算に使用された情報の一部を提供し、XML 表現として返すことが挙げられます。これに対して、`spell:suggest` は、候補の単語のシーケンスを返します。ほとんどのスペリングアプリケーションでは、`spell:suggest` で十分ですが、用意する候補を細かく調整する場合（例えば、候補を返す順序を独自に計算する場合）、`spell:suggest-detailed` を使用して、XML 出力として返された基準の一部を使ってフィルタリングができます。

23.2.2 スペリング辞書管理モジュール関数

辞書ドキュメントを管理する XQuery モジュールが用意されています。スペリング修正辞書管理関数は、次の XQuery モジュールファイルにインストールされています。

- *`install_dir`*`/Modules/MarkLogic/spell.xqy`

ここで、*`install_dir`* は MarkLogic サーバーのインストール先のディレクトリです。スペリングモジュールの関数は、サーバーで事前定義された `spell:` という名前空間プレフィックスを使用します。このモジュールの関数を使用するには、次のように、XQuery プログラムのプロローグにモジュール宣言を含めます。

```
import module "http://marklogic.com/xdmp/spell" at "/MarkLogic/spell.xqy";
```

23.3 辞書ドキュメント

MarkLogic サーバーに読み込む辞書ドキュメントは、次のような基本構造を備えている必要があります。

```
<dictionary xmlns="http://marklogic.com/xdmp/spell">
    <metadata>
    </metadata>
    <word></word>
    <word></word>
    ......
</dictionary>
```

サンプル辞書ドキュメントが docs.marklogic.com で公開されています。このようなドキュメントを使用するか、独自の辞書を作成できます。`spell:make-dictionary` スペリング管理関数を使用して、辞書ドキュメントを作成し、`spell:load` を使用して、辞書をデータベースに読み込みます。

23.4 大文字 / 小文字の区別

スペリング参照関数（`spell:is-correct`、`spell:suggest`、および `spell:suggest-detailed`）では大文字と小文字が区別されるので、辞書内の単語では大文字と小文字の区別が重要です。さらに、スペリング参照では最初の文字の処理に、特別なルールが用意されています。次に示すのは、スペリング修正関数の大文字 / 小文字対応のルールです。

- スペリング参照クエリでは、最初の大文字が `spell:is-correct` でスペルミスとされることはありません。例えば、*Word* は、辞書の *word* というエントリーとマッチします。
- ただし、辞書で最初の文字が大文字に設定されている場合、`spell:is-correct` では、完全一致だけが正しいものとして処理されます。例えば、辞書に含まれている単語が *Word* である場合、*word* は誤りとされます。
- 辞書で単語の他の文字が大文字に設定されている場合、`spell:is-correct` では、完全一致（単語の最初の文字の大文字 / 小文字を除く）だけがマッチされます。例えば、*word* は *woRd* というエントリーとはマッチしません。同様に、*WOrd* もマッチしませんが、*WoRd* はマッチします。
- `spell:suggest` 関数（および `spell:suggest-detailed` 関数）では、大文字 / 小文字については最初の文字だけが確認されます。例えば、`spell:suggest("THe")` は *The*、*Thee*、*They* などを候補として返すのに対して、`spell:suggest("tHe")` は *the*、*thee*、*they* などを返します。つまり、`spell:suggest` 関数に対する引数の最初の文字を大文字に設定すると、必ず大文字で始まる候補が返されます。それ以外の場合は、小文字の候補が返されます。

アプリケーションで大文字と小文字の区別を無視する場合、すべて小文字の単語で辞書を作成しておきます。例えば、XQuery `fn:lower-case` 関数を使用して、`spell:is-correct` 関数および `spell:suggest` 関数の単語の引数をすべて小文字に設定してから、クエリを送る必要があります。

23.5 辞書ドキュメントの管理

任意の数の辞書ドキュメントをデータベースに用意できます。既存の辞書ドキュメントのエントリーの追加または変更もできます。このセクションでは、辞書ドキュメントを読み込んで更新する方法について説明します。このセクションは、次のトピックから構成されます。

- 辞書ドキュメントの読み込み
- サンプル XML 辞書の読み込み
- 辞書ドキュメントの更新
- 辞書ドキュメントのセキュリティ上の考慮事項

23.5.1 辞書ドキュメントの読み込み

クエリで辞書を使用するには、`spell:load` 関数または `spell:insert` 関数を使用して、ドキュメントを辞書として読み込みます。例えば、URI が `/mySpell/spell.xml` の辞書ドキュメントを読み込むには、次のようなクエリを実行します。

```
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")
```

この XQuery は、`c:\dictionaries\spell.xml` ファイルから、すべての `<word>` 要素を URI が `/mySpell/spell.xml` の辞書に追加します。このドキュメントがすでに存在している場合、指定したファイルの新しいコンテンツで上書きされます。

23.5.2 サンプル XML 辞書の読み込み

MarkLogic のコミュニティサイト（developer.marklogic.com/code#dictionaries）からサンプル辞書をダウンロードできます。このコミュニティサイトは、github にリンクしています。github には、大中小の各バージョンの辞書が用意されています。辞書の XML ファイルをダウンロードしたら、`spell:load` 関数を使用して、辞書ドキュメントとして読み込むことができます。

次の手順に従って、サンプル辞書をダウンロードして読み込みます。

1. developer.marklogic.com の［*Code*］ページに移動します。

   http://developer.marklogic.com/code/#dictionaries

2. 辞書ドキュメントのセクションに移動して、github のリンクをクリックします。

   ```
   https://github.com/marklogic/dictionaries
   ```

3. dictionaries フォルダで、`small-dictionary.xml`、`medium-dictionary.xml`、または `large-dictionary.xml` のいずれかのファイル（あるいは、利用可能なその他の辞書ドキュメント）を選択します。large-dictionary には、約 10 万個の単語が含まれており、ダウンロードするファイルのサイズは約 3MB です。

4. `<size>-dictionary.xml` をファイル（例えば、`c:\dictionaries\spell.xml`）に保存します。

5. 次のようなコマンドで、この辞書を読み込みます。

   ```
   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")
   ```

この結果、URI が `/mySpell/spell.xml` の辞書が読み込まれます。これで、この URI をスペリング修正モジュール関数で使用できます。

23.5.3 辞書ドキュメントの更新

次の辞書関数を使用して、既存の辞書ドキュメントを修正します。

- `spell:add-word`
- `spell:remove-word`

`spell:insert` 関数は、既存の辞書ドキュメントを指定した場合、既存の辞書を上書きします。指定する URI に辞書ドキュメントが存在しない場合は新規作成されます。

> **注：** MarkLogic サーバーのトランザクション単位はクエリです。このため、同じ辞書ドキュメントに対して複数の更新を実行する場合は、それぞれの更新を個別のクエリとして実行してください。複数の更新ステートメントの間にセミコロンを挿入すると、新しいクエリ（つまり、新しいトランザクション）を開始できます。セミコロンを使用して、スペリング修正関数を使用する新しいクエリを開始する場合、それぞれのクエリのプロローグに `import` ステートメントを含めて、スペリングモジュールを解決する必要があります。

23.5.3.1 例：辞書への新しい単語の追加

次の XQuery では、`spell:add-word` 関数を使用して、*albatross* というエントリーを URI が `/mySpell/Spell.xml` の辞書に追加しています。

```
xquery version "1.0-ml";
import module "http://marklogic.com/xdmp/spell" at
              "/MarkLogic/spell.xqy";

spell:add-word("/mySpell/spell.xml", "albatross")
```

`/mySpell/spell.xml` 辞書に同じエントリーが含まれている場合、辞書は変更されません。含まれていない場合は、*albatross* というエントリーが辞書に追加されます。

23.5.3.2 例：辞書からの単語の削除

次の XQuery では、`spell:remove-word` 関数を使用して、*albatross* というエントリーを URI が `/mySpell/Spell.xml` の辞書から削除しています。

```
xquery version "1.0-ml";
import module "http://marklogic.com/xdmp/spell" at
              "/MarkLogic/spell.xqy";

spell:remove-word("/mySpell/spell.xml", "albatross")
```

この結果、*albatross* という単語が `/mySpell/spell.xml` 辞書ドキュメントから削除されます。

23.5.4 辞書ドキュメントのセキュリティ上の考慮事項

辞書ドキュメントは、XML 形式でデータベースに格納されます。したがって、他のドキュメントと同様にクエリの対象になります。セキュリティと辞書ドキュメントについては、次の点に注意してください。

- デフォルトでは、辞書ドキュメントは次のコレクションに読み込まれます。
  - `http://marklogic.com/xdmp/documents`
  - `http://marklogic.com/xdmp/spell`
- 辞書ドキュメントは、読み込んだユーザーのデフォルトのパーミッションで読み込まれます。辞書ドキュメントの読み込みを実行するユーザーには、適切な権限が必要です。権限がなかった場合、読み取りや更新に必要なパーミッションをドキュメントに設定できません。詳細については、『Loading Content Into MarkLogic Server Guide』の「Setting Document Permissions」を参照してください。
- ドキュメントが読み込まれたときのデフォルトのパーミッションの範囲を超えて、辞書ドキュメントへのアクセス（読み書き）を制御する場合、`spell:load` 操作の後で、`xdmp:document-set-permissions` を実行します。

23.6 単語のスペルが正しいかどうかのテスト

`spell:is-correct` 関数テストを使用して、指定する辞書に従って、単語のスペルが正しいかどうかを確認できます。例えば、以下のクエリについて考えてみましょう。

```
spell:is-correct("/mySpell/spell.xml", "alphabet")
```

alphabet という単語のスペルは正しいので、このクエリは true を返します。今度は、以下のクエリについて考えてみましょう。

```
spell:is-correct("/mySpell/spell.xml", "alfabet")
```

alfabet という単語のスペルは正しくないので、このクエリは `false` を返します。

23.7 単語のスペルが正しくない場合のスペルの候補の取得

指定する辞書の単語に基づいてスペルの候補を返すクエリを作成できます。例えば、以下のクエリについて考えてみましょう。

```
spell:suggest("/mySpell/spell.xml", "alfabet")
```

このクエリは、次の結果を返します。

```
alphabet albeit alphabets aloft abet alphabeted affable alphabet's
alphabetic offbeat
```

この結果はランク付けされており、最初の単語は実際のスペルである可能性が最も高いものです。アプリケーションにおいて、候補のいずれかの単語が、意図していた単語であるかどうか、ユーザーに確認を促すことができます。

今度は、以下のクエリについて考えてみましょう。

```
spell:suggest("/mySpell/spell.xml", "alphabet")
```

このクエリは、単語のスペルが正しいことを示す、空のシーケンスを返します。

注： スペリング修正関数が候補を提示できる単語は、64 文字未満のものだけです。また、スペリング修正関数が返す候補は、64 文字未満です。

24.0 distinctive terms と cts:similar-query

MarkLogic サーバーには、cts:similar-query と cts:distinctive-terms が含まれています。これらの search API では、検索の観点から、通常は検索結果でノードの特徴（distinctive であること）を検索できます。この章では、cts:similar-query と cts:distinctive-terms について説明します。この章は、次のセクションから構成されます。

- cts:similar-query の概要
- 一連のノードに含まれている distinctive terms の検索
- cts:distinctive-terms の出力の概要
- デザインパターンの例：タグクラウドの作成

24.1 cts:similar-query の概要

cts:similar-query を使用して、検索の観点から、最初のパラメータで渡すモデルノードと類似したノードを検索できます。cts:similar-query コンストラクタは cts:query コンストラクタの一種であり、「cts:query 式の構成」（ 239 ページ）の説明に従って、他の cts:query コンストラクタと組み合わせることができます。

他の cts:query コンストラクタのように、インデックスを参照して、クエリとマッチするタームを検索する代わりに、cts:similar-query は、渡されたノードに対してインデックス化プロセスを実行します。そして、高い関連度でモデルノードとマッチする cts:query を返します。さまざまなインデックスオプションやスコアオプションを cts:similar-query に渡して、そこから生成される cts:query に影響を与えることができます。

生成されるクエリは、*データベース内の他のドキュメントに基づいて*モデルノードの distinctive terms を検索します。

24.2 一連のノードに含まれている distinctive terms の検索

cts:similar-query が cts:query の生成に使用されるタームを検索するには、cts:distinctive-terms を使用します。cts:distinctive-terms の出力は、いくつかの cts:term の子を含む cts:class 要素です。それぞれの cts:term 要素には、タームを表す cts:query コンストラクタが含まれています。それぞれの cts:term 要素には、そのタームのスコアと信頼度も含まれています。MarkLogic サーバーでは、このスコアを関連度の計算に使用します。

さまざまな種類のオプションを cts:distinctive-terms に渡して、生成されるタームを制御できます。データベースオプションで、モデルノードに対して最も関連度が高くなるタームを制御できます。これにより、cts:distinctive-terms の出力に影響を及ぼします。反復的なアプローチを使用する場合、さまざまなインデックス化オプションを試して、モデルノードに対して最適な結果が得られるのはどのオプションかを確認できます。

distinctive terms は、データベース内の他のドキュメントに基づいて生成または区別されるので、大規模なデータベースに対して実行したほうが、良い結果が得られます。

24.3 cts:distinctive-terms の出力の概要

次に示すのは、シンプルな cts:distinctive-terms クエリとその出力です。

```
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>
```

この出力は cts:class 要素であり、それぞれの子は cts:term 要素です。cts:term 要素は、cts:query によって識別されるデータベース内のタームを表します。それぞれのタームには、val、score、confidence、および fitness の数値があります。

val 属性と score 属性は、そのタームのスコアに対する影響度の概算値です。confidence 属性は、タームの cts:confidence 値を表します。fitness 属性は、タームの cts:fitness 値を表します。スコア（score）、適合度（fitness）、および信頼度（confidence）の詳細については、「関連度スコア：概要とカスタマイズ」（ 289 ページ）を参照してください。

前のクエリでは、ワードクエリのタームだけを考慮していました。要素内のタームや、語のペアであるターム（cts:near-query の distance が 1 という条件）に対しては、cts:element-word-query のタームや cts:near-query のタームも使用できます。この種のタームを確認するには、次のようなクエリを実行してみます。

```
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>)
```

このクエリでは、db:fast-element-word-searches および db:fast-element-phrase-searches オプションを有効にして、特定の要素に制約されているタームを出力に表示します。データベースのオプションを cts:distictive-terms に変更して、出力の違いを確認してみると、インデックスオプションによる distinctive terms への影響がわかります。また、cts:similar-query では、この同じ設定を使用できるので、ドキュメントがモデルノードと類似しているかどうかを cts:similar-query でどのように判別しているのかもわかります。

24.4 デザインパターンの例：タグクラウドの作成

タグクラウドは、通常は検索に関連する、さまざまなタームを示す視覚化の手法として一般的であり、大きいサイズのフォントや目立つ色のフォントで関連度の高い語を示します。cts:distinctive-terms を使用して、タグクラウド作成用のデータを提供できます。基本的なデザインパターンは、次のとおりです。

- さまざまなオプションを試しながら、満足のいく結果が得られる cts:distinctive-terms クエリを作成します。
- タグクラウドに使用するターム数に合わせて、max-terms サイズを設定します。
- スコア（あるいは適合度）をフォントサイズに変換するアルゴリズムを用意します。例えば、適合度に 20 を乗算した値をフォントサイズに使用します。
- 上記のアルゴリズムに基づいて結果全体を反復処理して、タグクラウドを作成する html を生成します。

次のサンプルコードは、このデザインパターンの簡単な例です。

```
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>
```

上記のクエリから返された html をブラウザで表示した場合、100 語の distinctive terms が表示され、そのうち最も関連度の高いものが大きいフォントで表示されます。

25.0 分類子の学習

MarkLogic サーバーは、XML サポートベクターマシン（SVM）分類子（classifier）を備えています。この章では、分類子について、またコンテンツで分類子を使用する方法について説明します。この章は、次のセクションから構成されます。

- 学習と分類の仕組みの概要
- 分類子 API
- 分類子による XML の活用
- 学習用セットの作成
- クラスごとのしきい値を判断する方法
- 例：分類子の学習と実行

25.1 学習と分類の仕組みの概要

*分類子*は、*クラス*（ノードのカテゴリ）を定義するための一連の API です。分類子を通してクラスのサンプルを実行して、特定のクラスを構成する要素について分類子に学習させます。その後、この学習済みの分類子を不明なドキュメントやノードに対して実行して、それぞれの所属クラスを判別できます。分類のプロセスでは、MarkLogic サーバーの全文インデックス化機能ならびに XML 認識を利用して、学習用コンテンツのタームに統計分析を実行してクラスのメンバーシップを判別します。このセクションでは、分類子の概念について説明します。このセクションは、次のパートから構成されます。

- 学習と分類
- XML SVM 分類子
- クラスのハイパープレーンとしきい値
- 分類子の学習用コンテンツ

25.1.1 学習と分類

分類子を使用する際には、学習と分類という 2 つの基本的なステップがあります。学習とは、特定のクラスに属していることが既知のコンテンツを受け取り、これに基づいて分類子を作成するプロセスです。分類とは、このような学習用コンテンツセットから作成された分類子を未知のコンテンツに対して実行して、そのコンテンツがどのクラスに属するのか（クラスのメンバーシップ）を判別するプロセスです。学習のプロセスは反復的なプロセスで、できるだけ優れた分類子を作成できるように、繰り返し実行されます。これに対して、分類は 1 回限りのプロセスで、不明なコンテンツに対して実行されます。

25.1.2 XML SVM 分類子

MarkLogic サーバーの分類子は、サポートベクターマシン（SVM）を実装します。SVM 分類子は、学習用データに基づいて特定のクラスのメンバーシップを判別する有名なアルゴリズムを使用します。SVM 分類子の数学的根拠については、`svm classifier` で web 検索するか、Wikipedia の情報を参照してみてください。

基本的に、分類子はクラスの既知の例である一連の学習用コンテンツに対して統計分析を実行します。その後、学習用コンテンツから得られた知見に基づいて、他の未知なコンテンツの所属クラスを判別します。分類子を使用することで、学習時の統計分析に基づいて、コンテンツに関する知見が得られます。

従来の SVM 分類子では、サポートベクターマシンの演算への入力としてタームの頻度を使用して、統計分析を実行します。MarkLogic の XML SVM 分類子は、MarkLogic サーバーの XML を認識する全文インデックス化機能を利用します。したがって、分類子への入力として使用されるタームには、コンテンツ（単語など）、構造の情報（要素など）、またはコンテンツと構造の組み合わせ（要素と単語の関係など）が含まれます。タームに影響を及ぼす MarkLogic サーバーのインデックスオプションはすべて、分類子 API でオプションとして使用できます。したがって、さまざまなインデックス技術を使用して、サンプルコンテンツに対して最適に動作するように分類子をチューニングできます。

最初に、一連の学習用コンテンツに対してクラスを定義します。分類子は、このクラスに基づいて、他のコンテンツを分析して、分類します。分類子でコンテンツを分析するときには、時に矛盾する次の 2 種類の値が使用されます。これを使って新しいコンテンツ内の情報がクラスに属しているのかどうかを判別します。

- *精度（precision）*：クラスに属していると分類された要素が、本当にそのクラスに属している確率。精度が高い場合、他のクラスの他の結果のタームに似ている、一部の結果を見逃している可能性があります。
- *再現度（recall）*：実際にクラスに属している項目がそのクラスに属しているものとして分類される確率。再現度が高い場合、対象クラスの結果にタームが似ている他のクラスの結果が含まれる可能性があります。

分類子を調整するときに、精度と再現度の適切なバランスを見つける必要があります。このバランスは、アプリケーションの目的や要件によって異なります。例えば、コンテンツのトレンドを見つけたい場合、精度が高いほうがいいでしょう。これは無関係なノードが分析に含まれないようにするためです。これに対して、ある分類のすべてのインスタンスを特定する必要がある場合は、再現度が高いほうがいいでしょう。これはアプリケーションの目的上、メンバーを見逃したくないからです。ほとんどのアプリケーションでは、この両者のバランスをとる必要があります。この最適な値は、分類子の学習プロセスにおいて、トレードオフを考慮しながらアプリケーションに適したものを判断します（学習用コンテンツセットに基づいて）。

25.1.3 クラスのハイパープレーンとしきい値

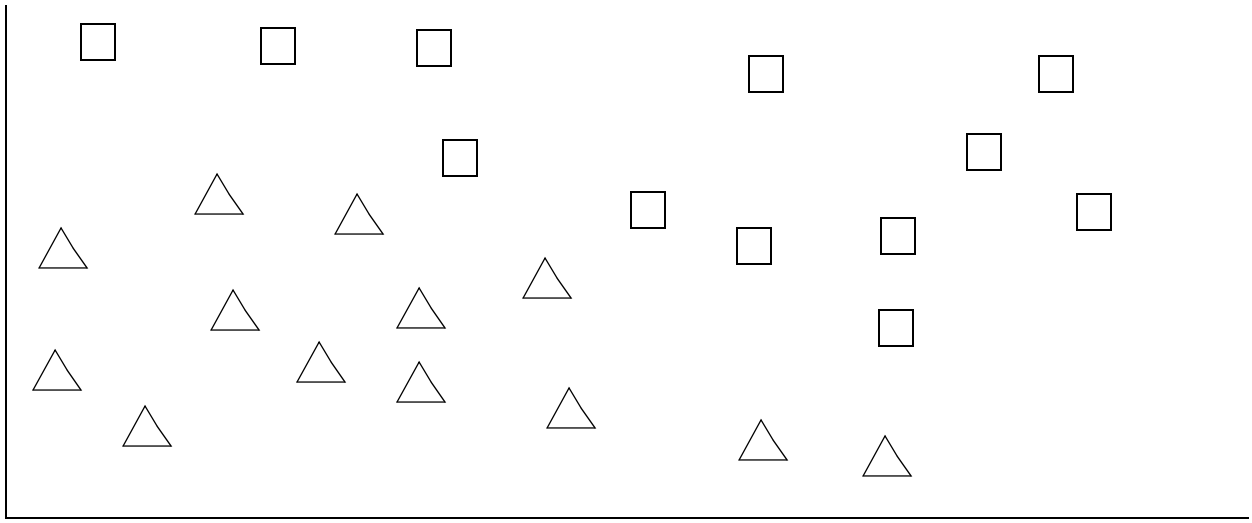
XML SVM 分類子の後ろでは、主に次の 2 種類の処理が実行されます。

- 各クラス間の境界を決定します。この処理は、学習中に行われます。
- クラスのメンバーシップの判別時に最も明確な結果を返す境界のしきい値を決定します。

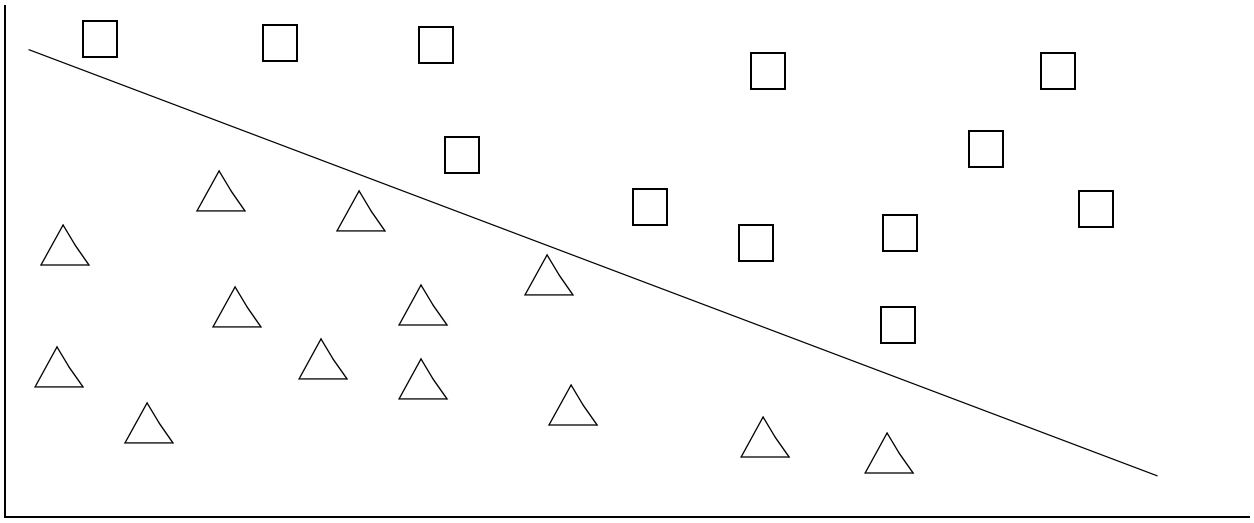
クラスの数は任意です。*タームベクター*は、インデックスオプションの定義に従って、ノード内のすべてのタームを表現します。したがってクラスは、類似性が高い一連のタームベクターから構成されます。

ここで、各タームが次元を形成すると考えてみます。例えば、クラスを 2 次元（x-y グラフ）や 3 次元（幅、奥行き、高さ）で視覚化するのは簡単です。しかし、4 次元以上では視覚化が難しくなります。このような場合に役立つのが、*ハイパープレーン*という概念です。

ハイパープレーンについて詳しく見ていく前に、クラスが 2 つあるコンテンツ（1 つは正方形、もう 1 つは三角形セット）について考えてみましょう。次の図では、正方形と三角形はそれぞれのクラスのメンバーであるタームベクターを表しています。

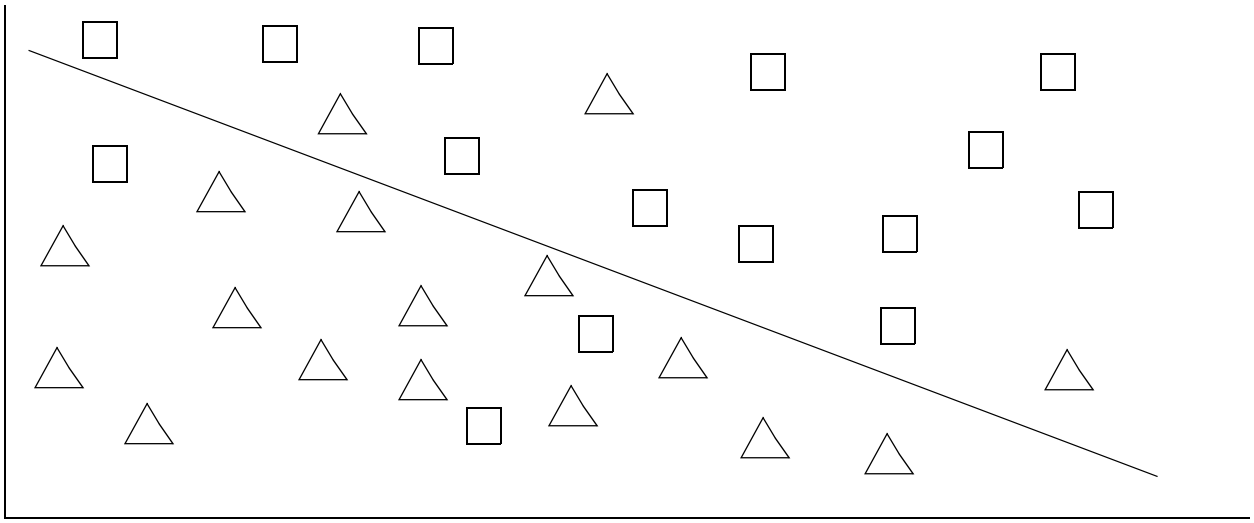


ここで、正方形と三角形を区分する線を引いてみます。この場合、次のような線で 2 つのクラスをきれいに区分できます。

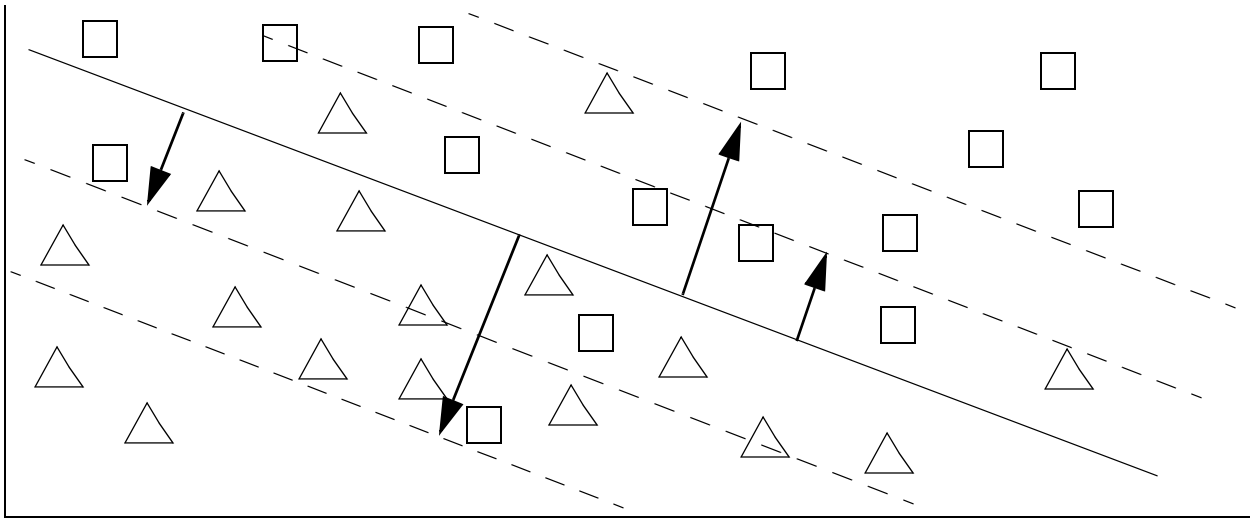


これが 3 次元の場合、線ではなくプレーン（面）でクラスを分けます。4 次元以上の場合、このプレーンを拡張したものを*ハイパープレーン*と呼びます。これは、クラスの境界（「エッジ」とも呼ばれます）を一般化した表現です。

上記の例は若干単純化されています。というのも、一方のクラスがもう一方と完全に区分されるように、ハイパープレーンを描けるようになっているからです。現実の世界においてはほとんどのコンテンツにおいて、次のようにそれぞれのクラスの一部のメンバーが境界の反対側にも存在しています。



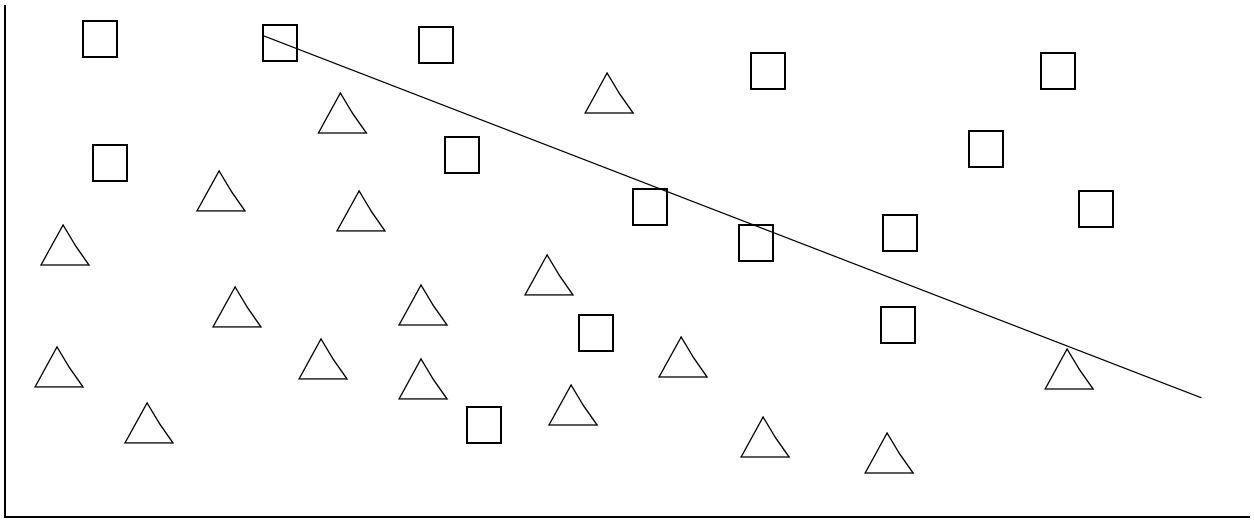
このような場合、境界に対して平行な線（*n* 次元の場合は、ハイパープレーン）を引いてみます。これらの線は、クラスの*しきい値*を表します。境界線としきい値の線の間の距離がしきい値を表します。しきい値は負数であり、クラスの外れ値（メンバー）が境界からどの程度離れているかを示します。次の図は、このしきい値を表しています。



点線は、いずれかの有効なしきい値を表しています。線が境界に近いほど、精度が高い(「完璧」ではない）しきい値を表しています。これに対して、線が境界から遠いほど、再現度が高くなります。三角形のクラスのメンバーが、正方形のクラスの境界側に存在している場合、そのメンバーは三角形のクラスには属していないことになります。ただし、選択したしきい値の範囲内に存在している場合、そのメンバーはクラスに属していると判別されます。

`cts:thresholds` という分類子 API を使用すると、学習用コンテンツセットに適したしきい値を見つけやすくなります。このため、分類子に対して未知のコンテンツを実行してクラスのメンバーシップを判別するときに、精度と再現度の適切なバランスが得られます。

次の図は、三角形クラスの境界を示しています（しきい値の線の下側が三角形クラス)。しきい値に基づく精度と再現度の値も示しています。



三角形の精度 = 16/25 = .64
三角形の再現度 = 16/17 = .94

25.1.4 分類子の学習用コンテンツ

コンテンツに最適なしきい値を見つけるには、すべてのクラスのメンバーを表現しているサンプルコンテンツで、分類子に*学習*させる必要があります。学習の質は、分類の質に直接影響を及ぼします。したがって、適切な学習用サンプルを見つけることが非常に重要です。

各クラスのサンプルは統計的に意味がある必要があり、クラスにちゃんと属しているもの（クラスの境界からしきい値の正側にあるサンプル）とクラスの境界に近いサンプルの両方を用意する必要があります。境界に近いサンプルは、コンテンツに最適なしきい値を決定する際に役立つので、特に重要です。学習用セットとしきい値の設定の詳細については、「学習用セットの作成」（452 ページ）および「クラスごとのしきい値を判断する方法」（454 ページ）を参照してください。

25.2 分類子 API

分類子には、3 つの XQuery ビルトイン関数が用意されています。このセクションでは、API の概要を紹介するとともに、API のいくつかの機能についても説明します。このセクションは、次のパートから構成されます。

- XQuery ビルトイン関数
- データは任意の場所に格納することも、作成することも可能
- API は多様なチューニングが可能
- 分類子の supports 形式と weights 形式の比較
- カーネル（マッピング関数）

- 精度と再現度のバランスを取るしきい値を見つける

分類子 API のシンタックスおよび使用方法の詳細については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

25.2.1 XQuery ビルトイン関数

分類子 API には、次の 3 つの XQuery 関数が含まれています。

- `cts:classify`
- `cts:thresholds`
- `cts:train`

これらの関数は、学習用ノードを受け取り、分類子の計算に使用します。分類子の仕様の作成は反復的なプロセスです。具体的には、学習用コンテンツの作成、学習用コンテンツによる分類子の学習（`cts:train`）、他の学習用コンテンツでの分類子のテスト（`cts:classify`）、学習用コンテンツでのしきい値の計算（`cts:threshold`）を実行します。満足のいく結果が得られるまで、このプロセスを繰り返します。分類子 API のシンタックスおよび使用方法の詳細については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

25.2.2 データは任意の場所に格納することも、作成することも可能

分類子 API はノードと要素を受け取ります。この際、分類または学習するノードのデータを Xquery で作成することや、データをデータベースなどの場所に格納できます。API はパラメータとしてノードを受け取るので、学習 / 分類用データを格納する方法には、さまざまな選択肢があります。

> **注：** ただし例外として、分類子の `supports` 形式を使用する場合は、学習用データがデータベース内に存在している必要があります。そして未知のコンテンツを分類するとき（つまり、`cts:classify` を実行するとき）に、これを学習用ノードに渡す必要があります。

25.2.3 API は多様なチューニングが可能

分類子 API にはさまざまなオプションが用意されており、多様なチューニングが可能です。`cts:train` では、さまざまなインデックスオプションやカーネルタイプを選択できます。また、制限やしきい値を指定できます。`cts:train` のカーネルタイプを変更すると、分類の結果やパフォーマンスに影響があります。分類は、反復的なプロセスです。自分で用意したコンテンツセットで試行錯誤することで、分類子から得られる結果を改善していきます。反復の際に、いくつかのパラメータを変更することで、どのような設定がコンテンツの分類を改善するのかを確認できます。

次のセクションでは、分類子の supports 形式と weights 形式の違いについて説明します。分類子の各オプションの機能の詳細については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

25.2.4 分類子の supports 形式と weights 形式の比較

分類子には、次の 2 種類の形式があります。

- `supports`：より洗練された方法でカーネルを使用します。学習用セット内の特定ドキュメントを参照して分類子をエンコードします。このため学習用ドキュメント全体を分類に使用できるので、より正確になります。ただし分類の際には、学習用セット全体が使用可能であるとともに、データベースに格納されている必要があります。さらに、タームベクターの作成はインデックス化とまったく同じものです。つまり分類子が呼び出されるたびに、学習用セット全体に対してインデックスタームが再生成されます。その一方で、分類子の実際の表現（`cts:train` から返される XML）は、かなりコンパクトである可能性があります。これ以外にも、`supports` 形式の分類子の利点として、特定の学習用ドキュメントに関するエラー推定が挙げられます。このようなエラーは、誤分類や、他のパラメータが最適な値に設定されていないせいである可能性があります。
- `weights`：それぞれのタームの重みをエンコードします。数学的な理由により、多くの問題で最適な結果が得られる Gaussian カーネルや Geodesic カーネルでは使用できません。学習用セットのすべてのタームに重みがない場合（タームの圧縮による）もあるので、この形式の分類子は根本的に精度が劣ります。クラスとタームが数多く存在する場合、分類子の表現自体も大きくなる可能性があります。ただし、分類の際に学習用セットを用意したり、そこからタームベクターを作成する（つまり、インデックスタームを再生成する）必要がないので、`weights` 形式の分類子のほうが、`cts:classify` は速くなります。

どちらの形式を使用するかは、いくつかの質問に対する回答や、基準によって異なります。具体的には、パフォーマンス（`supports` 形式では、時間やリソースがかかりすぎるか）、正確さ（`weights` 形式から得られる結果に満足できるか）、そして、それぞれの形式で試行錯誤する際の他の要因が挙げられます。一般に、分類子はさまざまなチューニングが可能です。使用データにおいて最適な結果を得るには、学習用データに何を使うのか、また分類にどのオプションを使うのかの両方に関して反復的なプロセスが必要です。

25.2.5 カーネル（マッピング関数）

学習の際には、さまざまなカーネルを選択できます。カーネルはマッピング関数であり、クラスのエッジからタームベクターまでの距離の判別に使用されます。各カーネルマッピング関数の詳細については、『MarkLogic XQuery and XSLT Function Reference』の `cts:train` についての記述を参照してください。

25.2.6 精度と再現度のバランスを取るしきい値を見つける

学習により分類子の仕様を作成する反復プロセスにおける最優先の目的の 1 つは、実際のクラスやコンテンツセットに最適なしきい値を見つけることです。精度（precision）と再現度（recall）のバランスがいいしきい値を見つけることが理想です（精度と再現度の詳細については、「XML SVM 分類子」（445 ページ）を参照）。`cts:thresholds` 関数を使用して、学習用セットに基づいたしきい値を計算します。適切なしきい値を見つけるための反復プロセスの概要については、「クラスごとのしきい値を判断する方法」（454 ページ）を参照してください。

25.3 分類子による XML の活用

分類子は XQuery のコンテキストに基づいて機能し、MarkLogic サーバーに組み込まれているので、本質的に XML 対応であると言えます。これには、さまざまな利点があります。必要に応じて分類が基づく特定の要素や要素階層（またそれ以上に複雑な XML 構造）を選択できます。そしてこの分類子を類似する他の要素や要素階層、さらにはまったく異なる一連の要素や要素階層に対して使用できます。XML ベースの検索を実行して、最適な学習用データを見つけることができます。コンテンツに XML 構造がある場合、その構造を分類子で活用できます。

例えば記事を分類する場合、記事の `<executive-summary>` 部分だけを使って分類できます。これによりコンテンツの他のセクションへの参照を除外できます。またこのセクションは、記事の詳細な本文よりも一般的な文体や用語が使われています。このアプローチでは、それぞれの記事のトピックに関して重要なタームを使用して、クラスのメンバーシップが判別されます。

25.4 学習用セットの作成

このセクションでは、分類子の作成に使用する学習用コンテンツセットについて説明します。このセクションは、次のパートから構成されます。

- 学習用セットの重要性
- 学習用セットのラベルの定義

25.4.1 学習用セットの重要性

分類の質は、分類子の学習用セットに左右されます。学習用のサンプルノードを選択する際には、明らかにクラスに属しているサンプルだけでなく、クラスのエッジに近いサンプル（クラスに属するものと属さないものがある）も選択することが非常に重要です。

分類のプロセスとはクラスのエッジを判断することなので、エッジに近いサンプルを用意することが重要です。ただし、学習用サンプルを見ただけで、どれがエッジサンプルとなりうるのか判断できません。このため、学習用セットとして可能な限り多様なサンプルを用意しておくことをお勧めします。

分類子の学習プロセスでは、必要に応じてサンプルを追加したり、サンプルが実際に適切であるかどうかを確認します。場合によっては、サンプル（あまりよくないサンプルだと判明したもの）を一部のクラスから除外します。また、クラスの負サンプルも指定できます。実際のデータに対して有効な分類子が得られるまで、適切な学習用データの発見やさまざまな学習オプションの設定を繰り返し実行します。

25.4.2 学習用セットのラベルの定義

cts:train の 2 番目のパラメータは、ラベル指定です。ラベル指定は cts:label 要素のシーケンスであり、それぞれの要素に 1 つの子 cts:class があります。cts:label 要素はそれぞれ、学習用セット内のノードを表します。cts:label 要素は、特定の学習用ノードの順序に合わせる必要があります。また、それぞれで、対応する学習用ノードの所属クラスを指定します。例えば、次の cts:label ノードでは、最初の学習用ノードはクラス comedy、2 番目はクラス tragedy、3 番目はクラス history にそれぞれ属していると指定しています。

```
<cts:label>
  <cts:class name="comedy"/>
</cts:label>
<cts:label>
  <cts:class name="tragedy"/>
</cts:label>
<cts:label>
  <cts:class name="history"/>
</cts:label>
```

ラベルは学習用ノードの順序に合わせる必要があるので、学習用ノードからラベルを生成できると便利です。例えば、次のコードでは、学習用ノードのプロパティに格納されているプロパティ名 playtype から、ラベルのクラス名を抽出しています。

```
for $play in xdmp:directory("/plays/", "1")
return
  <cts:labels>
    <cts:class name={
      xdmp:document-property(xdmp:node-uri($play))//playtype/text()}/>
  </cts:labels>
```

クラスの負サンプル（つまり、クラスに属していないサンプル）である学習用サンプルがある場合、次のように、cts:class 要素で val="-1" 属性を指定することで、これにラベルを設定できます。

```
<cts:class name="comedy" val="-1"/>
```

また 1 つのラベルに複数のクラスを含めることができます。これはあるクラスのメンバーシップと別のクラスのメンバーシップは無関係であるためです。例：

```
<cts:label>
  <cts:class name="comedy" val="-1"/>
  <cts:class name="tragedy"/>
```

```
    <cts:class name="history"/>
  </cts:label>
```

25.5 クラスごとのしきい値を判断する方法

次の方法に従って、クラスごとに適切な分類用のしきい値を判断します。

1. 学習用セットを 2 つの部分（パーティション）に分割します。これらのパーティションは、統計的に等しいことが理想です。これを実現する 1 つの方法として、ノードを 2 つのパーティションにランダムに振り分けます。

2. 学習用セットの最初の半分に対して cts:train を実行します。

3. 前の手順の最初の半分から得られた cts:train の出力に基づいて、学習用セットの残り半分に対して cts:classify を実行します。こうすることで、ここで使用した学習用データによって、適切な分類結果が得られることが検証されます。ここでは、thresholds オプションのデフォルト値を使用します。このデフォルト値は、極めて大きな負数です。つまりこの実行では、学習用セット内のノードごとに、実際のクラスの境界からの距離が測定されます。

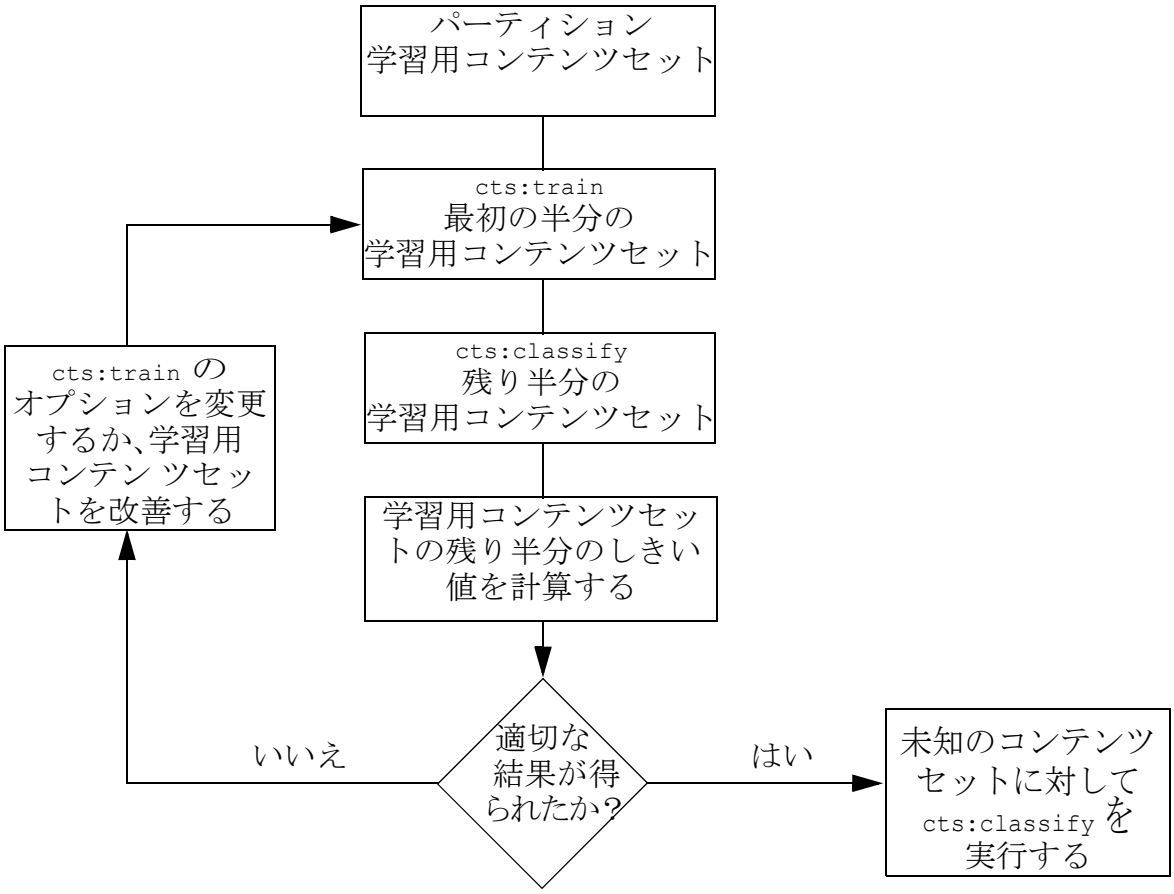
4. cts:thresholds を実行して、学習用セットの残り半分のしきい値を計算します。こうすることで、学習用データに対して cts:train を実行した際に、学習用データとパラメータがさらに検証されます。

5. 学習用コンテンツから満足のいく結果が得られるまで、つまり、作成される分類子に満足がいくまで、上記の手順を繰り返し実行します。場合によっては、必要な分類結果が得られるまで、cts:train のさまざまなオプション設定（カーネルタイプやインデックス設定など）で試行錯誤する必要があります。

6. 満足のいく結果が得られたら、クラスのメンバーシップを判別する際の境界として計算されたしきい値（cts:thresholds の値）を使用して、未知のドキュメントに対して cts:classify を実行します。

   注： しきい値を cts:train に渡すたびに、そのしきい値は cts:classify に適用されます。cts:train と cts:classify のどちらにも渡すことができますが、得られる効果は同じです。

次の図に、この反復プロセスを示します。



25.6 例：分類子の学習と実行

このセクションでは、シェークスピアの戯曲で構成されるコンテンツセットを分類子に学習させるうえで必要な手順について説明します。この簡単な例は、分類子の使用方法を説明することを目的としており、必ずしも分類子から得られる最適な結果の例を取り上げているわけではありません。大まかな手順は次のとおりです。

- シェークスピアの戯曲：学習用セット
- Comedy、Tragedy、History：クラス
- 学習用コンテンツセットを分割する
- 学習用コンテンツの最初の半分に対してラベルを作成する
- 学習用コンテンツの最初の半分に対して cts:train を実行する
- コンテンツセットの残り半分に対して cts:classify を実行する
- cts:thresholds を使用して、残り半分のしきい値を計算する
- 結果を評価して、変更を加えたうえで、このプロセスを反復する
- 他のコンテンツに対して分類子を実行する

25.6.1 シェークスピアの戯曲：学習用セット

分類子の作成では、最初の手順として、学習用コンテンツを選択します。この例では、シェークスピアの戯曲を学習用セットとして使用して分類子を作成します。

シェークスピアの戯曲は、次の URL で XML 形式のものが公開されています（これらの戯曲には、記載されている著作権の制限事項が適用されます）。

http://www.oasis-open.org/cover/bosakShakespeare200.html

この例では、`/shakespeare/plays/` ディレクトリにある MarkLogic サーバーデータベースに戯曲を読み込むと想定します。戯曲は 37 本あります。

25.6.2 Comedy、Tragedy、History：クラス

学習用セットを決定したら、次の手順では、セットの区分に使用するクラスを選択します。また、各クラスのラベルも選択します。シェークスピアの場合、クラスには、`COMEDY`、`TRAGEDY`、および `HISTORY` を使用します。各クラスにどの戯曲が属しているのか、決定する必要があります。シェークスピアのどの戯曲が喜劇（comedy）、悲劇（tragedy）、史劇（history）に該当するかを決定するには、シェークスピアを専門とする研究者に相談してください（どの戯曲がどのクラスに属するかについてはほぼ見解が一致していますが、現在も一部意見が分かれているものもあります）。

便宜上、各戯曲の URI にあるプロパティドキュメントにクラスを格納します。各ドキュメントのプロパティを作成するには、次のような手順を戯曲ごとに実行します。このとき、適切なクラスをプロパティ値として挿入します。

```
xdmp:document-set-properties("/shakespeare/plays/hamlet.xml",
      <playtype>TRAGEDY</playtype>)
```

MarkLogic サーバーのプロパティの詳細については、『Application Developer's Guide』の「Properties Documents and Directories」を参照してください。

25.6.3 学習用コンテンツセットを分割する

次に、学習用セットを 2 つに分割します。どちらの部分においても、各ノードのクラスがわかっています。最初の部分は学習に使用します。残りの部分は、学習用セットの最初の半分から作成した分類子の検証に使用します。2 つの部分は統計的にランダムであることが求められるため、最初の半分は、`xdmp:directory` 呼び出しから返されるドキュメントの順序そのままで使用します。必要に応じて、さらに高度なランダム化手法を選択できます。

25.6.4 学習用コンテンツの最初の半分に対してラベルを作成する

戯曲の最初の半分を学習用コンテンツに使用するので、各ノードにラベルが必要になります（この例では、各戯曲のドキュメントノードを学習用ノードとして使用します)。コンテンツの最初の半分に対してラベルを作成するには、次のようなクエリステートメントを実行します。

```
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>
```

注： わかりやすくするため、この例では、コンテンツセット内の最初の 19 項目を学習用ノードとして使用します。学習用セットに使用するサンプルは、統計的にランダムな必要があるので、学習用セットの選択に際しては、ランダム性を確保するためより複雑な方法を使用する場合も考えられます。

25.6.5 学習用コンテンツの最初の半分に対して cts:train を実行する

次に、用意した学習用コンテンツとラベルで `cts:train` を実行します。次のコードでは、ラベルを作成し、`cts:train` を実行して分類子の仕様を生成しています。

```
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>)
```

必要に応じて、生成した分類子の仕様をデータベース内のドキュメントに保存するか、このコードを次のステップで動的に実行します。

25.6.6 コンテンツセットの残り半分に対して cts:classify を実行する

次に、学習用セットの最初の半分で作成した分類子の仕様を使用して、次のように、コンテンツセットの残り半分に対して `cts:classify` を実行します。

```
let $firsthalf := xdmp:directory("/shakespeare/plays/", "1")[1 to 19]
let $secondhalf := xdmp:directory("/shakespeare/plays/", "1")[20 to 37]
let $classifier :=
  let $labels := for $x in $firsthalf
        return
        <cts:label>
          <cts:class name={xdmp:document-properties(xdmp:node-uri($x))
                           //playtype/text()}/>
        </cts:label>
  return
  cts:train($firsthalf, $labels,
      <options xmlns="cts:train">
        <classifier-type>supports</classifier-type>
      </options>)
return
cts:classify($secondhalf, $classifier,
      <options xmlns="cts:classify"/>,
      $firsthalf)
```

25.6.7 cts:thresholds を使用して、残り半分のしきい値を計算する

次に、残り半分のコンテンツの cts:label 要素を計算して、その結果に基づいて分類子で使用するしきい値を計算します。次のコードでは、わかりやすいように、cts:train と cts:classify を再実行していますが、それぞれの出力をドキュメントに格納しておくこともできます。

```
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)
```

これで、次のような出力が得られます。

```
<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>
```

25.6.8 結果を評価して、変更を加えたうえで、このプロセスを反復する

最後に、`cts:thresholds` の結果を分析します。しきい値はゼロであることが理想です。実際には、比較的ゼロに近い負数であれば、しきい値として良好です。上記の tragedy のしきい値はかなり良好ですが、他のクラスのしきい値はそれほど良好ではありません。しきい値を改善するには、カーネルやインデックス化オプションなどのパラメータを変更して、すべての処理を実行し直します。あるいは、例えば comedy の例として適切なものを見つけるため、学習用データを変更します。

25.6.9 他のコンテンツに対して分類子を実行する

満足できる分類子が得られたら、他のコンテンツに対して実行してみます。例えば、シェークスピアの戯曲の SPEECH 要素に対して実行したり、他の作家の戯曲に対して実行します。

26.0 cts:cluster を使用した結果のクラスタ化

MarkLogic サーバーには、統計アルゴリズムを使用して検索結果のクラスタを見つけてラベル付けする、cts:cluster があります。この章では、cts:cluster について説明します。次のセクションから構成されます。

- cts:cluster とは
- cts:cluster のオプション
- cts:cluster の出力とは
- クラスタの HTML レポートを作成する例

シグネチャ、パラメータのシンタックスの詳細、その他の例については、『MarkLogic XQuery and XSLT Function Reference』の「cts:cluster」を参照してください。

26.1 cts:cluster とは

cts:cluster 関数は、通常は検索結果からのノードの集まり（あるいは任意のノードの集まり）を取得し、結果ノードをクラスタに分類したレポートを提供します。クラスタとは、統計的に類似した結果のサブセットです。各クラスタごとに、そのクラスタで最も特徴的なタームからラベルを生成します。

出力は XML ノードであり、この出力を使用して、結果を表示するユーザーインターフェイスを生成できます。出力のサンプルについては、「cts:cluster の出力とは」（464 ページ）を参照してください。

クラスタ化機能では、cts:cluster に渡すノードを取得し、MarkLogic サーバーインデクサーに通すことにより、クラスタが作成されます。このプロセスは、ドキュメントをデータベースに読み込むときのプロセスと非常に似ていますが、結果のクラスタ化用のインデックス付けはすべてメモリ内で実行されるのに対して、データベース内ではインデックスはディスクに格納されます。インデックス付けによって生成されるのはタームであり、各タームには頻度（ドキュメントおよび結果セットでの出現回数）が付いています。使用するインデックス設定によって、得られるタームは異なります。クラスタ化機能は、クラスタの計算において各タームならびにタームに関する情報（例えば、重みやターム頻度）を考慮します。

cts:cluster には、クラスタの動作を決定するオプションを渡すほか、クラスタ作成時に使用するインデックス設定を指定します。これらのオプションの詳細については、「cts:cluster のオプション」（462 ページ）、および『MarkLogic XQuery and XSLT Function Reference』の cts:cluster に関する API の説明を参照してください。

クラスタ化機能の使い方は、各自の要件に基づいて判断してください。クラスタ化機能の設定の多くは、パフォーマンスと結果クラスタの質の間にトレードオフの関係があります。アプリケーションに適した設定を見つけるためには、試行錯誤が必要です。

クラスタ化機能について、次のことに注意してください。

- クラスタ化するたびに、データ生成のために供給されたノードに対してインデクサーが実行されます。
- `cts:cluster` に送るノードが多くなるほど、時間がかかります。リアルタイム分析では、数千ノードを超えると、遅くなりすぎる可能性があります。理想的には、100 ～ 1000 ノードで、パフォーマンスと有効な結果のバランスが取れます。
- クラスタ内にクラスタを生成する場合は、`<hierarchical-levels>` に 1 よりも大きい値を設定します。親（parent）`attribute` は、どのクラスタが親であるかを示してくれます。これにより、結果セットを反復処理して、ツリー状の階層を表示するユーザーインターフェイスを作成できます。
- ラベルは、実行ごとに変わる可能性があります。`<num-tries>` の値が大きくなると、実行ごとのラベルの一貫性が向上する傾向がありますが、クラスタ生成にかかる時間が長くなります。
- ラベルは、最も特徴的なタームから取得されます。いくつかのターム（要素タームなど）は文字列に変換されます。ラベルの作成に使用されたタームを表示する場合は、`<details>true</details>` オプションを設定します。

26.2 cts:cluster のオプション

options ノードで `cts:cluster` のオプションを設定できます。次のタイプのオプションを設定できます。

- クラスタ化（cts:cluster）のオプション
- インデックス付け（db:）オプション

これらのタイプのオプションはそれぞれ、専用の名前空間内にあります。

26.2.1 クラスタ化（cts:cluster）のオプション

クラスタ化のオプションは、`cts:cluster` 名前空間内にあります。これらのオプションにより、クラスタ化機能の出力と動作が決定されます。クラスタ化機能のオプションについて、次のことに注意してください。

- オプションを調整することで、結果のパフォーマンス、精度、および質のバランスを取るようにしてください。

- <details> オプションにより、各クラスタに使用されている特徴的なターム（これらは cts タームです）が返されます。各タームから cts:query コンストラクタを生成することにより、これらのタームを使用して独自のラベルを作成できます。その後、アプリケーション的に意味があるのであれば、一部のデータに対してそれらのクエリを使用していくつかのラベルを生成できます。
- <algorithm> オプションで、MarkLogic サーバーがクラスタの演算に使用するアルゴリズムを設定できます。k-means または lsi を設定します。いずれも統計的アルゴリズムであり、これらに関しては有名な論文があります（詳細は、まず http://en.wikipedia.org/wiki/K-means_clustering や http://en.wikipedia.org/wiki/Latent_semantic_indexing を確認してみてください）。デフォルトは k-means です。これは、lsi よりもわずかに速くなりますが、結果の安定性がわずかに低下します。
- <min-clusters> および <max-clusters> 設定で、クラスタの個数を制御できます。cts:cluster では、データから算出されるクラスタ数の最大が <min-clusters> の値よりも小さい場合、この個数のクラスタを返すことがあります。
- <num-tries> オプションは、指定されたデータに対してクラスタ化機能を実行する回数を指定します。デフォルトは 1 です。このアルゴリズムでは、クラスタを複数回実行するとターム数が増え、クラスタの精度が向上する傾向があります。これはパフォーマンスを犠牲にしているので、実行するたびに、処理量が増えます。

26.2.2 インデックス付け（db:）オプション

インデックス付けオプションは、どのタームが作成されるかを制御します。MarkLogic サーバーでは、これらのタームを使用して、ターム頻度、特徴的なターム（distinctive terms）、その他の重要性に基づいてクラスタを計算します。db のオプションでは、次のことに注意してください。

- これらは、options ノードで設定し、http://marklogic.com/xdmp/database 名前空間内にあります。
- cts:cluster のデータベースオプションは、cts:distinctive-terms のデータベースオプションと同じです。
- オプションは手動あるいは、Admin API を使用して作成します。
- フィールドは、関心のある単語のみをインデックス付けするための優れた手段です。またこれを使用すると、特定の要素に重みを設定できます。フィールドの詳細については、『Administrator's Guide』の「Fields Database Settings」を参照してください。
- （cts:cluster 名前空間内の）<use-db-options> cts:cluster オプションは、コンテキストデータベースで設定されたデータベースオプション、指定されたデータベースオプション、およびオプションのデフォルト値の組み合わせを取ります。これは、複雑なオプションを設定する場合に便利です。

- 異なるオプションを試してみて、パフォーマンスとタームの適切な組み合わせを判断します。

26.3 cts:cluster の出力とは

次に、`cts:cluster` 出力のサンプルを示します。

```
<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="1248501351668626361" 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>
```

出力は、cts:clustering 要素です。出力には、各クラスタならびにその作成に使用された options ノードが含まれます。XQuery または XSLT を使用してこの出力を反復処理することで、(HTML 形式などの) 結果レポートを作成できます。

<cluster> 要素の属性は、クラスタについて記述しています。次の表で、<cluster> 要素の属性について説明します。

cluster の属性	説明
id	クラスタの識別に使用されるランダムな番号。
parent-id	親クラスタの ID (<hierarchical-levels> に 2 以上の値が設定されている場合)。
label	ラベルを構成するターム。カンマで区切られます。独自のラベルを作成するには、<details> を返し、タームを使用してラベルを生成します。
count	クラスタ内のノードの個数。
nodes	NMTOKEN 値の集まり。各値でノードの位置がリストされます。この位置は関連度で順序付けられます。最初のものがクラスタに対する関連度が最も高く、最後が最も低くなっています。この番号は、cts:cluster へのノード入力内の位置を表します。例えば値 10 は、cts:cluster の最初のパラメータに渡されたシーケンス内の 10 番目のノードであることを示します。

26.4 クラスタの HTML レポートを作成する例

次の例では、クラスタの HTML レポートを作成しています。シェークスピアの戯曲のデータベースを使用しています。結果を表示するには、この例をカット&ペーストし、シェークスピアの戯曲を含んでいるデータベースに対して実行します (cts:search で使用されているディレクトリの URI を、シェークスピアの戯曲を読み込んだデータベースディレクトリの URI に修正します)。

```
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() )
```

27.0 MarkLogic サーバーの言語サポート

MarkLogic サーバーでは、複数の言語でのコンテンツの読み込みとクエリ実行がサポートされています。この章では、MarkLogic サーバーで言語が扱われる仕組みについて説明します。次のセクションから構成されます。

- MarkLogic サーバーでの言語サポートの概要
- トークン化とステミング
- ドキュメントの読み込み時および更新時の言語
- 言語別のドキュメントのクエリ実行
- サポートされている言語
- 一般言語のサポート

27.1 MarkLogic サーバーでの言語サポートの概要

MarkLogic サーバーでは、コンテンツの言語はコンテンツの読み込み時に指定し、クエリの言語はコンテンツにクエリを実行する際に指定します。読み込みの際、コンテンツは指定された言語に基づいてトークン化、インデックス付け、およびステミング（オンになっている場合）されます。また MarkLogic サーバーでは、コンテンツの XML マークアップにおいて要素レベルで指定された言語が使用されるので、複数言語を含むドキュメントを読み込むことができます（「xml:lang 属性」（473 ページ）を参照）。同様にクエリの際に、検索タームは、`cts:query` 式で指定された言語に基づいてトークン化（およびステミング）されます。このため、同一のクエリを異なる言語に対して実行した場合、同じ結果が得られないことがあります。コンテンツ情報を格納するインデックスと、このコンテンツに対して実行されるクエリの両方で、言語が認識されるからです。

コンテンツで 1 つの言語しか使われていない場合でも、MarkLogic サーバーでは複数言語の認識が可能です。コンテンツで 1 つの言語しか使われておらず、その言語がこのデータベースのデフォルト言語の場合、またコンテンツに言語（`xml:lang`）属性がなく、すべてのクエリにおいてコンテンツが読み込まれた際の言語を指定している（またはその言語がデフォルトとなる）場合、単一の言語を扱っている（言語が 1 つしかない）ものとして全体が動作します。

MarkLogic サーバーは複数の言語を認識するため、MarkLogic サーバーでのコンテンツの読み込みおよびクエリ実行時の、言語の基本的側面を理解することが重要です。この章の以降、特に「ドキュメントの読み込み時および更新時の言語」（473 ページ）および「言語別のドキュメントのクエリ実行」（475 ページ）では、これらの側面を詳細に説明します。

27.2 トークン化とステミング

ドキュメントのクエリ実行および読み込みに対する言語の意味を理解するには、まずトークン化とステミングについて理解する必要があります。これらはいずれも言語固有です。このセクションでは、これらのトピックについて説明します。次のパートがあります。

- 言語固有のトークン化
- さまざまな言語でのステミング検索

27.2.1 言語固有のトークン化

MarkLogic サーバーで文字列（通常は単語または語句）を検索した場合、または（テキスト文字列から構成されている）コンテンツを MarkLogic サーバーに読み込んだ場合、文字列はトークンと呼ばれる部分に分割されます。各トークンは単語、句読点、またはスペースとして分類されます。文字列をトークンに分解するプロセスは、*トークン化*と呼ばれます。トークン化は、クエリの評価時ならびにドキュメントの読み込み時に行われます。これらは、互いに独立しています。

トークン化は言語固有です。つまり、文字列は言語に応じて異なった方法でトークン化されます。この言語は、読み込み時またはクエリ時に指定された言語（指定されていない場合はデータベースのデフォルト言語）、およびコンテンツ内の `xml:lang` 属性（詳細は「xml:lang 属性」（473 ページ）」を参照）に基づいて決定されます。

MarkLogic サーバーでの文字列のトークン化について、次のことに注意してください。

- `cts:tokenize` API は、指定された言語でテキストがどのようにトークン化されるかを返します。
- `cts:tokenize` 式の `xdmp:describe` を使用すると、指定された文字列から生成されたトークンと、トークンのタイプが返されます。例えば次のようになります。

```
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"))
```

- 各クエリには言語が関連付けられます。`cts:query` 式で言語が明示的に指定されていない場合、データベースのデフォルト言語が採用されます。

- MarkLogic サーバーでは、要素がアジアまたは中東の言語である場合、ラテン文字は英語としてトークン化されるように設定されています。これにより、アジアまたは中東の言語の要素にある英単語を見つけることができます。例えば次のように、英語で検索することで、簡体字中国語の要素にあるラテン文字を見つけることができます。

```
let $x := <el xml:lang="zh">ここに中国語テキスト hello</el>
return
$x//el[cts:contains(.,
         cts:word-query("hello", ("stemmed", "lang=en")))]

=> <el xml:lang="zh">ここに中国語テキスト hello</el>
```

  ただし、英語以外の言語でラテン文字をステミング検索すると、英語以外の語幹は見つかりません（非英語の言語の単語それ自体のみが見つかります。これはこの語がそれ自体にステミングされるためです)。同様に、アジアおよび中東の文字は、その文字セットに該当する言語でトークン化されます（別の言語の要素に出現した場合であっても)。このため、英語での検索が、アジアまたは中東の文字セットとしてラベル付けされたコンテンツとマッチすることがあります。これは、逆も同様です。例えば、次について考えてみましょう（zh は簡体字中国語の言語コードです)。

```
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>
```

  これは英語での検索ですが、英語の要素と中国語の要素の両方が返されます。中国語の要素が返されている理由は、単語「hello」はラテン文字であるため英語としてトークン化されるからです。また、これは中国語のクエリともマッチします(この場合も、英語の「hello」がトークン化されます)。

- アプリケーションにおいて特別なトークン化要件がある場合は、カスタムのトークナイザーを使って、文字のトークン化の方法を修正できます。詳細については、「カスタムトークン化」（480 ページ）を参照してください。

27.2.2 さまざまな言語でのステミング検索

タームのステミング検索では、(クエリで指定された言語の完全に同じタームも含む)検索タームと同じ語幹のすべてのタームと一致します。同じ意味と品詞から派生した単語は、同じ語幹を持ちます(例えば、「mouse」と「mice」など)。一部の単語は複数の語幹を持つことがあります(同じ単語が異なる品詞として存在する場合、またはスペリングが同じ別の単語の場合)。また、高度なステミングを使用した場合(単語の複数の語幹を見つけることができます)、ステミング検索では、いずれかの語幹と同じ語幹を持つすべての単語が検索されます。ステミングの目的は、検索の再現度(recall)を高めることです。利用可能なさまざまなステミングタイプも含め、MarkLogic サーバーでのステミングの詳細については、「ステミング検索の概要と使い方」(385 ページ)を参照してください。このセクションでは、言語設定がステミング検索にどのように影響するかについて説明します。

検索タームの語幹を得るには、言語を考慮する必要があります。例えば、語「chat」は、フランス語と英語では別の言葉です(フランス語では名詞「猫」、英語では動詞「おしゃべりする」)。フランス語では、「chatting」という語は存在しないため、「chat」にステミングされません。しかし英語では、「chatting」は「chat」にステミングされます。このようにステミングは言語固有です。このため、ある言語でのステミング検索と他の言語でのステミング検索では結果が異なる場合があります。

クエリでは言語を指定できます(指定しなかった場合、データベースのデフォルト言語が使用されます)。この言語が、ステミング検索で使用されます。言語指定は、`cts:query` 式のオプションにあります。例えば、次の `cts:query` 式では、フランス語での語「chat」のステミング検索を指定しており、フランス語でステミングされたトークンにのみマッチされます。

```
cts:word-query("chat", ("stemmed", "lang=fr"))
```

言語がクエリに与える影響の詳細については、「言語別のドキュメントのクエリ実行」(475 ページ)を参照してください。

読み込み時には、指定されている言語がドキュメント内の単語のステミングに使用されます。ドキュメント読み込み時の言語の詳細については、「ドキュメントの読み込み時および更新時の言語」(473 ページ)を参照してください。

さまざまな `cts:query` コンストラクタのシンタックスの詳細については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

27.3 ドキュメントの読み込み時および更新時の言語

このセクションでは、言語がドキュメントの読み込みと更新に与える影響について説明します。次のセクションから構成されます。

- トークン化とステミング
- xml:lang 属性
- ドキュメントの読み込みおよび更新の際の言語に関する注意

27.3.1 トークン化とステミング

ドキュメントのクエリ実行時とまったく同様に、ドキュメントの読み込み時にトークン化とステミングが行われます（詳細については、「言語固有のトークン化」（470 ページ）と「さまざまな言語でのステミング検索」（472 ページ）を参照してください）。ドキュメントの読み込み時に、言語に基づいて `stemmed search` インデックスが作成されます。読み込み時のトークン化とステミングは、クエリ時のトークン化とステミングとは完全に独立して行われます。

27.3.2 xml:lang 属性

XML ドキュメントでは、`xml:lang` 属性を使用することにより、要素レベルで言語を指定できます。MarkLogic サーバーでは、要素のコンテンツのトークン化およびステミングに使用する言語の特定に、`xml:lang` 属性を使用します。`xml:lang` 属性について、次のことに注意してください。

- `xml:lang` 属性（http://www.w3.org/TR/2006/REC-xml-20060816/#sec-lang-tag を参照）には、特別なプロパティがいくつかあります。例えば `xml` プレフィックスにバインドされる名前空間を宣言する必要がありません。またこれは、すべての子要素に継承されます（明示的に異なる `xml:lang` 値が指定されている場合を除く）。
- `xdmp:document-load` に `default-language` オプションを指定することで、読み込み時にドキュメントのルートノードに `xml:lang` 属性を明示的に追加できます。`default-language` オプションがない場合、ルートノードは何も変更されません。
- `xml:lang` 属性がない場合、ドキュメントは読み込み先のデータベースのデフォルト言語で処理されます。
- タームのインデックス付けに関しては、`xml:lang` 属性によって指定された言語は、ステミングされる検索タームにのみ適用されます。データベース設定 `word searches`（ステミングなし）では、言語に関係なくタームにインデックスが付けられます。タームのトークン化においては、データベース設定のインデックス設定 `stemmed searches` と `word searches` の両方に関して `xml:lang` 値が優先されます。

- xml:lang 属性を持つ要素のすべての子テキストノードおよび子孫テキストノードの言語は、xml:lang 属性で指定された言語として扱われます（子要素に異なる値の xml:lang 属性が指定されていない場合）。子要素に異なる値が指定された場合、子テキストノードおよび子孫テキストノードの言語は、他の xml:lang 属性が出現するまで、その新しい言語として扱われます。
- xml:lang 属性の値は、レキシコン標準 http://www.ietf.org/rfc/rfc3066.txt に準拠する必要があります。次に、一般的な xml:lang 属性をいくつか示します（それぞれ、フランス語、簡体字中国語、および英語を指定しています）。

```
xml:lang="fr"
xml:lang="zh"
xml:lang="en"
```

- 要素において xml:lang 属性（xml:lang=""）の値が空の文字列である場合、有効だったあらゆる xml:lang 値（祖先の xml:lang 値など）がこれで上書きされてしまい、この値としてデータベースの言語のデフォルトが採用されます。さらに、読み込み時に default-language オプションが指定された場合、空の文字列である xml:lang 値は、default-language オプションで指定された言語によって置き換えられます。例えば、次の XML について考えてみましょう。

```
<rhone xml:lang="fr">
   <wine>vin rouge</wine>
   <wine xml:lang="">red wine</wine>
</rhone>
```

このサンプルでは、語句「vin rouge」はフランス語として扱われ、語句「red wine」はデータベースのデフォルト言語（英語）で扱われます。

もしこのサンプルを、default-language オプションでイタリア語を指定して（例えば、xdmp:document-load オプションに <default-language>it</default-language> を指定して）読み込んだ場合、結果として得られるドキュメントは次のようになります。

```
<rhone xml:lang="fr">
   <wine>vin rouge</wine>
   <wine xml:lang="it">red wine</wine>
</rhone>
```

27.3.3 ドキュメントの読み込みおよび更新の際の言語に関する注意

MarkLogic サーバーではコンテンツを読み込む際、いくつかの要因に基づいてコンテンツのインデックス付け方法が決定されます。例えば、読み込み操作中に指定された言語、データベースのデフォルト言語、`xml:lang` 属性によってコンテンツにエンコーディングされた言語などが考慮されます。コンテンツの読み込み、コンテンツの更新、およびデータベースの言語設定の変更の際には、言語について次のことに注意してください。

- `reindex enable` が `true` に設定されている場合、デフォルト言語を変更すると、再インデックス付け操作が開始されます。
- `xml:lang` 属性がないドキュメントは、読み込み時または更新時にデータベースのデフォルト言語でインデックス付けされます。
- `xml:lang` 属性を持つ要素内のあらゆるコンテンツは、その言語でインデックス付けされます。また `xml:lang` 値は、他の `xml:lang` 値が検出されるまで、その要素のすべての子孫によって継承されます。
- MarkLogic サーバーでは、要素がアジアまたは中東の言語である場合、ラテン文字は英語としてトークン化されるように設定されています。したがって、非英語のラテン文字が含まれるドキュメントでは、そのラテン文字が英語としてステミングされたインデックスタームが作成されます。同様に、アジアまたは中東の文字がそれぞれの言語ではない要素に含まれていた場合でも、これらの文字はそれぞれの言語でトークン化されます。

27.4 言語別のドキュメントのクエリ実行

全文検索クエリ（`cts:search` または `cts:contains` を使用するクエリ）では、言語が認識されます。つまり、特定の言語でテキストを検索し、検索タームをトークン化し、（オンになっている場合は）ステミングします。このセクションでは、クエリが言語をどのように認識するのか、またその動作について説明します。次のトピックから構成されます。

- トークン化、ステミング、および xml:lang 属性
- 言語を認識した検索
- ステミングなし検索
- 不明な言語

27.4.1 トークン化、ステミング、および xml:lang 属性

トークン化とステミングはともに言語固有です。つまり言語が異なる場合、文字列は異なる方法でトークン化およびステミングされる可能性があります。検索では、cts:query コンストラクタによって言語が指定されます（指定されていない場合、データベースのデフォルト言語によって指定されます）。詳細については、「トークン化とステミング」（470 ページ）を参照してください。XQuery で構築されたノードでは、`xml:lang` 属性は、ドキュメントがデータベースに読み込まれる際と同様に扱われます。詳細については、「xml:lang 属性」（473 ページ）を参照してください。

27.4.2 言語を認識した検索

MarkLogic サーバーではすべての検索で、言語が認識されます。検索は `cts:search` または `cts:contains` を使用して構築できます。それぞれが `cts:query` 式を取ります。`cts:query` 式の各リーフレベルの `cts:query` コンストラクタで、言語を指定します（あるいはデフォルトの言語が使用されます）。`cts:query` コンストラクタの詳細については、「cts:query 式の構成」（239 ページ）を参照してください。

あらゆる検索では、`cts:query` コンストラクタの言語設定を使用して、検索タームをトークン化する方法を特定します。ステミング検索では、語幹の生成にもこの言語設定を使用します。ステミングなし検索では、トークン化に指定された言語が使用されますが、ステミングなし（`word searches`）インデックスが使用されます。このインデックスは言語に依存しません。

27.4.3 ステミングなし検索

*ステミングなし*検索では、検索タームそのままの（まったく同じ）タームとマッチされます。語幹は考慮されません。ステミングなし検索では、言語に依存せずにタームとマッチされますが、指定された言語によって検索がトークン化されます。したがって、ステミングなしクエリで言語を指定した場合、その言語はトークン化にのみ適用されます。ステミングなしクエリでは、クエリとマッチするあらゆる言語のあらゆるテキストとマッチされます。

ステミングなし検索では、次の特性に注意してください。

- ステミングなし検索には、`word search` インデックスが必要です。それ以外の場合は、例外がスローされます（これは、MarkLogic サーバー 3.1 からの動作の変更点です。詳細については、『Release Notes』を参照してください）。ただし、`cts:contains` を使用することで `word search` インデックスがなくてもステミングなし検索を実行できます。`word search` インデックスをオンにせずにステミングなし検索を実行するには、`let` を使用してステミング検索の結果を変数にバインドした後、ステミングなしクエリと `cts:contains` を使用して結果をフィルタリングします。

次の例で、これを示します。ステミングなし検索を変数にバインドした後、FLWOR ループで検索の結果を反復することにより、where 節で、ステミングなしの結果以外はすべてフィルタリングで除外しています（unstemmed オプションを指定する cts:query とともに cts:contains を使用しています)。

```
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
```

注： この検索によって返されるすべてのものに、cts:query にマッチする英語が含まれる可能性が高いですが、必ずしも、返されるものすべてが英語だと*保証*されるわけではありません。この検索ではドキュメントが返されるため、言語固有のクエリとはマッチされない他の言語がドキュメントに含まれており、しかしながらこれがステミングなしの検索にマッチしているということも起こりえます（ドキュメントに複数の言語のテキストが含まれる場合で、さらにステミングの cts:query で指定された言語とは別の言語の中に「my words」が含まれるとき)。

- word search インデックスには言語情報が含まれません。
- ステミングなし検索では、トークン化の言語の特定に lang=<language> オプションが使用されます。
- ステミングなし検索では、言語にかかわらず（および lang=<language> オプションにかかわらず)、すべてのコンテンツが検索されます。この言語は、検索タームがトークン化される方法にのみ影響します。例えば、ステミングなし検索

```
(: returns true :)
let $x := <el xml:lang="fr">chat</el>
return
cts:contains($x, cts:word-query("chat", ("unstemmed", "lang=en")))
```

では true が返されるのに対して、次のステミング検索では false が返されます。

```
(: returns false :)
let $x := <el xml:lang="fr">chat</el>
return
cts:contains($x, cts:word-query("chat", ("stemmed", "lang=en")))
```

27.4.4 不明な言語

検索に指定された言語では、言語固有のステミングおよびトークン化がサポートされていない場合、またはライセンスキーがない場合、これは一般言語として扱われます。ラテン文字の一般言語は通常、英語と同じ方法でトークン化されます。つまりスペースと句読点によるトークン区切りで、単語がそれ自体にステミングされます。ただし、これが常に当てはまるわけではありません（特に、MarkLogic サーバーによってサポートされている言語（「サポートされている言語」（478 ページ）を参照）のうち、ライセンスがない言語の場合）。詳細については、「一般言語のサポート」（479 ページ）を参照してください。

27.5 サポートされている言語

このセクションでは、MarkLogic サーバーでの高度なステミングとトークン化に対応している言語のリストを紹介します。英語以外のすべての言語で、それぞれの言語対応のライセンスキーが必要です。ライセンスキーに特定の言語のサポートが含まれない場合、その言語は、一般言語として扱われます（「一般言語のサポート」（479 ページ）を参照してください）。次に、サポートされている言語を示します。

- 英語
- フランス語
- イタリア語
- ドイツ語
- ロシア語
- スペイン語
- アラビア語
- 中国語（簡体字および繁体字）
- 韓国語
- ペルシア語（現代ペルシア語）
- オランダ語
- 日本語
- ポルトガル語
- ノルウェー語（ニーノシュクおよびブークモール）
- スウェーデン語

各言語で使用されるベースコレーションおよび文字セットのリストについては、「言語ごとのコレーションと文字セット」（502 ページ）を参照してください。

27.6 一般言語のサポート

文字エンコーディングを UTF-8 に変換できる限り、任意の言語のドキュメントを MarkLogic サーバーに読み込み、クエリを実行できます。その言語が高度なサポート対象でない場合、またはライセンスされていない場合、トークン化は一般的な方法（アジア以外の文字ではスペースおよび句読点に基づき、アジアの文字ではそれぞれの文字に基づく方法）で実行され、各タームはそれ自体にステミングされます。

例えば、ドキュメント

```
<doc xml:lang="nn">
  <a>Some text in any language here.</a>
</doc>
```

を読み込む場合、ドキュメントは言語 nn として読み込まれ、他の言語でのステミング検索ではマッチされません。したがって、

```
(: does not match because it was stemmed as "nn" :)
cts:search(doc(), cts:word-query("language", ("stemmed", "lang=en"))
```

はこのドキュメントとマッチされず、次の検索はそのドキュメントとマッチされます。

```
(: does match because the query specifies "nn" as the language :)
cts:search(doc(), cts:word-query("language", ("stemmed", "lang=nn"))
```

一般言語のサポートにより、ライセンスされている言語や高度なサポートがある言語でなくても、任意の言語のドキュメントに対してクエリを実行できます。一般言語のサポートでは、単語はそれ自体にしかステミングされないため、このような言語でのクエリでは、意味に基づく単語のバリエーションが結果に含まれません。特定の言語について一般言語よりも進んだサポートが必要な場合は、MarkLogic テクニカルサポートにお問い合わせください。

28.0 カスタムトークン化

コンテンツおよびクエリテキストでの文字の分類を変更するには、フィールドに対してカスタムのトークナイザーオーバーライドを使用します。文字の分類が変更されると、テキストのトークン化が変更されるため、検索に影響します。次のトピックを扱います。

- カスタムのトークナイザーオーバーライドに関する概要
- 文字の分類のトークン化への影響
- xdmp:describe を使用してトークン化を調べる
- トークナイザーオーバーライドを使用した場合のパフォーマンスへの影響
- カスタムのトークナイザーオーバーライドの定義
- カスタムのトークナイザーオーバーライドの例

28.1 カスタムのトークナイザーオーバーライドに関する概要

カスタムのトークナイザーオーバーライドにより、フィールド内で文字が出現したときの、トークナイザーによる文字の分類を変更できます。この柔軟性を利用して、検索効率を向上させたり、特別な文字パターンの検索を可能にしたり、データを正規化したりできます。

> **注：** フィールドには、カスタムのトークナイザーオーバーライドのみを定義できます。詳細については、『Administrator's Guide』の「Overview of Fields」を参照してください。

「トークン化とステミング」（470 ページ）で説明したように、トークン化によって、テキストのコンテンツおよびクエリテキストが単語トークン、句読点トークン、およびスペーストークンに分割されます。ビルトインの言語固有ルールにより、テキストをトークンに分割する方法が定義されています。トークン化時に、各文字は単語、記号、句読点、スペース文字のいずれかに分類されます。各記号、句読点、またはスペース文字は 1 つのトークンです。単語の隣接する文字は、まとめて 1 つの単語トークンとしてグループ化されます。単語トークンと記号トークンにはインデックスが付けられ、スペーストークンと句読点トークンにはインデックスが付けられません。

例えば、デフォルトのトークン化では、「456-1111」形式のテキストは、2 つの単語トークンと 1 つの句読点トークンに分解されます。次のようなクエリを使用して、分解について調べることができます。

```
xquery version "1.0-ml";
xdmp:describe(cts:tokenize("456-1111"));

==> ( cts:word("456"), cts:punctuation("-"), cts:word("1111") )
```

カスタムのトークナイザーオーバーライドを定義して、ハイフンを削除すべき文字として分類した場合、このトークナイザーは 1 つの単語トークン「4561111」を生成します。これを他のデータベース設定の変更と組み合わせて、より正確なワイルドカード検索をオンにしたり、456-1111 や 4561111 などのようなバリエーションがある入力とマッチさせたりできます。例の全体については、「例：ワイルドカードがオンにされている検索の精度の向上」（485 ページ）を参照してください。

トークン化ルールは、コンテンツとクエリテキストの両方に適用されます。トークナイザーオーバーライドはフィールドにのみ適用できるので、オーバーライドを有効活用するには、`cts:field-word-query` や `cts:field-value-query` のようなフィールドクエリが必要です。

文字をそのデフォルトのクラスにオーバーライドすることはできません。例えばデフォルトでは、スペース文字（" "）のクラスは `space` です。したがって、これを `space` として分類するオーバーライドは定義できません。

NFKD Unicode 正規化が適用されている場合、複数の文字に分解される複合文字をオーバーライドすることはできません。

28.2 文字の分類のトークン化への影響

カスタムのトークナイザーオーバーライドは、文字を次のカテゴリのいずれかとして分類するように定義できます。

- `space`：スペースとして扱います。スペースにはインデックスが付けられません。
- `word`：単語の隣接する文字は、1 つのトークンとしてグループ化され、インデックスが付けられます。
- `symbol`：各記号文字は、1 つのトークンとなり、インデックスが付けられます。
- `remove`：トークンの作成時に、その文字を対象から除外します。

文字がクエリテキスト内で句読点として扱われるように分類を変更すると、句読点による区別がある（punctuation-sensitive）フィールドワードクエリおよびフィールド値クエリがトリガーされる可能性があることに注意してください。詳細については、「ワイルドカード機能と句読点による区別」（411 ページ）を参照してください。

次の表に、各種類のオーバーライドによるトークン化への影響について示します。

文字のクラス	入力例	デフォルトのトークン化	オーバーライド	結果
space	10x40	10x40（単語）	admin:database-tokenizer-override("x","space")	10（単語） x（スペース） 40（単語）
word	456-1111	456（単語） -（句読点） 1111（単語）	admin:database-tokenizer-override("-","word")	456-1111（単語）
symbol	@1.2	@（句読点） 1.2（単語）	admin:database-tokenizer-override("@","symbol")	@（記号） 1.2（単語）
remove	456-1111	456（単語） -（句読点） 1111（単語）	admin:database-tokenizer-override("-","remove")	4561111（単語）

28.3 xdmp:describe を使用してトークン化を調べる

cts:tokenize とともに xdmp:describe を使用して、フィールドとトークナイザーオーバーライドの使用がトークン化にどのような影響を与えるかを調べることができます。フィールドの名前を cts:tokenize に渡すと、フィールドで定義されたオーバーライドがトークン化に適用されます。

次の例は、デフォルトのトークン化ルール（フィールドなし）と、トークナイザーオーバーライドを定義する「dim」という名前のフィールドを使用したトークン化の違いを示しています。この例で使用されているフィールドの設定の詳細については、「例：単語内の検索」（488 ページ）を参照してください。

```
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"))
```

この方法で、新しいオーバーライドの効果をテストできます。

また、cts:tokenize に対する 2 番目の引数として言語を含めて、トークン化への言語固有の効果を確認することもできます。例えば次のようになります。

```
xdmp:describe(cts:tokenize("20x40","en","dim"))
```

言語とトークン化の関係の詳細については、「MarkLogic サーバーの言語サポート」（469 ページ）を参照してください。

28.4 トークナイザーオーバーライドを使用した場合のパフォーマンスへの影響

トークナイザーオーバーライドを使用すると、インデックス付けと検索にかかる時間が長くなるため、オーバーライドは本当に必要な場合にのみ使用してください。例えば、カスタムオーバーライドが検索のスコープに入っている場合、フィルタリングがあるフィールドクエリでは、フィルタリング時にチェックされる各テキストノードを再トークン化する必要があります。

トークナイザーオーバーライドの数が少ない場合、影響はあまり大きくありません。

非常にスコープが広いフィールドに対してカスタムトークナイザーを定義した場合、パフォーマンスが大きな影響を受けます。""（スペース）のような一般的な文字の分類を記号や単語の文字に変更すると、インデックスに必要な容量が大幅に増加することがあります。

28.5 カスタムのトークナイザーオーバーライドの定義

フィールドには、カスタムのトークナイザーオーバーライドのみを定義できます。カスタムのトークナイザーオーバーライドは、次の方法で設定できます。

- プログラムから、Admin API の関数 `admin:database-add-field-tokenizer-override` を使用する。
- インタラクティブに Admin 画面を使用する。詳細については、『Administrator's Guide』の「Configuring Fields」を参照してください。

例えば、データベース設定に `phone` という名前のフィールドがすでに含まれると仮定します。次の XQuery は、このフィールドに、「-」を `remove` 文字、「`@`」を記号にそれぞれ分類するカスタムのトークナイザーオーバーライドを追加します。

```
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)
```

28.6 カスタムのトークナイザーオーバーライドの例

このセクションには、カスタムのトークナイザーオーバーライドの使い方に関連する次の例が含まれています。

- 例：フィールドをトークナイザーオーバーライドとともに設定する
- 例：ワイルドカードがオンにされている検索の精度の向上
- 例：データの正規化
- 例：単語内の検索
- 例：symbol 分類の使い方

28.6.1 例：フィールドをトークナイザーオーバーライドとともに設定する

次のクエリでは、Admin API を使用してフィールド、フィールドレンジインデックス、およびカスタムのトークナイザーオーバーライドを作成する方法を示しています。これらの操作は、Admin 画面から実行することもできます。

XQuery を使用してこのセクションの以降の例で使用されているフィールドを設定する場合は、この例をテンプレートとして使用してください。データベースの名前、フィールドの名前、含まれる要素の名前、およびトークナイザーオーバーライドは各自の状況に合った設定で置き換えてください。

```
(: 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"))
  )
);
```

28.6.2 例：ワイルドカードがオンにされている検索の精度の向上

この例では、データベースで 3 文字検索をオンにしてワイルドカード検索をサポートしている場合に、カスタムトークナイザーを使用して、電話番号についてのフィルタリングなしの検索の精度と効率を向上させる方法を示しています。

Query Console で次のクエリを実行して、サンプルデータをデータベースに読み込みます。

```
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>)
```

Admin 画面または次のようなクエリを使用して、3 文字検索をオンにしてデータベースでのワイルドカード検索をサポートできるようにします。

```
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())
)
```

3 文字検索がオンにされると、次のフィルタリングなしの検索では誤検出（false positives）が返されます。これはクエリテキストがトークン化されて 2 語のトークン「202」および「456」になり、ワイルドカードが「456」トークンにのみ適用されているためです。

```
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
```

検索の精度を向上させるために、phone 要素に対してフィールドを定義し、さらにフィールド値からすべての句読点文字を除外するトークナイザーオーバーライドを定義します。Admin 画面を使用して次の特性を持つフィールドを定義するか、または「例：フィールドをトークナイザーオーバーライドとともに設定する」（484 ページ）にあるクエリを修正し、Query Console で実行します。

フィールドプロパティ	設定
field name	phone
field type	root

フィールドプロパティ	設定
include root	`false`
included elements	`phone`（名前空間なし）
range field index	scalar type：`string` field name：`phone` collation：`http://marklogic.com/collation/`（デフォルト） range value positions：`false`（デフォルト） invalid values：`reject`（デフォルト）
tokenizer overrides	`remove (`（左丸括弧） `remove )`（右丸括弧） `remove -`（ハイフン）

このフィールド定義により、電話番号が「4562021111」、「2021114560」など、1 つの単語トークンとしてインデックス付けされます。次の検索を実行した場合、誤検出が除外されます。

```
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
```

フィールド定義にトークナイザーオーバーライドが含まれない場合、フィールド語クエリには、最初のワードクエリと同じ誤検出が含まれます。

28.6.3 例：データの正規化

「例：ワイルドカードがオンにされている検索の精度の向上」（485 ページ）では、カスタムのトークナイザーオーバーライドを使用して、形式（202）456-1111 の電話番号から句読点を削除しています。トークナイザーオーバーライドは、コンテンツだけでなくクエリテキストにも適用されるので、オーバーライドにより、クエリテキストの正規化というメリットがさらに得られます。

"（"、"）"、"-"、" "（スペース）を `remove` 文字として定義した場合、（202）456-1111 などの電話番号は 1 つの語 2024561111 としてインデックス付けされるので、フィルタリングなしの検索では、次のすべてのクエリテキストの例が完全一致します。

- (202)456-1111
- 202-456-1111

- 202 456-1111
- 2024561111

また、同じオーバーライドで、読み込みの際にインデックス付けも正規化されます。phone 要素で入力データに前述のすべてのパターンが含まれている場合、すべての場合においてインデックス付けのためにデータが 1 つの単語トークンとして正規化されます。

入力のサンプルと、該当するフィールドの設定の詳細については、「例：ワイルドカードがオンにされている検索の精度の向上」（485 ページ）を参照してください。

28.6.4 例：単語内の検索

この例では、カスタムのトークナイザーオーバーライドを使用して、それ以外の場合では 1 つの単語と見なされるテキストから複数のトークンを作成する方法を示しています。これにより、単語の一部分をちゃんと検索できるようになります。

MxN の形式の dimensions 要素を含む入力ドキュメントを考えてみましょう。ここで、M と N はフィートです。例えば「10x4」は、10 フィート x4 フィートである領域を示す測定値です。10x4 は 1 つの単語としてトークン化されているため、「少なくとも 1 辺の長さが 10 フィートであることを含むすべてのドキュメント」を検索することはできません。

これを示すために、Query Console で次のクエリを実行してサンプルのドキュメントを読み込みます。

```
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>)
```

次に、データベースに対して次のワードクエリを実行し、マッチするものがないことを確認します。

```
xquery version "1.0-ml";
cts:search(fn:doc(), cts:word-query("10"),"unfiltered")
```

Admin 画面を使用して次の特性を持つフィールドを定義するか、または「例：フィールドをトークナイザーオーバーライドとともに設定する」（484 ページ）にあるクエリを修正し、Query Console で実行します。

フィールドプロパティ	設定
field name	`dim`
field type	`root`
include root	`false`
included elements	`dimensions`（名前空間なし）
range field index	scalar type：`string` field name：`dim` collation：`http://marklogic.com/collation/`（デフォルト） range value positions：`false`（デフォルト） invalid values：`reject`（デフォルト）
tokenizer overrides	`space x`

このフィールドでは、各 `dimension` テキストノードが「x」を境に 2 つのトークンに分割されます。したがって、次のフィールワードクエリにより、この例のドキュメント内でマッチするものが見つかります。

```
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>
```

28.6.5 例：symbol 分類の使い方

この例では、いくつかの文字を symbol に分類する値を示しています。ツイッターデータを操作する場合を考えてみましょう。ツイートテキストに出現する @word はユーザーを表し、#word はトピック識別子（「ハッシュタグ」）を表します。この例では、次の検索セマンティックを適用するものとします。

- NASA のような何もついていないタームを検索する場合、検索では、このターム自体（「NASA」）またはトピック（「#NASA」）とマッチさせるようにするが、ユーザー（「@NASA」）とはマッチさせないようにする必要があります。
- ユーザー（@NASA）を検索する場合は、ターム自体でもトピックでもなく、ユーザーとのみマッチさせる必要があります。
- トピック（#NASA）を検索する場合は、ターム自体でもユーザーでもなく、トピックとのみマッチさせる必要があります。

次の表に、望ましい検索結果をまとめます。

クエリテキスト	マッチさせるべき語句	マッチさせるべきでない語句
NASA	NASA #NASA	@NASA
@NASA	@NASA	NASA #NASA
#NASA	#NASA	NASA @NASA

トークンのオーバーライドを定義しなかった場合、ターム NASA、@NASA、および #NASA は、次のようにトークン化されます。

- NASA：`cts:word("NASA")`
- @NASA：`cts:punctuation("@"), cts:word("NASA")`
- #NASA：`cts:punctuation("#"), cts:word("NASA")`

句読点の区別がない（punctuation-insensitive）検索だとすると、これは、3 つのクエリ文字列すべてが NASA のみの検索となっていることを意味します。

@ を単語の文字に分類する、@ に対するトークナイザーオーバーライドを定義すると、@NASA は 1 つの単語としてトークン化され、ターム自体ともトピックともマッチしません。つまり、@NASA は次のようにトークン化されます。

```
cts:word("@NASA")
```

一方、# を単語の文字に分類しても、望ましい効果が得られません。これにより、クエリテキスト #NASA はトピックとマッチされます（これは意図したとおりです）が、ターム自体とのマッチが除外されることにもなります。解決方法は、# を記号に分類することです。これにより、次のトークン化が行われます。

```
cts:word("#"),cts:word("NASA")
```

これで、#NASA の検索では # と NASA の隣接の出現部分とマッチされ（これは、トピックの場合と同様です）、NASA のみの検索ではトピックならびにターム自体の両方とマッチされるようになりました。ユーザー（@NASA）は、@ に対するトークナイザーオーバーライドのため、引き続き除外されます。

29.0 エンコーディングとコレーション

MarkLogic サーバーでは、「MarkLogic サーバーの言語サポート」（469 ページ）に記載されている言語のサポートに加えて、多くの文字エンコーディング（文字コード）がサポートされ、さまざまなコレーションでコンテンツをソートできます。この章では、MarkLogic サーバーにおけるエンコーディングとコレーションのサポートについて説明します。次のセクションから構成されます。

- 文字エンコーディング
- コレーション
- 言語ごとのコレーションと文字セット

29.1 文字エンコーディング

MarkLogic サーバーでは、すべてのコンテンツを UTF-8 エンコーディングで格納します。UTF-8 以外のコンテンツを UTF-8 に変換せずに MarkLogic サーバーに読み込もうとすると、サーバーで例外がスローされます。UTF-8 以外のコンテンツが存在する場合は、読み込みの際にコンテンツのエンコーディングを指定できます。こうすることで、MarkLogic サーバーにより、コンテンツが UTF-8 に変換されます。エンコーディングを指定するには、`xdmp:document-load`、`xdmp:document-get`、および `xdmp:zip-get` において `encoding` オプションを使用します。このオプションにより、コンテンツのエンコーディングが MarkLogic サーバーに伝えられ、MarkLogic サーバーは、そのエンコーディングを UTF-8 に変換しようとします。コンテンツを変換できない場合、MarkLogic サーバーにより、UTF-8 以外のコンテンツが存在することを示す例外がスローされます。

文字エンコーディング、および `xdmp:document-load` 関数、`xdmp:document-get` 関数、`xdmp:unquote` 関数、`xdmp:zip-get` 関数における `encodings` オプションについては、次のことに注意してください。

- これらの関数は常に、コンテンツを UTF-8 に変換します。
- オプションが指定されておらず、HTTP ヘッダがない場合、エンコーディングがデフォルトで UTF-8 になります。
- オプションが指定されておらず、エンコーディングを指定している HTTP ヘッダがある場合、そのエンコーディングが使用されます。
- それ以外の場合は、これらの関数のエンコーディングはデフォルトで UTF-8 になります。
- エンコーディングが UTF-8 であるときに UTF-8 以外の文字が検出された場合、コンテンツに UTF-8 以外の文字が含まれていることを示す例外がスローされます。

- `encoding` オプションは、`xdmp:document-load`、`xdmp:document-get`、および `xdmp:zip-get` で利用可能です。
- MarkLogic サーバーでは、ユーザーが指定した文字セットを実際のコンテンツの文字セットだと見なします。実際の文字のエンコーディングとは異なるエンコーディングを指定した場合、どうなるかは予想できません。例外が発生する場合もあれば、誤った文字が表示される場合もあります。つまり、エンコーディングが間違って指定されている場合、正しくない文字が生成される可能性があります。

エンコーディングオプションのシンタックスの詳細については、『MarkLogic XQuery and XSLT Function Reference』を参照してください。

29.2 コレーション

このセクションでは、MarkLogic サーバーでのコレーションについて説明します。コレーションは、文字列のソート順および比較方法を指定するものです。このセクションは、次のパートから構成されます。

- コレーションの概要
- 一般的な 2 つのコレーションの URI
- コレーションの URI のシンタックス
- 3.1 のレンジインデックスとレキシコンとの後方互換性
- UCA ルートコレーション
- コレーションのデフォルトの決定方法
- コレーションの指定

29.2.1 コレーションの概要

*コレーション*は、文字列のソート順を指定します。コレーション設定により、(暗黙的または明示的に) 順序を指定する操作、およびレンジインデックスを使用する操作の順序が決定されます。順序を指定する操作の例としては、`order by` 節を含む XQuery ステートメントや、順序を比較する XQuery 標準関数 (例えば、`fn:compare`、`fn:substring-after`、`fn:substring-before` など)、レキシコン関数 (例えば、`cts:words`、`cts:element-word-match`、`cts:element-values` など) があります。また、コレーションは、文字列比較における一意性 (どの文字列を「同じもの」として扱うのか) を決定します。このため、あるコレーションによれば等しい 2 つの文字列が、他のコレーションによれば等しくない場合があります。

コードポイント順序コレーションでは、Unicode コードポイント順序に従ってソートされます（言語固有の情報は考慮されません）。他にも、言語固有のソートを指定するためによく使用されるコレーションがあります。例えば、コードポイントソートでは、すべての大文字が小文字よりも前に置かれるので、単語 `Zounds` は単語 `abracadabra` よりも前にソートされます。大文字と小文字を一緒にソートするコレーション（例えば、`A a B b C c` という順序など）を使用した場合、`abracadabra` は `Zounds` よりも前にソートされます。

コレーションは、URI（例えば、`http://marklogic.com/collation/`）で指定します。このコレーションの URI は MarkLogic サーバーに固有ですが、Unicode コレーション標準に従ってコレーションを指定しています。コレーションには多くのバリエーションがあり、さまざまな言語における好みと伝統に基づいて多くのソート順序があります。次のセクションでは、コレーションの URI のシンタックスについて説明します。使用可能なコレーションの URI の数は膨大ですが、ほとんどのアプリケーションでは、使用するコレーションは少数で済みます。コレーションの詳細については、http://icu.sourceforge.net/userguide/Collate_Concepts.html を参照してください。

29.2.2 一般的な 2 つのコレーションの URI

次に、MarkLogic サーバーで通常使用される 2 つのコレーションの URI を示します。

- `http://marklogic.com/collation/`
- `http://marklogic.com/collation/codepoint`

1 つ目のコレーションは、UCA ルートコレーション（「UCA ルートコレーション」（499 ページ）を参照してください）であり、システムのデフォルトです。2 つ目のコレーションは、コードポイント順序コレーションであり、MarkLogic サーバーの 3.2 よりも前のリリースではデフォルトでした。

29.2.3 コレーションの URI のシンタックス

MarkLogic サーバーでのコレーションは、URI で指定します。すべてのコレーションは、文字列 `http://marklogic.com/collation/` から始まります。コレーションのシンタックスは、次のとおりです。

```
http://marklogic.com/collation/<locale>[/<attribute>]*
```

このセクションでは、シンタックスの次の部分について説明します。

- コレーションの URI のロケール部分
- コレーションの URI の属性部分

29.2.3.1 コレーションの URI のロケール部分

コレーションの URI の <locale> 部分は有効なロケールであることが必要で、次のように定義します。

```
<locale> ::= <language>[-<script>][_<region>][@(collation=<value>;)+]
```

有効な言語コードのリストについては、次を参照してください。

```
http://www.loc.gov/standards/iso639-2/php/code_list.php
```

有効なスクリプトコードのリストについては、次を参照してください。

```
http://www.unicode.org/iso15924/iso15924-codes.html
```

有効な地域コードのリストについては、次を参照してください。

```
http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html
```

言語の中には、ロケールで指定できるコレーションが複数あるものもあります（例えば、ドイツ語や中国語）。これらの言語固有のコレーションのバリエーションを指定するには、シンタックスの `@collation=<value>` 部分を使用します。

コレーションの URI でロケールを指定しなかった場合、デフォルトでは、UCA ルートコレーションが使用されます（詳細については、「UCA ルートコレーション」（499 ページ）を参照してください）。

注： 数多くの種類の有効な言語、スクリプト、または地域コードを指定できますが、MarkLogic サーバーでフルサポートになっているのは、この製品でサポートされている言語に関連するもののうち、最も一般的に使用されているものだけです。サポートされている言語とその一般的なコレーションのリストについては、「言語ごとのコレーションと文字セット」（502 ページ）を参照してください。

次の表に、いくつかの典型的なロケールを簡単な説明とともにリストします。

ロケール	説明	コレーションの URI
en	英語	http://marklogic.com/collation/en
en_US	英語、米国地域	http://marklogic.com/collation/en_US
zh	中国語	http://marklogic.com/collation/zh
de@collation=phonebook	ドイツ語、phonebook コレーション	http://marklogic.com/collation/de@collation=phonebook

29.2.3.2 コレーションの URI の属性部分

コレーションの URI には、0 個以上の <attribute> を使用できます。属性ではさらに、使用するコレーションや、大文字 / 小文字の区別あり / なしなど、特性を指定します。指定する必要があるのは、属性が、指定するロケールのデフォルトと異なる場合だけです。属性のシンタックスは次のとおりです。

```
<attribute> ::= <strength> | <case-level> | <case-first> |
                <alternate> | <numeric-collation> |
                <variable-top> | <normalization-checking> |
                <french> | <hiragana>
```

次の表で、さまざまな属性について説明します。簡潔にするために、大文字 / 小文字の区別あり（case-sensitive）、発音符号による区別あり（diacritic-sensitive）などの用語を使用します。実際には、コレーションで使用するこれらの用語の定義は、これよりもある程度複雑です。各属性の正確な専門的意味については、http://icu.sourceforge.net/userguide/Collate_Concepts.html を参照してください。

属性	有効な値	説明
<strength> 使用する比較のレベル。	S1	大文字 / 小文字の区別なし、発音符号による区別なしを指定します。
	S2	発音符号による区別あり、大文字 / 小文字の区別なしを指定します。
	S3	大文字 / 小文字の区別あり、発音符号による区別ありを指定します。
	S4	句読点による区別ありを指定します。
	SI	アイデンティティ（識別されたコードポイント）を指定します。
<case-level> 大文字 / 小文字の区別ありのレベルをオンまたはオフにします。発音符号による区別ありのレベルはスキップします。したがって、発音符号による区別なし、大文字 / 小文字の区別ありの場合、/S1/EO になります。 デフォルト：EX	EO	大文字 / 小文字のレベルをオンにするように指定します。
	EX	大文字 / 小文字のレベルをオフにするように指定します。
<case-first> 大文字を小文字の前と後のいずれにソートするかを指定します。 デフォルト：CX	CU	大文字を最初にソートするように指定します。
	CL	小文字を最初にソートするように指定します。
	CX	オフ。

属性	有効な値	説明
<alternate> 可変文字を処理する方法を指定します（完全に無視する、あるいは通常の文字として扱う）。 デフォルト：AN	AN	すべての文字を無視できないように指定します。つまり、文字のソート時にすべてのスペースと句読点文字を含めます。
	AS	可変文字を variable-top 設定に従ってシフト（無視）するように指定します。
<numeric-collation> 数を、コレーション順序ではなく数として順序付けます（例えば 20 < 100）。 デフォルト：MX	MO	数値による順序付けを指定します。
	MX	数値ではない順序付けを指定します（コレーションに従って順序付けします）。
<variable-top> alternate とともに使用して、どの可変文字を無視するのかを指定します。カットオフ文字と比較して primary-less-than（この概念の詳細については、「UCA ルートコレーション」（499 ページ）の Unicode リンクを参照）であるあらゆる文字が無視されます。AS と組み合わせた場合にのみ、意味を持ちます。 デフォルト：T0000	T0000	すべての可変文字（通常は、スペースと句読点）が可変文字のソートにおいて無視されるように指定します。
	T0020	文字のソート時にスペースを無視するように指定します。例えば、/T0020/AS は、ピリオド（可変文字）は通常の文字として扱うが、スペースは無視されることを意味します。したがって、次のようになります。 A B = AB and AB < A.B.
	T00BB	文字のソート時にほとんどの句読点とスペース文字が無視されるように指定します。特に、ソートキーが 00BB 以下である文字が無視されます。
<normalization-checking> 入力文字列で Unicode 正規化を実行するかどうかを指定します。 デフォルト：NX	NO	Unicode を正規化します。
	NX	Unicode を正規化しません。

属性	有効な値	説明
<french> フランス語のアクセント順序付け規則を適用するかどうか（つまり、S3 レベルでの順序付けを逆にするかどうか）を指定します。 デフォルト：FX	FO	フランス語のアクセント順序付けを指定します。
	FX	（コレーションに従った）通常の順序付けを指定します。
<hiragana> ひらがなをカタカナと区別するためのレベルを追加するかどうかを指定します。 デフォルト：HX	HO	ひらがなモードをオンにします。
	HX	ひらがなモードをオフにします。

29.2.4 3.1 のレンジインデックスとレキシコンとの後方互換性

MarkLogic サーバー 3.1 で作成されたレンジインデックスとレキシコンでは、Unicode コードポイントコレーション順序を使用しています。これらのインデックスやレキシコンのいずれかで異なるコレーションを使用する場合は、コレーションを変更し、インデックスを再作成してから、データベースの再インデックス付けを行う必要があります（reindex enable が true に設定されている場合、再インデックス付けは自動的に開始されます）。

29.2.5 UCA ルートコレーション

MarkLogic サーバーでは、デフォルトが存在しない場合、Unicode コレーションアルゴリズム（UCA）ルートコレーションが使用されます。これは、S3（大文字 / 小文字の区別あり、発音符号による区別あり）の強度を持つ Unicode コードポイントコレーションを使用するもので、URI は次のとおりです。

```
http://marklogic.com/collation/
```

UCA ルートコレーションでは、Unicode コードポイント順序により有用な大文字 / 小文字の区別の有無と発音符号による区別の有無が追加されるため、大文字 / 小文字の区別の有無および発音符号による区別の有無を考慮する場合に、より妥当性のあるソート順序になります。UCA の詳細については、http://www.unicode.org/unicode/reports/tr10/ を参照してください。

29.2.6 コレーションのデフォルトの決定方法

MarkLogic サーバーでリクエストに使用されるコレーションは、Admin 画面のさまざまなパラメータの設定、および XQuery コードでの指定内容に基づきます。各アプリケーションサーバーでは、デフォルトのコレーションが指定されており、そのコレーションは優先される他のコレーションがない場合に使用されます。コレーションとそのデフォルト設定について、次のことに注意してください。

- コレーションは、アプリケーションサーバーレベル、レンジインデックス、およびレキシコンで指定します。
- 3.1 からアップグレードされたアプリケーションサーバー、レンジインデックス、およびレキシコンでは、コードポイント順序（`http://marklogic.com/collation/codepoint`）のままです。
- 新しいアプリケーションサーバーでは、デフォルトで UCA ルートコレーション（`http://marklogic.com/collation/`）になります。
- 新しいレンジインデックスおよびレキシコンでは、デフォルトで UCA ルートコレーション（`http://marklogic.com/collation/`）になります。
- XQuery プロローグ内でデフォルトのコレーションを指定できます。このコレーションは、アプリケーションサーバーのデフォルトよりも優先されます。例えば、次のクエリではフランス語のコレーションが使用されます。

```
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
```

- コードポイントコレーションの URI は、次のとおりです。

```
http://marklogic.com/collation/codepoint
```

  次に、（`1.0` の厳格な XQuery ダイアレクトで使用される）コードポイントコレーションの URI のエイリアスを示します。

```
http://www.w3.org/2005/xpath-functions/collation/codepoint
```

- Admin 画面に表示されるコレーションの URI は、入力した URI の基準表現として格納および表示されます。基準表現は、入力された URI と等価ですが、順序を変更し、コレーションの URI 文字列の部分を事前に決定された順序にシンプル化したものです。ビルトインの XQuery 関数 `xdmp:collation-canonical-uri` は、任意の有効なコレーションの URI の基準 URI を返します。
- 空の文字列 URI は、コードポイントコレーションになります。したがって、次の場合、次のような結果を返します。

```
xdmp:collation-canonical-uri("")
=> http://marklogic.com/collation/codepoint
```

- XQuery モジュールで使用されるコレーションは、モジュール単位で決定されます。したがって、モジュールは、異なるコレーションを使用する他のモジュールを呼び出すことがあります。これは、それぞれのモジュールが、呼び出し元のモジュールに依存することなく（アプリケーションサーバーのデフォルト、コレーションのプロローグ宣言などに基づいて）コレーションを決定するためです。
- モジュールが他のモジュールから起動または生成された場合、またはリクエストが他のモジュールから `xdmp:eval` 呼び出しによって送信された場合、新しいリクエストでは、呼び出し元モジュールのコレーションコンテキストが継承されます。このコンテキストは、（例えばプロローグ内の `declare default collation` 式によって）クエリ内で上書きすることもできますが、デフォルトで呼び出し元モジュールからのコンテキストになります。
- 有効であるコレーションが他にない場合（例えば、スケジュール設定されたタスクの場合）、コードポイントコレーションが使用されます。

29.2.7 コレーションの指定

コレーションは多くの場所で指定できます。次に、コレーションを指定する一般的な場所をいくつか示します。

- FLWOR 式の `order by` 節。
- Admin 画面のアプリケーションサーバー設定。
- Admin 画面でのレキシコンまたはレンジインデックスの指定。
- 多くの W3C 標準 XQuery 関数（例えば、`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`）。
- レキシコン API（`cts:words`、`cts:word-match`、`cts:element-words`、`cts:element-values` など）。
- レンジクエリコンストラクタ（`cts:element-range-query`、`cts:element-attribute-range-query`）。

29.3 言語ごとのコレーションと文字セット

次の表に、MarkLogic サーバーで言語固有のトークン化とステミングがサポートされている言語をリストします。また、各言語で共通のコレーションと文字セットもリストします。

言語	ベースコレーション		文字セット
英語	http://marklogic.com/collation/en	大文字 / 小文字と発音符号による区別あり	ISO-8859-1 cp1252
	http://marklogic.com/collation/en/S1	大文字 / 小文字と発音符号による区別なし	
	http://marklogic.com/collation/en/S2	発音符号による区別あり	
	http://marklogic.com/collation/en/S1/EO	大文字 / 小文字の区別あり	
フランス語	http://marklogic.com/collation/fr	大文字 / 小文字と発音符号による区別あり	ISO-8859-1 cp1252
	http://marklogic.com/collation/fr/S1	大文字 / 小文字と発音符号による区別なし	
	http://marklogic.com/collation/fr/S2	発音符号による区別あり	
	http://marklogic.com/collation/fr/S1/EO	大文字 / 小文字の区別あり	

言語	ベースコレーション		文字セット
イタリア語	http://marklogic.com/collation/it	大文字 / 小文字と発音符号による区別あり	ISO-8859-1 cp1252
	http://marklogic.com/collation/it/S1	大文字 / 小文字と発音符号による区別なし	
	http://marklogic.com/collation/it/S2	発音符号による区別あり	
	http://marklogic.com/collation/it/S1/EO	大文字 / 小文字の区別あり	
ドイツ語	http://marklogic.com/collation/de	大文字 / 小文字と発音符号による区別あり	ISO-8859-1 cp1252
	http://marklogic.com/collation/de/S1	大文字 / 小文字と発音符号による区別なし	
	http://marklogic.com/collation/de/S2	発音符号による区別あり	
	http://marklogic.com/collation/de/S1/EO	大文字 / 小文字の区別あり	
	http://marklogic.com/collation/de@collation=phonebook	代替のドイツ語のコレーション	
スペイン語	http://marklogic.com/collation/es	大文字 / 小文字と発音符号による区別あり	ISO-8859-1 cp1252
	http://marklogic.com/collation/es/S1	大文字 / 小文字と発音符号による区別なし	
	http://marklogic.com/collation/es/S2	発音符号による区別あり	
	http://marklogic.com/collation/es/S1/EO	大文字 / 小文字の区別あり	
	http://marklogic.com/collation/es@collation=traditional	ll と ch を独立した文字として扱う	

言語	ベースコレーション		文字セット
ロシア語	http://marklogic.com/collation/ru	大文字 / 小文字と発音符号による区別あり	cp1251 KOI8-R ISO-8859-5
	http://marklogic.com/collation/ru/S1	大文字 / 小文字と発音符号による区別なし	
	http://marklogic.com/collation/ru/S2	発音符号による区別あり	
	http://marklogic.com/collation/ru/S1/EO	大文字 / 小文字の区別あり	
アラビア語	http://marklogic.com/collation/ar	異形による区別あり	cp1256 ISO-8859-6
	http://marklogic.com/collation/ar/S1	異形による区別なし	
中国語（簡体字および繁体字）	http://marklogic.com/collation/zh (simplified)	大文字 / 小文字と発音符号による区別あり	Simplified: GB18030 GB2312 EUC-CN hz-gb-2312 cp936 Traditional: Big5 Big5-HKSCS cp950 GB18030
	http://marklogic.com/collation/zh-Hant (traditional)	大文字 / 小文字と発音符号による区別あり	
	http://marklogic.com/collation/zh-Hant@collation=stroke (traditional with simplified order)	ロケール固有の異形	
	http://marklogic.com/collation/zh@collation=pinyan (simplified with traditional order)	ロケール固有の異形	
韓国語	http://marklogic.com/collation/ko	大文字 / 小文字と発音符号による区別あり	ISO 2022-KR EUC-KR KS X 1001 cp949 GB12052 KSC 5636
	http://marklogic.com/collation/ko/S1	大文字 / 小文字と発音符号による区別なし	

言語	ベースコレーション		文字セット
ペルシア語 (現代ペルシア語)	http://marklogic.com/collation/fa	大文字 / 小文字と発音符号による区別あり	cp1256 ISO-8859-6
	http://marklogic.com/collation/fa/S1	大文字 / 小文字と発音符号による区別なし	
	http://marklogic.com/collation/fa/S2	発音符号による区別あり	
	http://marklogic.com/collation/fa/NX	正規化をオフにする	
オランダ語	http://marklogic.com/collation/nl	大文字 / 小文字と発音符号による区別あり	ISO-8859-1 cp1252
	http://marklogic.com/collation/nl/S1	大文字 / 小文字と発音符号による区別なし	
	http://marklogic.com/collation/nl/S2	発音符号による区別あり	
	http://marklogic.com/collation/nl/S1/EO	大文字 / 小文字の区別あり	
日本語	http://marklogic.com/collation/ja http://marklogic.com/collation/ja/S1	大文字 / 小文字と発音符号による区別なし	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	発音符号による区別あり	
	http://marklogic.com/collation/ja/S1/EO	大文字 / 小文字の区別あり	
	http://marklogic.com/collation/ja/S4/HX	ひらがなモードをオフにする	

言語	ベースコレーション		文字セット
ポルトガル語	http://marklogic.com/collation/pt	大文字 / 小文字と発音符号による区別あり	ISO-8859-1 cp1252
	http://marklogic.com/collation/pt/S1	大文字 / 小文字と発音符号による区別なし	
	http://marklogic.com/collation/pt/S2	発音符号による区別あり	
	http://marklogic.com/collation/pt/S1/E0	大文字 / 小文字の区別あり	
ノルウェー語（ニーノシュクおよびブークモール)	http://marklogic.com/collation/nn (Nynorsk)	大文字 / 小文字と発音符号による区別あり	ISO-8859-1 cp1252
	http://marklogic.com/collation/nn/S1	大文字 / 小文字と発音符号による区別なし	
	http://marklogic.com/collation/nn/S2	発音符号による区別あり	
	http://marklogic.com/collation/nn/S1/EO	大文字 / 小文字の区別あり	
	http://marklogic.com/collation/nb (Bokmål)	大文字 / 小文字と発音符号による区別あり	
	http://marklogic.com/collation/nb/S1	大文字 / 小文字と発音符号による区別なし	
	http://marklogic.com/collation/nb/S2	発音符号による区別あり	
	http://marklogic.com/collation/nb/S1/EO	大文字 / 小文字の区別あり	

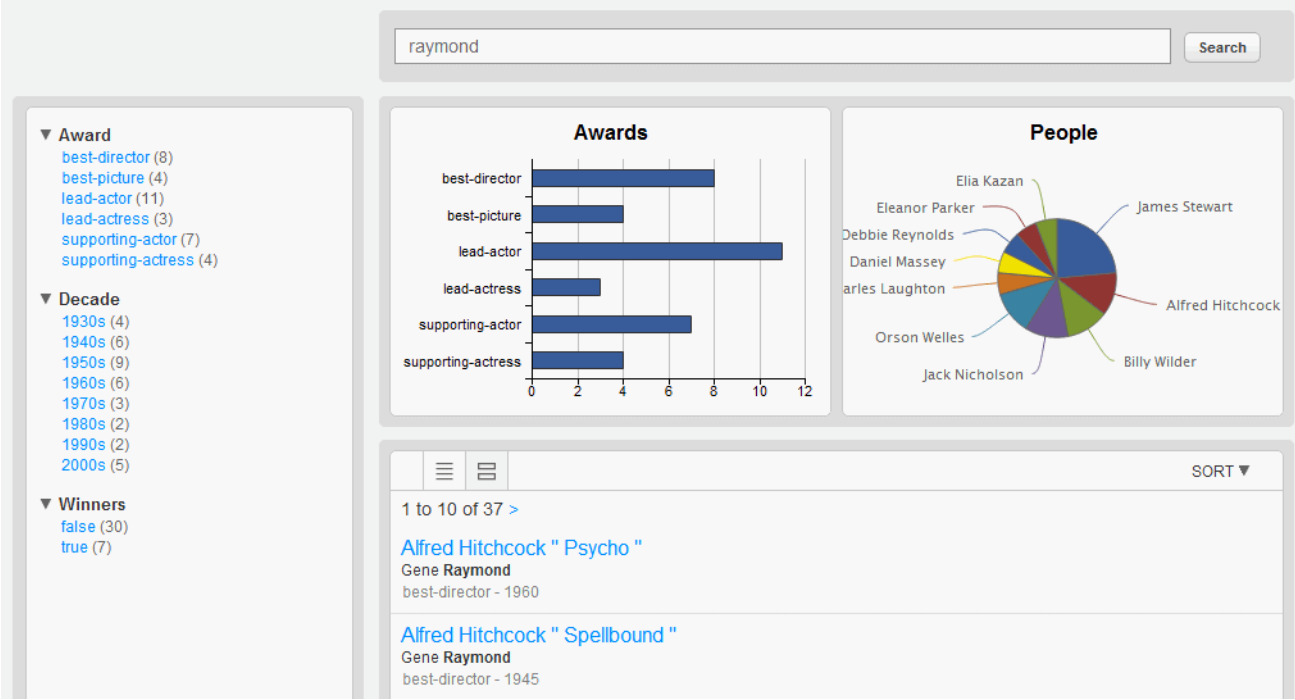
言語	ベースコレーション		文字セット
スウェーデン語	http://marklogic.com/collation/sv	大文字 / 小文字と発音符号による区別あり	ISO-8859-1 cp1252
	http://marklogic.com/collation/sv/S1	大文字 / 小文字と発音符号による区別なし	
	http://marklogic.com/collation/sv/S2	発音符号による区別あり	
	http://marklogic.com/collation/sv/S1/E0	大文字 / 小文字の区別あり	

英語を除くすべての言語は、ライセンスキーでオンにする必要があります。サポートされているいずれかの言語のライセンスキーがない場合、その言語は一般言語として扱われます。また、各単語はそれ自体にステミングされ、一般的な方法（アジア以外の文字ではスペースと句読点文字、アジアの文字ではそれぞれの文字に基づく方法）でトークン化されます。詳細については、「一般言語のサポート」（479 ページ）を参照してください。ライセンスキーがない言語においても、言語固有のコレーションは利用可能です。

30.0 データ視覚化ウィジェット

この章では、MarkLogic サーバーの検索結果を表示するための視覚化ウィジェットの使用方法について説明します。このウィジェットは、データを図またはグラフの形式で表示する手段を提供するものです。視覚化ウィジェットは、自分で構築したアプリケーションの表示出力に直接組み込こと も、Application Builder で作成したアプリケーションに組み込むこともできます。

次に、Application Builder によって作成された Oscars アプリケーションにおける検索ページを示します。ページ上のすべてのオブジェクトが視覚化ウィジェットです。



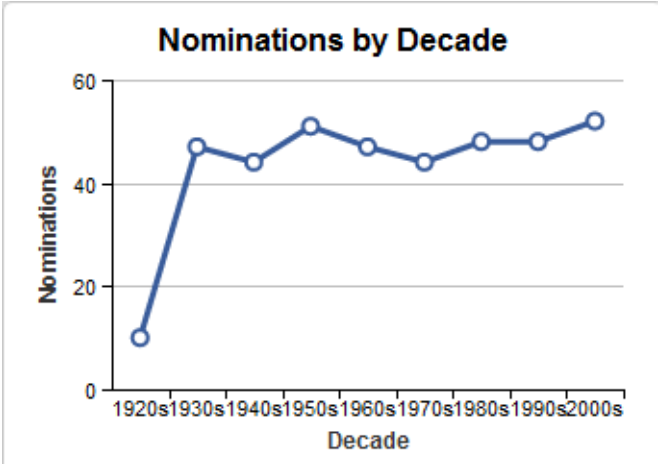
この章は、次のセクションを含んでいます。

- 視覚化ウィジェットの概要
- 視覚化ウィジェットの操作
- ウィジェットアーキテクチャの概要
- HTML ページへの視覚化ウィジェットの追加
- 例：アプリケーションへのウィジェットの追加
- 視覚化ウィジェットの制限
- プロキシをセットアップする
- ウィジェット API リファレンス

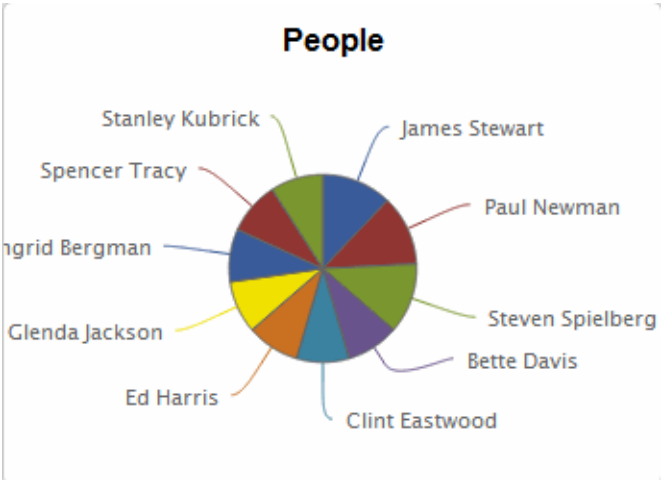
30.1　視覚化ウィジェットの概要

MarkLogic サーバーには、6 つのタイプの視覚化ウィジェットがあります。

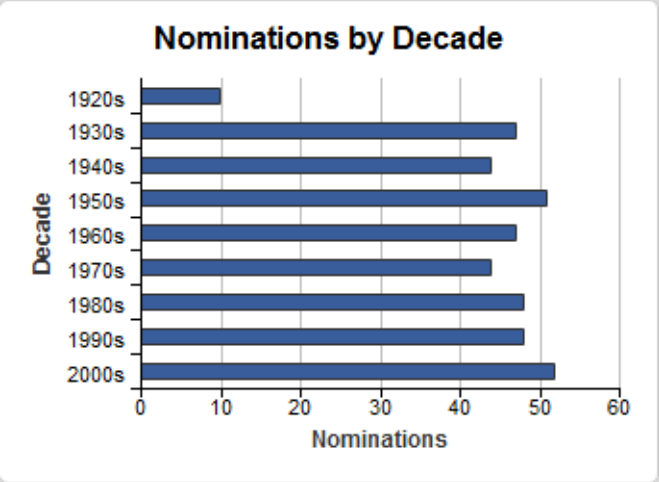
- 折れ線グラフ：2 つの連続する各データポイントを直線で結び、左から右へ一連のデータポイントを示す 2 次元のグラフ。折れ線グラフは、任意のファセットタイプに関連付けることができますが、通常は時間の経過による変化を表す線で時系列を示すために使用されます。



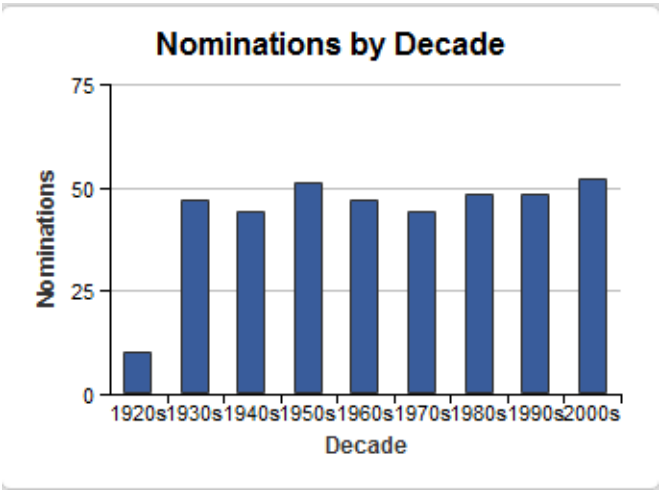
- 円グラフ：色分けされたセクション（データ値ごとに 1 つのセクション）に分割された 2 次元の円。円の面積のうちのセクションの割合は、データ全体におけるデータ値のコンテンツの割合に相当します（例えば、合計 16 個のデータのうち、8 個が値「red」を持つ場合、「red」値に対応するセクションは円の半分になります）。データ値は、非連続量（色、レストランの料理）にすることも、連続量（高さ、重さ、テストの点数）にすることもできます。



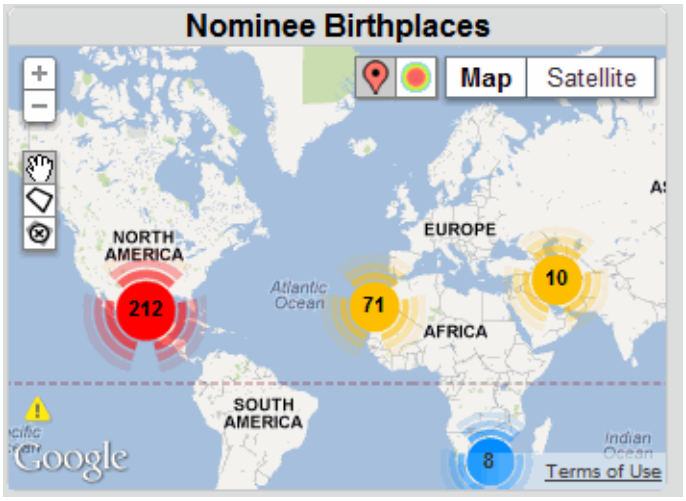
- 横棒グラフ：特定の離散値を表す水平方向の棒によるグラフ。値は、非連続量（色、レストランの料理）にすることも、連続量（高さ、重さ、テストの点数）にすることもできます。



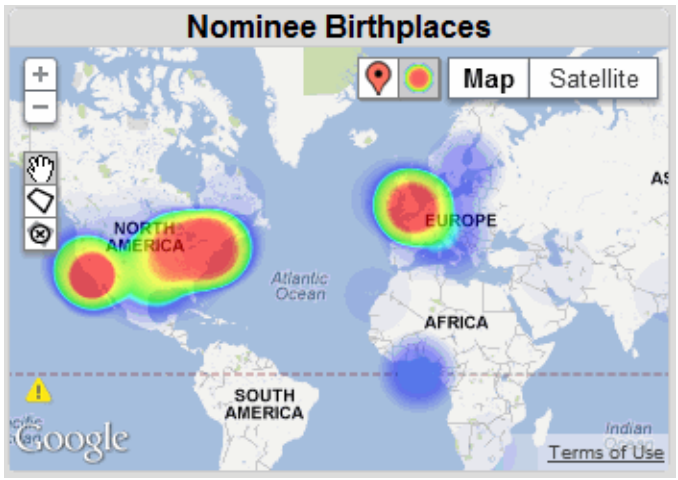
- 棒グラフ：特定の離散値を表す垂直方向の棒によるグラフ。値は、非連続量（色、レストランの料理）にすることも、連続量（高さ、重さ、テストの点数）にすることもできます。



- ポイントマップ：検索結果を表すデータポイントを含む Google マップ。各検索結果は、位置情報値に関連付けられています。例えば、データベース内のデータに対して地図ウィジェットが設定されているとすると、俳優 / 監督であるジョン・ヒューストンの出生地として、ミズーリ州ネバダにピンが表示されます。地図では、ズームインおよびズームアウトに加えて、全方向に移動できます。



- ヒートマップ：結果における各場所の頻度を色の違いとして表現した Google マップ。例えば、結果としてカリフォルニア州サンカルロスに関するものの頻度が大きかった場合、その場所を中心とした大きな色の塊が表示されます。ロサンジェルスに頻度が小さい結果がある場合、その場所を中心とする小さな色の塊が表示されます。



30.2 視覚化ウィジェットの操作

このセクションは、次のサブセクションを含んでいます。

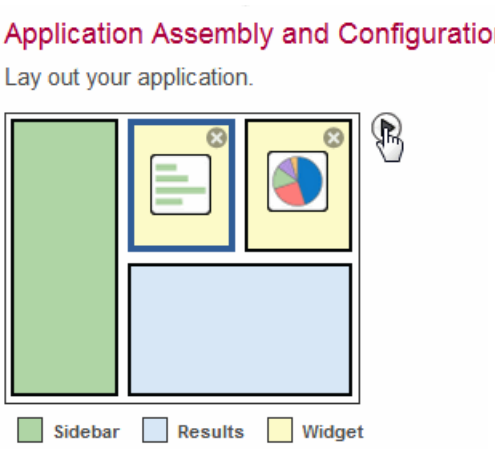
- Oscars サンプルアプリケーションの構築
- ウィジェットの全般的な操作
- 折れ線グラフウィジェットの操作
- 棒グラフウィジェットと横棒グラフウィジェットの操作
- 円グラフウィジェットの操作
- 地図ウィジェットの操作

30.2.1 Oscars サンプルアプリケーションの構築

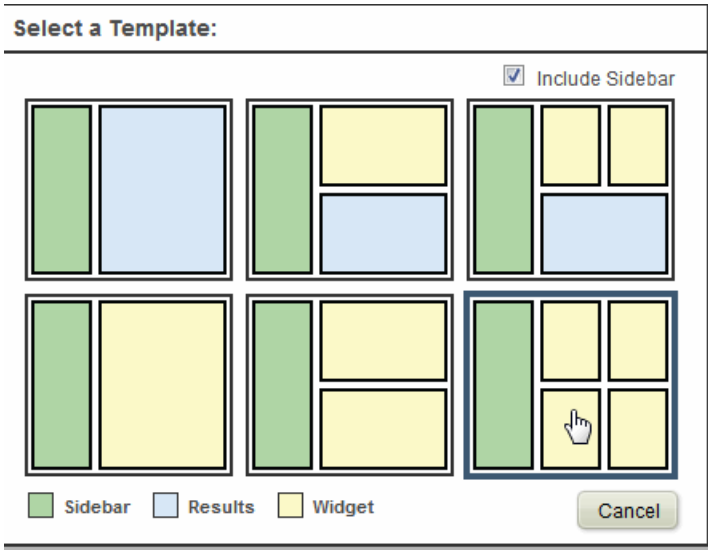
視覚化ウィジェットの使用方法を最もすばやく簡単に学習する方法は、Application Builder を使用して、『Application Builder Developer's Guide』の「Building the Oscars Sample Application」で説明されているように Oscars サンプルアプリケーションを構築するというものです。

Oscars アプリケーションにすべてのウィジェットタイプを追加するには、Application Builder で［Assemble］ページに移動し、次の手順を実行します。

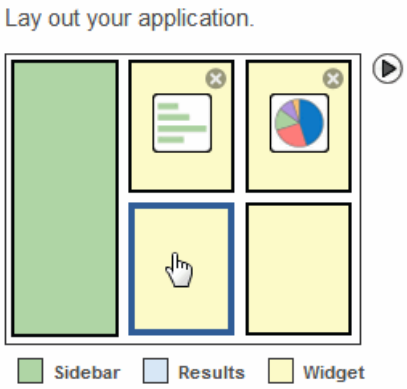
1. ページの左上にある［Layout your application］セクションの右側で、円で囲まれたポインタをクリックします。



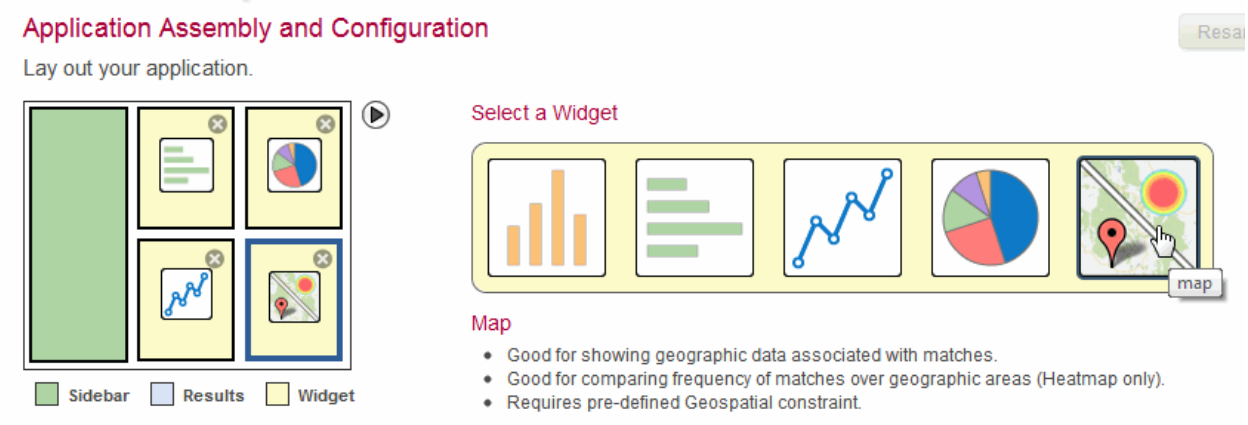
2. ［Select a Template］ポップアップウィンドウが表示され、このアプリケーションの結果ページのレイアウトの選択肢が示されます。右下にある 4 つのウィジェットのレイアウトを選択します。



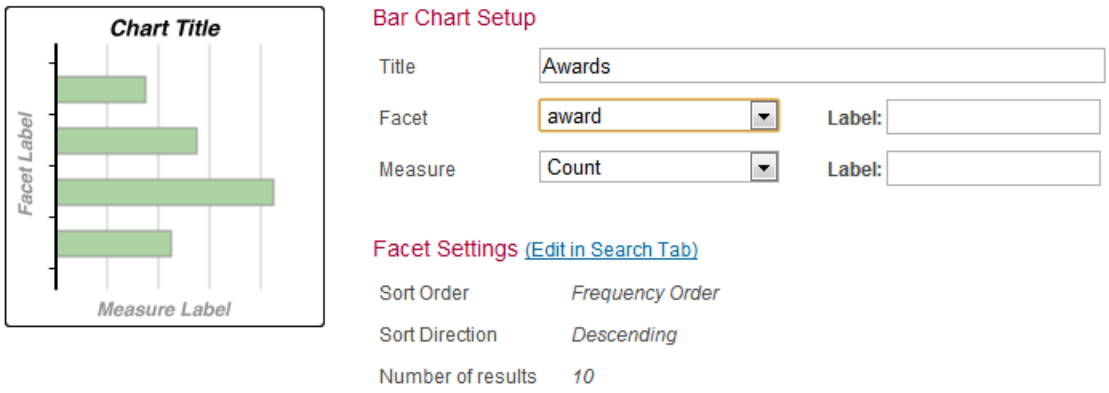
3. アプリケーションレイアウト内の各ウィジェットボックスをクリックします。



4. レイアウトに追加するウィジェットを右側で選択します。



5. 各ウィジェットはファセットに関連付ける必要があります。ファセットは、［Assemble］ページの下部で設定します。例えば、データセットの受賞回数を表す横棒グラフを作成するために、［Facet］に「award」、［Measure］に「Count」を設定します。



注： 現時点では、Count だけがウィジェットで利用可能です。

6. アプリケーションを、『Application Builder Developer's Guide』の「Deploy Page」で説明されているように公開します。

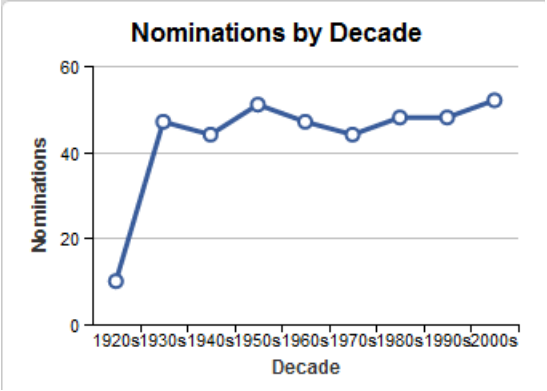
30.2.2 ウィジェットの全般的な操作

一般的に、ウィジェットの項目（例えば棒グラフの棒）をクリックすると、ページの検索クエリが変更され、その結果として、ページ上のサイドバー、結果、および他のウィジェットが新しい検索タームを反映するように更新されます。例外は、折れ線グラフウィジェット、地図ウィジェットでマーカーをクリックした場合だけです。

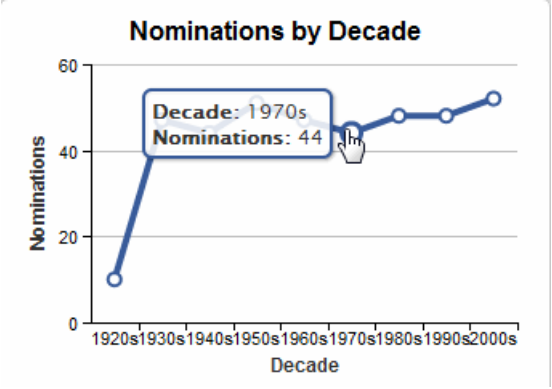
このウィジェットによる検索はステミング検索です。つまり、「name」のようなタームに対して、「names」、「named」などが結果として返されます。

30.2.3 折れ線グラフウィジェットの操作

次の折れ線グラフは、「decade」ファセットに関する検索結果を示しています。各ドットはその「decade」中のノミネーション（候補）の回数を表しています。グラフの Y 軸はノミネーションの回数、X 軸は年（1920 年の最初のオスカーから現在まで）です。



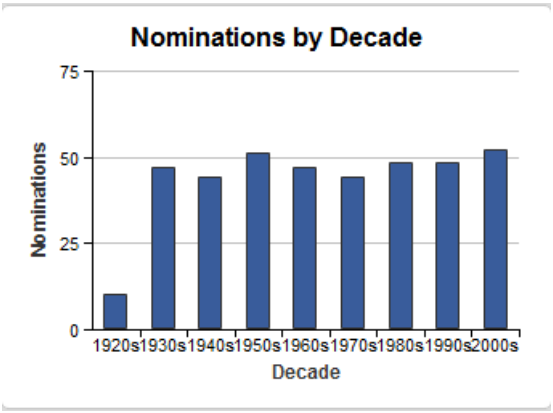
折れ線グラフ上にカーソルを移動すると、ツールヒントウィンドウが表示され、最も近いデータポイントに関連付けられているノミネーションの日付と回数が示されます（例えば、「Decade：1970s」と「Nominations：44」）。

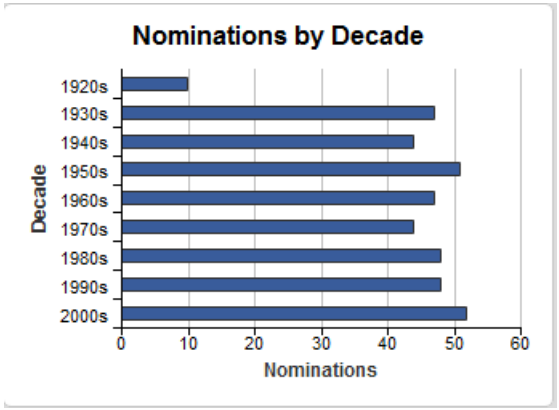


注： 折れ線グラフは現時点では、読み取り専用であり、ページ上のサイドバー、結果、その他のウィジェットを更新しません。

30.2.4 棒グラフウィジェットと横棒グラフウィジェットの操作

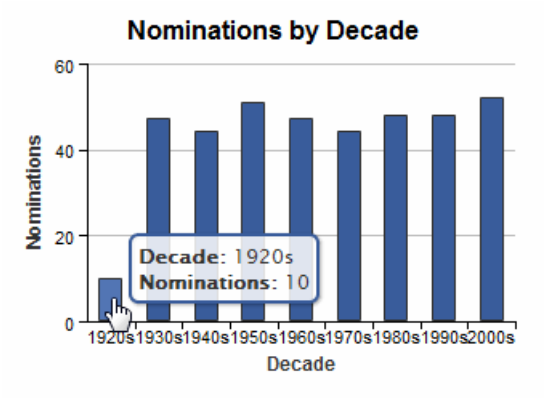
次の棒グラフと横棒グラフは、「decade」ファセットの検索結果を示しています。





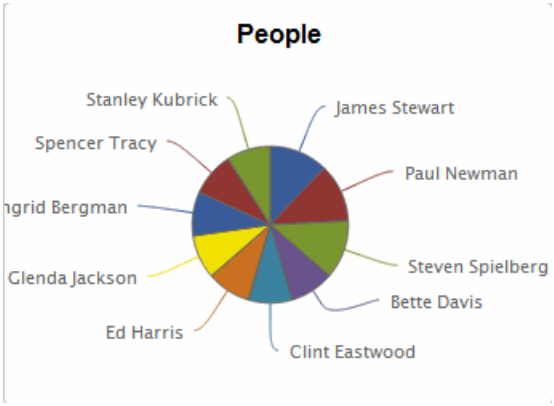
棒グラフまたは横棒グラフ上にカーソルを移動すると、ツールヒントウィンドウが表示され、最も近い棒に関連付けられている（グラフに応じて年代ごとまたは映画ごとのいずれかの）ノミネーションの回数、（やはり、グラフに応じて）映画名または年代が示されます。グラフ上でマウスをクリックしたりドラッグしたりしても、何も起きません。1 つの棒をクリックすると、グラフに応じて映画または年代に関連付けられている検索結果のみを表示するようにページの結果領域が変更されます。グラフをクリックすると、ページ上の他のウィジェットも変更され、クリックした棒の値が追加の検索タームとしてそれらのウィジェットに適用されます。

1920 年代の棒をクリックした次のグラフでは、結果の個数が 10 であり、これは 1920 年代に発生した結果の個数と同じです。

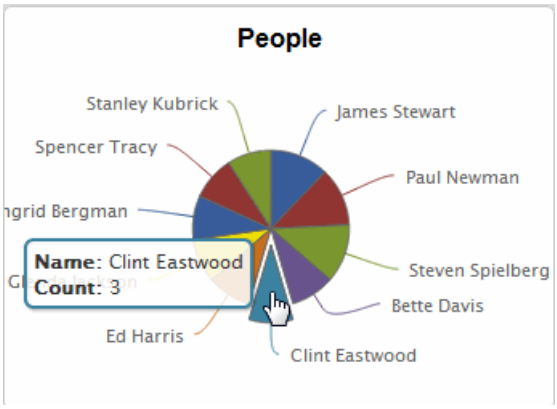


30.2.5　円グラフウィジェットの操作

次の円グラフでは、「name」ファセットに関する検索結果が示されています。

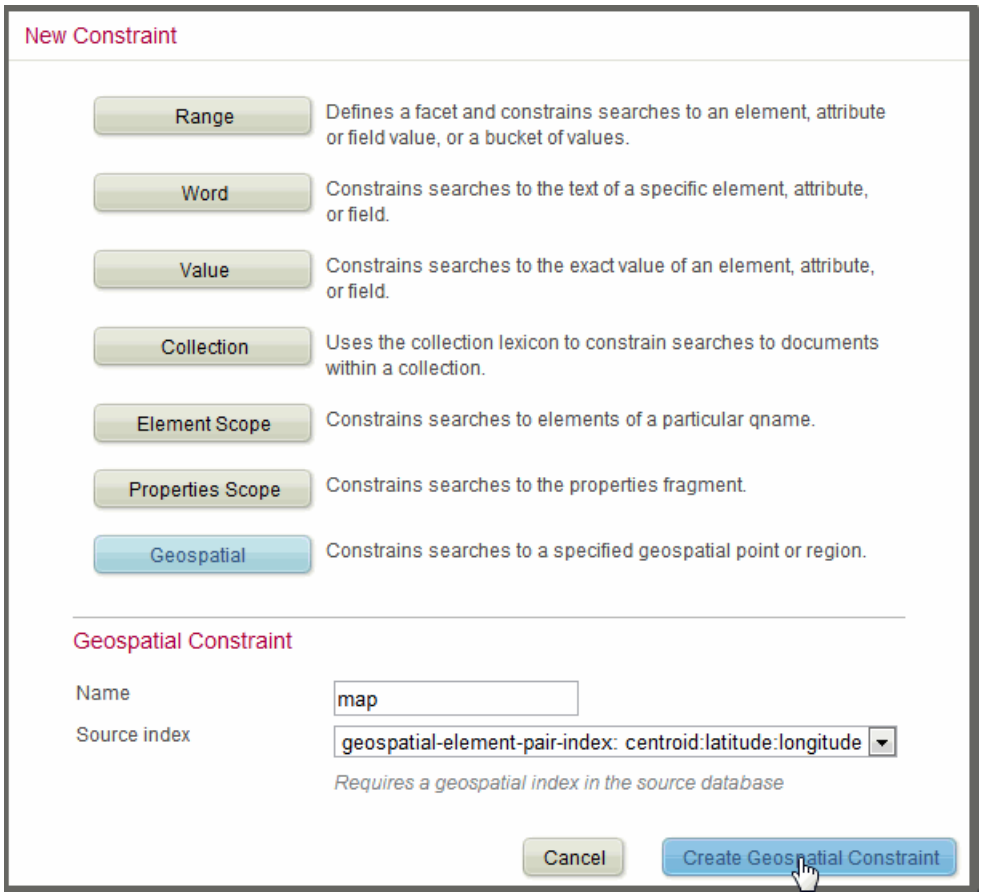


円グラフ上にカーソルを移動すると、ツールヒントウィンドウが表示され、最も近いセクションに関連付けられている名前値と、検索と一致するノミネーションの数が示されます（例えば「Clint Eastwood：3)。円グラフの 1 つのセクションをクリックすると、サイドバーと結果セクションが更新され、円グラフのそのセクションに関連付けられている名前値の結果のみが示されます。例えば、Clint Eastwood のセクションをクリックすると、Clint Eastwood に関連付けられている検索結果のみがページのサイドバーと結果セクションに表示されます。

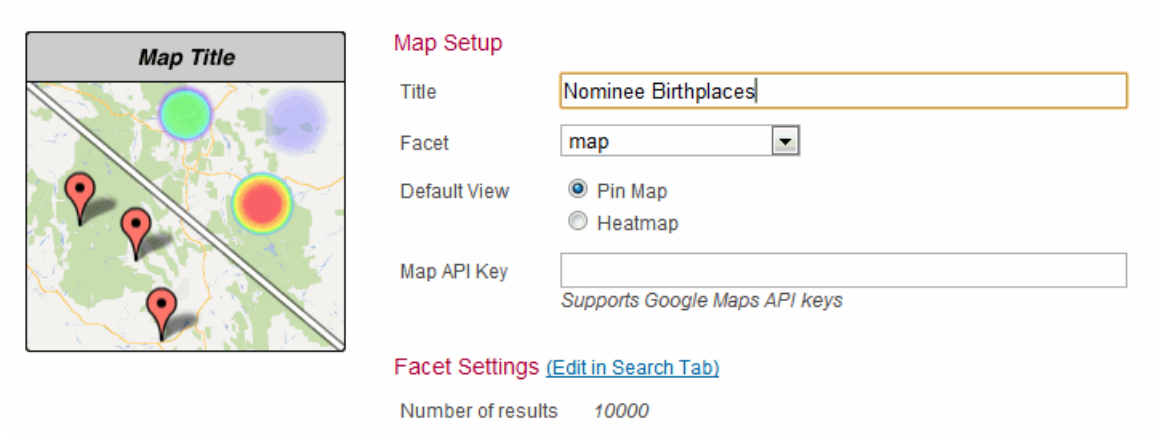


30.2.6　地図ウィジェットの操作

地図ウィジェットを選択するには、位置情報要素ペアインデックス（geospatial element pair index）を作成する必要があります。Application Builder の［Search］タブに移動し、ページの下部にある［Add New］をクリックし、［Geospatial］を選択し、「map」という名前を付けます。［Create Geospatial Constraint］をクリックします。

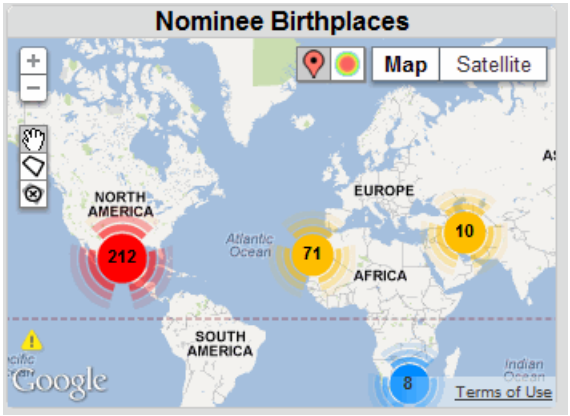


［Assemble］ページに移動し、地図ウィジェットを選択し、地図ファセットを割り当てます。アプリケーションを公開します。



注： 1 日あたりの地図の読み込みが 25,000 回を超えることが想定されるアプリケーションを構築する場合は、Google から Map API キーを購入する必要があります。キーは［Map Setup］セクションの［Map API Key］フィールドに入力します。

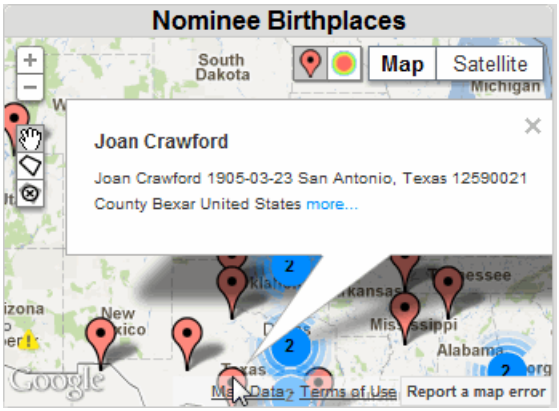
導入したアプリケーション内の地図ウィジェットには、Oscars データセット全体に関する検索結果が示されます。各円は、マーカーのクラスタを表します。また、マーカークラスタの色と表示されている数は、結果の個数を表します。ファセットを選択してデータセットを絞り込んだ場合、結果のサブセットを表すように地図が変更されます。



地図ウィジェットでマウスをクリックしたりドラッグしたりすると、地図の表示が移動します。マーカークラスタをクリックしていくと、より細かいマーカークラスタにドリルダウンし、最終的には 1 つ 1 つのマーカーまでドリルダウンできます。個々のマーカーは、ノミネートされた各個人の出生地を表します。



他のウィジェットとは異なり、マーカーをクリックしてもサイドバーも結果も更新されません。代わりに、「Joan Crawford / Born 1905-03-23 / San Antonio, Texas」など、俳優の名前、生年月日、出生地を含む情報ウィンドウが開かれます。



Draw Shape ツールを使用して地図上で 1 つあるいは複数のマーカーを選択した場合、サイドバーおよび結果が更新されます。地図を Draw Shape モードに設定するには、多角形のアイコンをクリックします。



例えば、2 つのマーカーに基づいて検索結果を更新するには、次の図に示すようにポイントを左クリックしてマーカーの周囲に線をドラッグして、多角形を描きます。



30.3　ウィジェットアーキテクチャの概要

ウィジェットのフロントエンドアーキテクチャは、メッセージの送受信ができるモジュールに基づいていますが、モジュールは、他のモジュールやページ全体についての情報を持ちません。各モジュールには、特定のパラメータを満たすメッセージを listen するリスナーがあります。同様に、モジュールは、メッセージの受信者に関する具体的な情報を持たずにメッセージを共通メディアに送出しますが、適切なリスナーを持つ他のモジュールがそのメッセージを受け取ります。この方法によって、アプリケーションのフローを乱すことなく、リアルタイムでモジュールの追加や削除ができます。また、コントローラでアプリケーションのすべてのモジュールを監視する必要がありません。これにより、利害関係がはっきりと分離され、単体テストをより簡単に行うことができます。

モジュールは、他のモジュールを認識しません。イベントをトリガーし、イベントを listen するだけです。モジュール間で行われる唯一の情報交換は、共通メディア内のメッセージの listen を使用することです。特定の情報ソースまたは特定の宛先をトラッキングすることはありません。メッセージは、リリースされて共通メディアに入り、その特定のメッセージをピックアップする適切なリスナーを持つモジュールによって読み取られます。

ウィジェットアーキテクチャモデルでは、DOM（ドキュメントオブジェクトモデル）のイベントバブリングを大量に利用します。モジュールには、ウィジェットモジュールとコントローラモジュールの 2 つの基本タイプがあります。ほとんどのインスタンスでは、1 つのコントローラモジュールと 1 つあるいは複数のウィジェットモジュールがあります。

このセクションは、次のトピックを含んでいます。

- ウィジェット
- コントローラ
- 識別
- ウィジェット内の相互作用
- シャドウクエリ

30.3.1 ウィジェット

ウィジェットは、イベントハンドラを使用して、レンダリングするデータの新しい断片を listen するために newData イベントを listen します。同様に、ウィジェットは、ウィジェットに対する内部変更に基づいてクエリで変更を伝達するために newQuery イベントを生成できます。

一般的なシンプルな 2 つの例として、検索ボックスウィジェットと結果ウィジェットがあります。

- 検索ボックスは、newData メッセージに応答しないウィジェットですが、ユーザーがテキストクエリを入力して［Search］をクリックした場合、newQuery メッセージを生成します。
- 結果ウィジェットは、通常は newQuery メッセージを送出せず、代わりにクエリの結果をレンダリングするだけです。結果がページネーションで表示される場合は例外であり、結果ウィジェットがページネーションのために newQuery メッセージを送信します。

ウィジェットは一般に、2 つのタイプのハイブリッドを使用します（いずれのタイプも、ビューを更新してクエリに入れるために、情報を表示し、そのクエリを更新します)。個々のウィジェットは、クエリの全体を必ずしも認識せず、その一部を制御する傾向があります。ほとんどの視覚化処理では、特定のファセットを利用し、そのファセットに基づいて検索にフィルタを適用するか、削除します。

30.3.2 コントローラ

コントローラは、ウィジェットとよく似たモジュールです。その主な役目は、newQuery イベントを listen し、newData イベントを投稿することです。コントローラは、新しいデータを得るためにサーバーに対して Ajax 呼び出しを行う役割を持つプライベートな comm（communicator）オブジェクトを利用します。例えば、地図ウィジェットでパン（移動）が起こると、地図ではマーカークラスタを更新する必要がありますが、これにはクエリが必要です。コントローラがヒアリングしていて comm オブジェクトに渡すイベントを、地図ウィジェットは送出します。comm オブジェクトはデータをリクエストし、地図ウィジェットに返します。

コントローラと comm オブジェクトにより、キャッシュ処理やデータ操作のような機能をいろいろと拡張できます。コントローラでは、サーバー、アプリケーションラッパー、ウィジェットクラスなどでアクセスするエンドポイントなどを指定するオプション設定オブジェクトが利用できます。制限されたアプリケーションラッパーを指定することにより、例えばページ上に、それぞれが個別のアプリケーションを実行する複数のコントローラを作成できます。これらは、アプリケーションコンテナ内のメッセージにのみ応答します。同様に、2 つのアプリケーションでページ上のコンテナを共有し、異なるウィジェットクラスを使用して個々のウィジェットを制御できます。

コントローラは、newQuery イベントを受信し、クエリオブジェクトを構造化クエリに変換します。この構造化クエリは、comm オブジェクト内で生成される Ajax 呼び出しに追加されます。

30.3.3 識別

ウィジェットは、HTML のクラスを使用して識別され、コントローラにリンクされます。Application Builder によって生成されたアプリケーションでは、このクラスの名前は「widget」になります。

例えば、ページに結果ウィジェットを追加するには、次のように指定します。

```
<div id="results" class="widget"></div>
```

30.3.4　ウィジェット内の相互作用

ウィジェット内の相互作用として、データの選択またはページネーションがあります。各ウィジェットには、`updateQuery` という内部メソッドがあります。このメソッドは、newQuery イベントを送出するために、クエリオブジェクトとともに呼び出すことができます。視覚的ウィジェットの場合、ファセットでの「選択」によりクエリが更新されます。横棒グラフまたは円グラフは単一の選択内容に基づき、地図は領域（多角形）に基づいています。例えば、横棒グラフの棒をクリックすると、DOM からバブルアップし、ウィジェットによって読み取られる選択イベントが生成されます。そして、ウィジェットにより、選択内容が newQuery イベントに変換されます。変換ロジックはウィジェット内で稼働します。

30.3.5　シャドウクエリ

シャドウクエリの基本的なアイデアは、ページ上に多数のウィジェットがあると複数の選択を同時に行うことができるため、検索によってそれぞれのウィジェットが更新中に 1 つのウィジェットをクリックすると、基本的に検索がそこで終わってしまう（現在の選択内容である 1 つの結果しか表示されないため）ということに関するものです。

シャドウクエリのアイデアは、これらの選択されたウィジェットに「コンテキスト」または「シャドウ」を提供することです。特定のウィジェットを選択すると、選択したものは強調表示されますが、ページ上の他のウィジェットが更新される間、他のカテゴリは表示されたままです。次のウィジェットを選択すると、2 つのクエリが起動されます。その 1 つは選択されていないウィジェットと結果ペインを更新する一般的な検索であり、もう 1 つは選択されたウィジェットの「シャドウクエリ」です。このシャドウクエリは、選択されたウィジェットがその検索に影響しない点を除いて、メイン検索クエリのコピーとなります。これにより、ウィジェット内の選択内容の「周囲」のビューが表示されデータを深く確認できるので、ページ上のウィジェットの機能が大幅に拡張されます。

シャドウクエリは、データストリームを活用して動作します。データストリームは、2 つの箇所で設定されます。各ウィジェットのインスタンス化時（`datastream` は「ML.createWidget（Null Widget）メソッド」（539 ページ）で説明されている `createWidget` メソッドのパラメータです）と、コントローラでのシャドウクエリの作成時です。データストリームは、シャドウクエリをシャドウ処理中のウィジェットと一致させるために使用されます。

次に、データストリームについて説明します。

- クエリにデータストリームがない場合、選択されていないすべてのウィジェット（または、独自のデータストリームが設定されていないウィジェット）が更新されます。

- クエリにデータストリームがあり、これがウィジェットのデータストリームと一致する場合、そのウィジェットが更新されます（ウィジェットが選択されているかどうかにかかわりません）。
- クエリにデータストリームがあるが、ウィジェットのデータストリームと一致しない場合、そのウィジェットは何もしません。

  注： ウィジェットでは、選択によってシャドウクエリが起動されると、次回受信する更新がレンダリングされません。これは、その更新により、すでに最新の状態であることがウィジェットに通知されるためです。

30.4 HTML ページへの視覚化ウィジェットの追加

一部のコードは、グラフウィジェット（棒グラフ、横棒グラフ、折れ線グラフ、円グラフ）と地図ウィジェット（ポイント、ヒート）の両方で共通です。

30.4.1 <head> 要素内の一般的な JavaScript と CSS

<head> 要素内で、まず折れ線グラフ、棒グラフ、横棒グラフ、および円グラフ用に外部ライブラリの依存関係を追加します。

```
<!-- external library dependencies -->
<script src="lib/external/jquery-1.7.1.min.js"
        type="text/javascript"></script>
<script src="lib/external/highscharts.src.js"
        type="text/javascript"></script>
```

地図ウィジェットを使用する場合は、次の外部ライブラリも含める必要があります。

```
<!-- external web map files... -->
<script
src="http://maps.googleapis.com/maps/api/js?key=&amp;sensor=false&amp;
region=US&amp;libraries=drawing"
type="text/javascript">
</script>
<script src="lib/external/mxn-2.0.18/mxn.js?(googlev3)"
type="text/javascript"></script>
<script src="lib/external/heatmap.js" type="text/javascript"></script>
<script src="lib/external/heatmap-gmaps.js"
type="text/javascript"></script>
<script src="lib/external/markerclusterer_min.js"
type="text/javascript"></script>
```

次に、内部ウィジェットフレームワークライブラリの呼び出しを追加します。もちろん、ヒストグラムウィジェットを使用しない場合は、chart.js ライブラリを含める必要はありません。また、地図ウィジェットを使用しない場合は、map.js ライブラリを含める必要はありません。

```
<!-- internal widget framework library -->
<script src="lib/controller.js" type="text/javascript"></script>
<script src="lib/widget.js" type="text/javascript"></script>
<script src="/lib/viz/chart/chart.js" type-"text/javascript"></script>
<script src="/lib/viz/map/map.js" type-"text/javascript"></script>
```

最後に、ウィジェット要素用の CSS スタイルを定義できます。ライブラリにさまざまなウィジェットタイプ用のデフォルトの CSS スタイルがあるので、これはオプションです（/lib/viz/chart.css および /lib/viz/map.css を参照）。カスタムスタイルを定義する場合は、次のようなことを実行して、アプリケーション要件に応じて、ウィジェットの表示パラメータを定義します。もちろん、視覚化ウィジェットごとに別々の表示パラメータを設定し、そのため要素を別個に定義する必要がある場合、自分で作成する要素を widget と呼ぶ必要はありません。

```
<style type="text/css">
        .widget {
                  width: 500px;
                  display: block;
                  float: left;
                 }
</style>
```

30.4.2 <body> 要素内の一般的なコード

デフォルトのクラス「widget」であっても、ウィジェットの初期化時に ML.controller.init とその widgetClass オプションで指定したクラス名であっても、各ウィジェットは別個の 要素として定義する必要があります（「ML.controller メソッド」（537 ページ）を参照）。3 つの視覚化ウィジェット（2 つのグラフと 1 つの地図）がある後述する例では、次を使用します。

```
<div id="decadeContainer" class="widget"></div>
<div id="actorContainer" class="widget"></div>
<div id="locationContainer" class="widget"></div>
```

次に、次のことを行う <script> を定義します。

1. サーバーで認証し、クエリを MarkLogic サーバー上の REST エンドポイントに転送する PHP 検索プロキシを定義します。

   ```
   ML.controller.init({proxy: "proxy.php"});
   ```

   PHP プロキシの定義方法の詳細については、「プロキシをセットアップする」（536 ページ）を参照してください。

2. そのまま <script> 内で、各視覚化ウィジェットの設定変数を定義します。設定パラメータは、グラフウィジェットと地図ウィジェット（さらに、ポイントマップウィジェットとヒートマップウィジェット）で異なります。詳細な例については次の例を確認してください。すべての視覚化ウィジェットタイプで可能な設定パラメータの全リストについては後述します。ここで示す全体的な例の 3 つのウィジェットを表すプレースホルダは、次のようになります。

```
var yearConfig = {
...chart configuration parameters...
}

var actorConfig = {
...chart configuration parameters...
}

var mapConfig = {
...point and heat map configuration parameters...
}
```

3. 最後に、ウィジェットコントローラとウィジェット自体をアクティブ化し、<script> を閉じます。ML.controller.init は、前述の 1）で定義し、プロキシサーバーとこのアプリケーションの検索エンドポイントに関する情報を含んでいる変数（この場合は config）を引数として取ります。

   ML.chartWidget と ML.mapWidget はいずれも、ウィジェット要素名、ウィジェットタイプ、および設定パラメータの変数名から構成される同じシグネチャを持ちます。グラフウィジェットが作成されるか地図ウィジェットが作成されるかという点のみが異なります。

```
ML.controller.init(config); //connect to the var with our proxy value

//chartWidget can take 'line' or 'bar' or 'column' or 'pie'
//mapWidget just takes 'map'
ML.chartWidget('decadeContainer', 'line', yearConfig);
ML.chartWidget('actorContainer', 'bar', actorConfig);
ML.mapWidget('locationContainer', 'map', mapConfig);
```

30.4.3　表示を初期化する

ウィジェットの表示を初期化するには、コントローラの ML.controller.loadData メソッドを使用します。

```
ML.controller.loadData();
```

通常の </script> でスクリプトを終了します。

30.5 例：アプリケーションへのウィジェットの追加

このセクションでは、既存の検索アプリケーションに視覚化ウィジェットを追加する方法について説明します。視覚化ウィジェットを追加するために、次を実行する必要があります。

- 視覚化ウィジェットを作成するための一連のファイルを入手します。
- MarkLogic REST API のインスタンスを実装するアプリケーションサーバーを作成します。
- REST サーバーに接続するためのプロキシをセットアップします。このドキュメントでは、プロキシのホスティングに PHP サーバーを利用します。
- ウィジェットでファセットデータを利用できるようにするための、検索の options ノードを定義します。
- アプリケーションにあるウィジェットを設定および表示する HTML ページを作成します。

次のプロセスを開始するときは、MarkLogic データベースがセットアップされていることが前提になります。

1. Admin 画面または Information Studio のいずれかを使用して、MarkLogic サーバーに `restaurants` という名前のデータベースを作成します。

2. 『Information Studio Developer's Guide』の「Creating a REST API Instance」に説明されているように、データベースの REST サーバーを作成します。これにより、REST サーバーの名前で始まり、`-modules` が後に続くモジュールデータベースが自動的に作成されます。例えば、REST サーバーに `restaurants` という名前を付けた場合、`restaurants-modules` という名前のデータベースが自動的に作成されます。この例では、REST サーバーは、`myhost:5437` にあります。

3. Admin 画面を使用して、`restaurants` データベースに次の要素レンジインデックスを作成します。これらは、ウィジェットを使用して視覚化するファセットに対応します。

scalar type	localname
`string`	`Type`
`int`	`SqFtEst`

4. Query Console を使用して次のクエリを実行して、restaurants データベースにいくつかのドキュメントを読み込みます。

```
xquery version "1.0-ml";

xdmp:document-insert(
  "/1.xml",
  <restaurant>
    <id>1</id>
    <DBA> 123 Deli - Lee's </DBA>
    <Type> Restaurant, &lt; 500 sq' </Type>
    <SqFtEst> 500 </SqFtEst>
    <StreetAddress> 123 1st St </StreetAddress>
    <City> San Francisco </City>
    <State> CA </State>
    <FullAddress> 123 1st St, San Francisco CA </FullAddress>
    <Latitude>37.78946</Latitude>
    <Longitude>-122.39717</Longitude>
  </restaurant>),

xdmp:document-insert(
  "/2.xml",
  <restaurant>
    <id>2</id>
    <DBA> Dave's Dive </DBA>
    <Type> Restaurant, &lt; 700 sq' </Type>
    <SqFtEst> 700 </SqFtEst>
    <StreetAddress>2 1 Mission St </StreetAddress>
    <City> Bismarck </City>
    <State> ND </State>
    <FullAddress> 21 Mission St, Bismarck ND </FullAddress>
    <Latitude>46.48</Latitude>
    <Longitude>-100.47</Longitude>
  </restaurant>),

xdmp:document-insert(
  "/3.xml",
  <restaurant>
    <id>3</id>
    <DBA> Wayne's Steakhouse </DBA>
    <Type> Restaurant, &lt; 1000 sq' </Type>
    <SqFtEst> 1000 </SqFtEst>
    <StreetAddress> 444 Feick St </StreetAddress>
    <City> Albany </City>
    <State> NY </State>
    <FullAddress>444 Feick St, Albany NY</FullAddress>
    <Latitude>42.40</Latitude>
    <Longitude>-73.45</Longitude>
  </restaurant>),
```

```
xdmp:document-insert(
  "/4.xml",
  <restaurant>
    <id>4</id>
    <DBA> Megha </DBA>
    <Type> Restaurant, &lt; 1000 sq' </Type>
    <SqFtEst> 1000 </SqFtEst>
    <StreetAddress> 4531 Front St </StreetAddress>
    <City> Carlsbad </City>
    <State> NM </State>
    <FullAddress> 4531 Front St, Carlsbad NM </FullAddress>
    <Latitude>32.26</Latitude>
    <Longitude>-104.15</Longitude>
  </restaurant>),

xdmp:document-insert(
  "/5.xml",
  <restaurant>
    <id>5</id>
    <DBA> Gordon's Grubhaus </DBA>
    <Type> Restaurant, &lt; 300 sq' </Type>
    <SqFtEst> 300 </SqFtEst>
    <StreetAddress> 3421 Capital Wy </StreetAddress>
    <City> El Paso </City>
    <State> TX </State>
    <FullAddress> 421 Capital Wy, El Paso TX </FullAddress>
    <Latitude>31.46</Latitude>
    <Longitude>-106.28</Longitude>
  </restaurant>)
```

5. Query Console を使用して、`restaurants-modules` データベースに、次の検索の options ノードを読み込みます。これにより、視覚化に必要なファセットが利用できるようになります。

   注： このクエリを実行するには、`rest-admin` 権限を持つユーザーとして MarkLogic サーバーにログインしている必要があります。

```
xdmp:http-put("http://myhost:5437/v1/config/query/all",
<options xmlns="xdmp:http">
  <authentication method="digest">
    <username>admin</username>
    <password>admin</password>
  </authentication>
  <headers>
    <content-type>application/xml</content-type>
    <accept>application/xml</accept>
  </headers>
</options>,
text { xdmp:quote(
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="Type">
    <range type="xs:string" facet="true">
```

```
        <element ns="" name="Type"/>
        <facet-option>limit=10</facet-option>
        <facet-option>frequency-order</facet-option>
        <facet-option>descending</facet-option>
      </range>
      </constraint>
      <constraint name="SqFtEst">
        <range type="xs:int" facet="true">
          <element ns="" name="SqFtEst"/>
          <facet-option>limit=5</facet-option>
          <facet-option>frequency-order</facet-option>
          <facet-option>descending</facet-option>
        </range>
      </constraint>
  </options>
  ) }
  )
```

6. 次の URL を入力します。ここで、`<server>:<rest-port>` は使用する REST サーバーを特定します。すべての制約がセットアップされている options ノードファイルが表示されます。

```
http://<server>:<rest-port>/v1/config/query/all
```

7. PHP がオンにされている Apache サーバーのパブリックディレクトリに移動します。この例では、`/var/www/html` です。

8. 視覚化ウィジェットのファイルを次からダウンロードします。

```
http://localhost:8000/appbuilder/customappfiles.xqy
```

9. `customappfiles.xqy` ファイルを解凍し、5 つのサブフォルダ（constraint、css、images、lib、および skins）を PHP がオンにされている Apache サーバーのパブリックディレクトリにコピーします。

   注： この例では、Application Builder によって作成されたアプリケーション内にあるような「application」ディレクトリはありません。

10. PHP がオンにされている Apache サーバーのパブリックディレクトリに、次のコンテンツを含む `proxy.php` という名前の PHP プロキシファイルを作成します。`CURLOPT_URL` オプションが `restaurants` サーバー（この例では `myhost:5437`）を指していること、および `CURLOPT_USERPWD` オプションに MarkLogic サーバー用のログイン資格情報が含まれていることを確認します。

```
<?PHP

$queryString = $_SERVER['QUERY_STRING'];
parse_str($queryString, $vars);
```

```
unset($vars['proxyPath']);
$queryString = http_build_query($vars);

$options = array(
CURLOPT_URL => 'http://myhost:5437' . $_GET["proxyPath"] . '?' .
$queryString, // location of the rest server. Be sure to preserve the
query string unaltered

CURLOPT_HTTPAUTH => CURLAUTH_DIGEST, // it is very important to set the
authentication type to digest

CURLOPT_USERPWD => 'admin:admin', // this is your username and password
for the server. be sure your role has read permissions on the database

CURLOPT_HTTPHEADER => array('Content-type: application/json'), // the
server won't respond properly unless the content-type is
application/json
);

// Only set POST options for POSTs, not GETs
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
$options[CURLOPT_POST] =  1; // queries are sent in the post body, with
options like pagination in the query string

$options[CURLOPT_POSTFIELDS] = file_get_contents('php://input'); // we
are not sending a traditional form map through the post, so avoid using
the standard $_POST
}

$ch = curl_init(); // create curl
curl_setopt_array($ch, $options); // set curl options
curl_exec($ch); // execute curl

?>
```

11. PHP がオンにされている Apache サーバーのパブリックディレクトリに、次のコンテンツを含む HTML ファイルを作成します。

```
<html>
<head>
  <title>Custom App</title>

  <!-- external library dependencies -->
  <script src="lib/external/jquery-1.7.1.min.js"
    type="text/javascript"></script>
  <script src="lib/external/highcharts.src.js"
    type="text/javascript"></script>

  <!-- internal widget framework library -->
  <script src="lib/controller.js" type="text/javascript"></script>
  <script src="lib/widget.js" type="text/javascript"></script>
```

```
  <!-- visualization framework libraries -->
  <script src="lib/viz/chart/chart.js" type="text/javascript">
  </script>
  <link media="screen, print" href="lib/viz/chart/chart.css"
    rel="stylesheet" type="text/css">

  <style type="text/css">
    .widget {
       width: 500px; display: block; float: left;
    }
  </style>
</head>

<body>
  <h2>MarkLogic Custom Application</h2>
  <div id="typeContainer" class="widget"></div>
  <div id="sqftContainer" class="widget"></div>
  <script type="text/javascript">

<!-- If you have a proxy, identify it here. -->
  var config = { proxy: "proxy.php" };

  var typeConfig = { constraint: 'Type',
                     constraintType: 'range-unbucketed',
                     dataType: 'xs:string',
                     title: 'Type',
                     dataLabel: 'Type' };

  var sqftConfig = { constraint: 'SqFtEst',
                     constraintType: 'range-unbucketed',
                     dataType: 'xs:int',
                     title: 'Square Feet',
                     dataLabel: 'Square Feet' };

  ML.controller.init(config);
  ML.chartWidget('typeContainer', 'column', typeConfig);
  ML.chartWidget('sqftContainer', 'bar', sqftConfig);

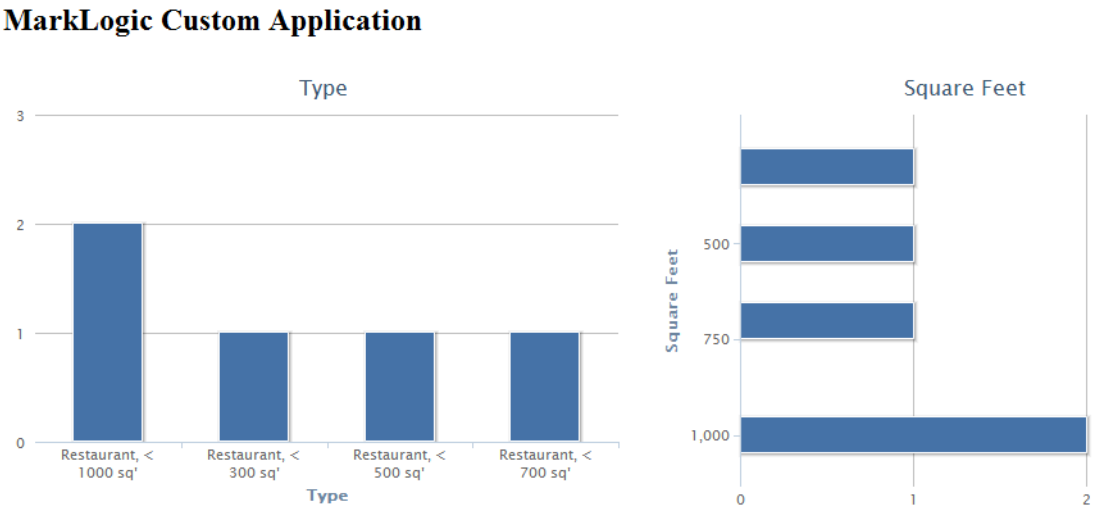
<!-- This will trigger the intial empty search -->
  ML.controller.loadData();
</script>
</body>
</html>
```

12. PHP がオンにされている Apache サーバーのパブリックディレクトリ内のすべてのファイルが、パブリックにアクセスできることを確認します。

13. ブラウザで、PHP がオンにされている Apache サーバーのホストの URL を入力します。例えば次のようになります。

```
http://myhost/index.html
```

14.　次のようなページが表示されます。



30.6　視覚化ウィジェットの制限

ほとんどまたはすべてのウィジェットタイプで共通する制限がいくつかあります。制限は次のとおりです。

- 地図以外のウィジェットは、それに与えられたコンテナに合わせたサイズになり、スクロールしません。最も一般的な地図プログラムとまったく同様に、地図ウィジェットは地図そのものの範囲でスクロールできます。ズームインまたはズームアウト、左右上下へのパン（移動）などを行うことができます。これは地図のコンテナに影響せず、地図そのもののビューにのみ影響します。地図は、ズームまたはパンを無効にするように設定できます。パンは南北方向では制限があり、北極または南極を超えることはできません。
- 地図以外のウィジェットでは、多次元の視覚化はサポートされません（つまり、1 つのウィジェットにつき、1 つのファセットです）。
- 地図の結果は、検索結果自体ではなく、geo ファセットから取得されます。
- 地図では、パンおよびズームを行うと、地図によってマーカーおよびマーカークラスタを「描く / 色づけする」ための新しいクエリが生成されます。このアプローチにした理由は、大きなデータセットを含む可能性がある地図表示に効率的なメカニズムとなるためです。このため、データクエリはすべてのデータポイントを返すのではなく、段階的に返します。
- ツールヒントの動作は、グラフウィジェットによって定義されているため、Application Builder でカスタマイズすることはできません。

30.7 プロキシをセットアップする

カスタムアプリケーションを実行するためには、一般に、適切なプロキシを通じてクエリを実行して、MarkLogic REST サーバーインスタンスで認証する必要があります。

コントローラ設定には、プロキシの基本パスとして設定される必要があるプロキシ変数があります。コントローラにより、プロキシが読み取る必要がある URL パラメータ「proxyPath」が追加されます。proxyPath は、クエリを代理の実行先にする必要がある MarkLogic REST サーバーインスタンス上の絶対パスを指します。プロキシは、POST で呼び出されると、POST データを MarkLogic REST サーバーインスタンスに変更せずに伝えます。また、proxyPath パラメータを除くすべての url パラメータも渡す必要があります。GET リクエストの場合、url パラメータのみが必要です。レスポンスは、変更せずにそのまま渡す必要があります。

PHP 導入の例については、「例：アプリケーションへのウィジェットの追加」（529 ページ）で説明しています。REST サーバーの場所、ユーザー名、パスワード値のようないくつかのオプションは、ユーザーの環境によって異なる場合があることに注意してください。自分の環境の値を使用するようにコードを編集します。このコードを、PHP サーバー上でパブリックにアクセス可能なファイル（例えば、`proxy.php`）に入れます。PHP サーバーは、アプリケーションの検索エンドポイントとして機能します。

正しく機能するプロキシがあることを確認した後、`index.html` ページの `ML.controller.init` メソッドに `proxy` オプションを追加して、PHP プロキシを特定します。例えば次のようになります。

```
ML.controller.init({proxy: "proxy.php"});
```

30.8 ウィジェット API リファレンス

このセクションは、各視覚化ウィジェットの設定オプションおよび関連メソッドのリファレンスです。棒グラフ、横棒グラフ、折れ線グラフ、および円グラフはすべて、同じ設定オプションを持ちます。円グラフを他のグラフと同じに扱うことは奇妙に感じられるかもしれませんが、円グラフの「スライス」が他のグラフでの棒 / 横棒 / データポイントと同じ目的を果たしていると考えると、理解できるでしょう。

このセクションで取り上げるトピックは次のとおりです。

- ML.controller メソッド
- ML.createWidget（Null Widget）メソッド
- 一般的なウィジェットのメソッド
- ML.chartWidget メソッド
- ML.mapWidget メソッド

- ウィジェットのイベント
- ウィジェットのクエリ構造
- ウィジェットのクエリ更新構造
- 検索結果の形式
- ファセット構造
- ウィジェットの内部イベント

30.8.1 ML.controller メソッド

メソッド	説明
`init`	上の表の、名前と値のペアによる設定オプションから構成される、オプションの config オブジェクトを取ります。すべての値はオプションであり、デフォルトでは、Application Builder が構築したアプリケーションで使用されている値になります。
`getData(query)`	これは、データの初回の読み込みや、何らかの理由からの検索リセットの要求のような操作のために、クエリで呼び出しを行う直接的なメソッドです。クエリオブジェクトは必須ですが、空にすることもできます。例えば次のようになります。 `ML.controller.getData({});`
`loadData`	ページ上での初回のデータ読み込みを実行します。ブックマーク機能がオンにされている場合、ブックマークされたクエリの url を読み取ります。loadData() はすべてのウィジェットを初期化した後に呼び出してください。

`ML.controller.init` メソッドは、次のシグネチャを持ちます。

```
ML.controller.init ({option1: value1, option2: value2...});
```

有効なオプションは次のとおりです。

オプション	説明
`appWrapper`	アプリケーションをページのセクション内にラップできるので、アプリケーションはその他の部分に干渉しません。これは、ウィジェットアプリケーションの最上位の要素です。デフォルトで body タグになり、ページ全体が対象となりますが、任意の jQuery セレクタを渡すことで、同一のページ内の複数の別個のウィジェットアプリケーションに対応できます(例えば、複数の異なるデータベースからのデータを表示する場合)。また、既存のアプリケーションへの干渉を防止するための安全策として渡すこともできます。イベントと相互作用は、appWrapper 内に完全に入れられます。 デフォルト：`body`
`version`	MarkLogic サーバー REST API のバージョン。 デフォルト：`v1`
`optionsNode`	クエリ実行で基準とする制約を含んでいる options ノード。デフォルトで「all」（Application Builder によって生成された options ノードの名前）になります。 デフォルト：`all`
`startPage`	結果のページネーションの、デフォルトの開始ページを設定します。これは、視覚化ウィジェットには影響しません。検索結果のみに影響します。 デフォルト：`1`
`useShadows`	アプリケーションにおけるシャドウクエリをグローバルにオンまたはオフにします。 デフォルト：`true`

オプション	説明
`widgetClass`	これは、アプリケーション内でウィジェットを特定する jQuery セレクタです。クラスが `widget` であるあらゆる要素は、デフォルトのセットアップで newData イベントを自動的に受信します。 デフォルト：`.widget`
`proxy`	プロキシを使用する場合は、ここにプロキシへのパスを指定します。プロキシのセットアップ手順については、プロキシのセクションを参照してください。 デフォルト：`false`
`enableBookmarking`	ブックマーク機能のサポートをオンにします。 デフォルト：`true`
`bookmarkingDelimiters`	URL で現在のクエリをエンコーディングするときに使用する区切り文字。 デフォルト：`['*_*', '*__*']`
`pageSize`	クエリで返される結果の個数（ウィジェットで表示される制約には影響しません）。 デフォルト：`10`

30.8.2　ML.createWidget（Null Widget）メソッド

「null widget」は、アプリケーション内のウィジェットのコアクラスです。null widget の役割は、一般的な通信動作と受信データを扱うことです。グラフや地図のような視覚化機能、検索ボックスのようなコントロール、結果ペインのような表示はすべて、基本的な null widget を拡張したものです。

```
ML.createWidget(container, renderCB, datastream, constraintType)
```

パラメータ	説明
`container`	必須 ウィジェットのコンテナ要素の ID。div にすることをお勧めします。

パラメータ	説明
renderCB	オプション ウィジェットが「newData」イベントを受信したときに実行するコールバック関数。関数には、その結果をレンダリングする目的でサーバーのレスポンスが渡されます。
datastream	オプション * ウィジェットがシャドウクエリに応答することを可能にする識別子。一般に、ウィジェットの制約名に設定する必要があります。
constraintType	オプション * ウィジェットでファセットデータを使用する場合、constraintTypeを設定する必要があります。その制約については、サーバーのインデックスに対する制約のタイプと一致する必要があります。 「range」、「constraint」、「geo」のいずれかにする必要があります。構造化クエリの生成方法を決定します。地図データのタイプは常に geo です。他のデータは、サーバー上でレンジインデックスであるかコレクションであるかに応じて、レンジまたは制約にすることができます。

* シャドウクエリを利用する場合には必須

30.8.3　一般的なウィジェットのメソッド

メソッド	説明
updateQuery(queryUpdate)	引数として queryUpdate オブジェクトを取り（前述のコントローラオブジェクトを参照）、ウィジェットに updateQuery メソッドをトリガーさせます。検索ボックスなど、クエリを更新する必要のある、選択内容のないウィジェットのための抽象メソッド。
getSelection()	ウィジェットで現在選択されている値が存在する場合はそれを返し、それ以外の場合は、null を返します。

30.8.4 ML.chartWidget メソッド

折れ線、棒、横棒、および円のウィジェットの作成に使用するメソッドは、次のとおりです。

```
ML.chartWidget(containerID, type, config);
```

パラメータ	説明
containerID	ウィジェットのコンテナ要素の ID。div にすることをお勧めします。(必須)
type	グラフのタイプ。これは、次のいずれかにすることができます。 • line：折れ線グラフ • bar：棒グラフ • column：横棒グラフ • pie：円グラフ
config	次の表に示すオプションから構成される設定オブジェクト。

ML.chartWidget の設定オプションは次のとおりです。

オプション	説明
constraint	グラフでデータをレンダリングする制約ファセットの名前。無効な場合、グラフはレンダリングされません。デフォルトでは "" になります。
constraintType	グラフでデータをレンダリングする制約ファセットのタイプ。クエリの構築に使用されます。使用可能な値：range、constraint、geo。値には、検索を実行するためにクエリを実行する制約のタイプを指定する必要があります。無効な場合、クエリが実行されないか、サーバーでエラーが発生します。デフォルトでは "" になります（実際のデフォルト値はありません)。
title	ウィジェットに表示されるタイトルの文字列。デフォルトでは "" になります。
subtitle	ウィジェットに表示されるサブタイトルの文字列。デフォルトでは "" になります。

オプション	説明
dataLabel	グラフに表示される値タイプのラベルの文字列。デフォルトでは "" になります。
dataType	グラフに表示されるデータ型の文字列。データベースにある制約のデータ型を使用してください。レンジの設定を無効にするには、dataType を「string」に設定します。「string」、「datetime」、「int」などを取ります。デフォルトで「string」になります。
includeOthers	true である場合、結果が制限されている際に、グラフ「その他」というカテゴリができます。例えば、オスカーにノミネーションされた回数で上位 10 人の俳優を表示し、他の俳優の合計回数についてはその他というカテゴリを示す場合などです。true または false を取ります。デフォルトで false になります。
othersLabel	「その他」の結果を表すのに表示されるラベルの文字列。includeOthers を参照してください。デフォルトでは「Other」になります。
hideXAxisValues	true である場合は、X 軸のラベルは非表示になり、それ以外の場合は、グラフの下に表示されます。棒が密集して、ラベルを読むことができなくなる場合に役立ちます。注：円グラフには X 軸がなく、また横棒グラフの X 軸は水平ではなく垂直の軸になります。true または false を取ります。デフォルトで false になります。

30.8.5 ML.mapWidget メソッド

地図ウィジェットの作成に使用するメソッドは次のとおりです。

```
ML.mapWidget('locationContainer', 'map', mapConfig);
```

次に、地図ウィジェットの設定オプションを示します。これらは、次のような設定変数を定義するために使用します。

```
var mapConfig = {
       constraintType: 'geo',
       dataStream: 'results',
       statusContainerId: 'debug',
       width: 698,
       height: 512,
       ...
};
```

ポイントマップの設定オプションは、次のとおりです。

オプション	説明
geoConstraint	検索結果に対する地理的制約を定義します。地図に関連付けられているデータベースからの geo 制約の値を持ちます。デフォルトでは "" になります。
mapWidth	ピクセル単位の地図（とそのコンテナ）の幅。デフォルトでは 485 になります。
mapHeight	ピクセル単位の地図（とそのコンテナ）の高さ。デフォルトでは 300 になります。
mapProvider	地図のソース。現時点では、Google マップ用のデフォルト値のみがサポートされています。デフォルトでは「googlev3」になります。
zoomControlType	ズームコントロールを表示します。「small」または「large」のいずれかを取ります。デフォルトでは「small」になります。
showMapTypes	利用可能な地図のタイプを表示します（衛星、道路など）。「true」または「false」のいずれかを取ります。デフォルトでは「false」になります。
showSelectControls	領域を選択するための地図ウィジェット検索選択コントロールをオーバーレイとして表示します。「true」または「false」のいずれかを取ります。デフォルトで「false」になります。
showMapControls	地図ウィジェットコントロール（ポイントマップからヒートマップへの切り替え、およびその逆を行います）のオーバーレイを表示します。「true」または「false」のいずれかを取ります。デフォルトでは「false」になります。
minZoomLevel	地図のズーム値の下限を定義します。デフォルトでは 2 になります。
maxZoomLevel	地図のズーム値の上限を定義します。デフォルトでは 14 になります。
constrainPan	地図のパン（移動）機能が制約されるかどうかを決定します（つまり、true の場合、地図の視覚的領域が固定されます）。「true」または「false」を取ります。デフォルトでは「true」になります。

オプション	説明
constrainZoom	地図のズーム機能が制約されるかどうかを決定します（つまり、true の場合、ズームインおよびズームアウトできません）。「true」または「false」を取ります。デフォルトでは「true」になります。
autoCenterZoom	true の場合、画面上の任意のオブジェクト（マーカー、多角形）が自動的に中心になります。「true」または「false」を指定します。デフォルトでは「true」になります。
showMarkersOnLoad	true の場合、地図が読まれたときに、検索の一致を示す地図ポインタが表示されます。「true」または「false」を取ります。デフォルトでは「true」になります。
showHeatmap	地図がヒートマップ（「true」）かポイントマップ（「false」）かを決定します。「true」または「false」を取ります。デフォルトでは「false」になります。
defaultMarkerIcon	地図上に検索結果のヒットを表示するために使用する画像。画像ファイルを取ります。デフォルトで「/images/map_red_shadow.png」になります。

前述のポイントマップのオプションに加えて、ヒートマップには次のオプションもあります。

オプション	説明
heatMapRadius	大きな値は、各ポイントにより、地図上で「ホットな」領域が大きくなることを意味します。整数を取ります。デフォルトでは 15 になります。
heatMapOpacity	エンティティの不透明度を決定する 0 ～ 100（不透明度が最も低い～最も高い）の整数を取ります。デフォルトでは 90 になります。
searchColor	検索結果での線（多角形、円、正方形など）の色を定義します。HTML の 16 進数のカラー値を取ります。デフォルトでは「#FF0000」（赤）になります。
searchFillColor	検索結果（つまり、検索の多角形領域）を塗りつぶす色を定義します。HTML の 16 進数のカラー値を取ります。デフォルトで "" になります。
searchOpacity	検索結果の円の色での不透明度のレベルを定義します。0 以上 1 以下の値を指定します。ここで、0 は最も不透明度が低く、1 は最も不透明度が高くなります。デフォルトでは 1 になります。

オプション	説明
searchFillOpacity	検索の塗りつぶし（つまり、検索の多角形領域）の不透明度レベルを定義します。0 以上 1 以下の値を指定します。ここで、0 は最も不透明度が低く、1 は最も不透明度が高くなります。デフォルトでは 0 になります。

30.8.6　ウィジェットのイベント

イベント	説明
updateQuery	このイベントは、選択を行うウィジェットによってポストされます。コントローラは、appWrapper 要素の DOM ツリーをバブルアップするこのイベントを listen します。アタッチされる JSON オブジェクトのデータ構造については、次を参照してください。

newData	このイベントは、アプリケーションのウィジェットすべてに、次の形式のアタッチされたデータオブジェクトとともにポストされます。
	
	{
	data: <server results>,
	query: <query object for search>
	}

newQuery	現在の検索クエリを更新する代わりに、現在のクエリを指定されたクエリにリセットします。
getBounds	REST サーバー上の値エンドポイントから指定された制約について上方の境界と下限の境界を取得します。制約はイベントにアタッチされます

newBounds	境界オブジェクトの制約を使用して視覚化ウィジェットの xAxis スケールを設定するウィジェットに、境界オブジェクトをパブリッシュします。これは、次の形式を取ります。
	
	{
	min: <min_value>
	max: <max_value>
	constraint: <constraint>
	}

30.8.7　ウィジェットのクエリ構造

クエリ構造（空でもかまいせん）

フィールド	説明
text	サーバーに改変なしで直接渡すテキストクエリ。以前のビルドアプリケーションバージョンのクエリシンタックスをすべてサポートしています（オプション）
facets	ページ上のウィジェットからの選択内容を示すファセットオブジェクトの配列。後の「ファセット構造」を参照してください（オプション）
page	検索結果のページネーションに使用するページ番号（オプション）
datastream	クエリをレンダリングするウィジェットを特定するために使用します（オプション）

30.8.8　ウィジェットのクエリ更新構造

フィールド	説明
facet	更新するファセット / 制約の名前
value	検索するファセット内の選択する値の名前
text	サーバーに改変なしで直接渡すテキストクエリ。以前のビルドアプリケーションバージョンのクエリシンタックスをすべてサポートしています
constraintType	「range」、「constraint」、「geo」 のいずれかにする必要があります。構造化クエリの生成方法を決定します。地図データのタイプは常に geo です。他のデータは、サーバー上でレンジインデックスであるかコレクションであるかに応じて、レンジまたは制約にすることができます。

30.8.9　検索結果の形式

検索結果の形式（REST API）

フィールド	説明
total	検索によって返された結果の総数
start	最初に返された結果の結果番号
page-length	このクエリで返された結果の個数
results	結果オブジェクトの配列

facets	検索に対するすべてのファセットとファセット結果のリスト。
	
	`<facet-name>: {`
	`type: "xs:<data-type>",`
	`facetValues: [`
	`{name: <record name>, count: <int>},`
	`…`
	`]`
	`}`

report	クエリに関する情報を含む文字列レポート。
metrics	サーバーからの検索のメトリック

30.8.10　ファセット構造

フィールド	説明
value	選択され、検索の更新に使用されるファセット値の「名前」。例えば、Oscars アプリケーションの俳優ファセットでは、これは「Humphrey Bogart」などです。
constraintType	「range」、「constraint」、「geo」のいずれかにする必要があります。構造化クエリの生成方法を決定します。地図データのタイプは常に geo です。他のデータは、サーバー上でレンジインデックスであるかコレクションであるかに応じて、レンジまたは制約にすることができます。

フィールド	説明
geo	geo プロパティには、次のいずれかが入ります。 • `geo.poly`：lat/lng ポイントの配列 • `geo.box`：値が lat/lng ポイントである 4 つのプロパティ（北、南、東、西）。 • `geo.circle`：半径、および緯度と経度を含むポイントオブジェクト。

30.8.11 ウィジェットの内部イベント

イベント	説明

`selection`	選択イベントは、ウィジェットコンテナによって捕捉され、queryUpdate イベントの生成に使用されます（コントローラのイベントを参照してください）。選択イベントは、次の形式のアタッチされた選択オブジェクトを持ちます。
	`{ facet:`
	`  <facet name>,`
	`  value: <selected value>`
	`}`
	選択解除するには、次のように、値プロパティがないファセットを渡すだけです。
	`{facet: "decade"}`

`page`	ページイベントは、検索結果のページネーションの調整に使用されます。イベントとともに渡されるデータには、表示するページ番号を指定するページプロパティがあります。