MarkLogic Server

Semantics Developer's Guide

MarkLogic 9
May, 2017

Last Revised: 9.0-8, December 2018

Copyright © 2019 MarkLogic Corporation. All rights reserved.

Table of Contents

Semantics Developer's Guide

1.0 Introduction to Semantics in MarkLogic 11
- 1.1 Terminology 12
- 1.2 Linked Open Data 13
- 1.3 RDF Implementation in MarkLogic 14
  - 1.3.1 Using RDF in MarkLogic 15
    - 1.3.1.1 Storing RDF Triples in MarkLogic 17
    - 1.3.1.2 Querying Triples 18
  - 1.3.2 RDF Data Model 20
  - 1.3.3 Blank Node Identifiers 21
  - 1.3.4 RDF Datatypes 21
  - 1.3.5 IRIs and Prefixes 22
    - 1.3.5.1 IRIs 22
    - 1.3.5.2 Prefixes 23
  - 1.3.6 RDF Vocabulary 23
- 1.4 Example Datasets 24

2.0 Getting Started with Semantics in MarkLogic 27
- 2.1 Setting up MarkLogic Server 27
  - 2.1.1 Configuring the Database to Work with Triples 27
  - 2.1.2 Setting Up Additional Servers 28
- 2.2 Loading Triples 28
  - 2.2.1 Downloading the Dataset 29
  - 2.2.2 Importing Triples with mlcp 29
  - 2.2.3 Verifying the Import 30
- 2.3 Querying Triples 32
  - 2.3.1 Querying with Native SPARQL 32
  - 2.3.2 Querying with the sem:sparql Functions 34

3.0 Loading Semantic Triples 37
- 3.1 Loading Embedded RDF Triples 37
- 3.2 Loading Triples 37
  - 3.2.1 Supported RDF Triple Formats 38
  - 3.2.2 Example RDF Formats 39
    - 3.2.2.1 RDF/XML 39
    - 3.2.2.2 Turtle 40
    - 3.2.2.3 RDF/JSON 40
    - 3.2.2.4 N3 41
    - 3.2.2.5 N-Triples 41

3.2.2.6 N-Quads 43
3.2.2.7 TriG 44
3.2.3 Loading Triples with mlcp 44
3.2.3.1 Preparation 45
3.2.3.2 Import Command Syntax 46
3.2.3.3 Loading Triples and Quads 46
3.2.3.4 Import Options 47
3.2.3.5 Specifying Collections and a Directory 49
3.2.4 Loading Triples with XQuery 51
3.2.4.1 sem:rdf-insert 52
3.2.4.2 sem:rdf-load 53
3.2.4.3 sem:rdf-get 53
3.2.5 Loading Triples with JavaScript 54
3.2.5.1 sem.rdfInsert 55
3.2.5.2 sem.rdfLoad 56
3.2.5.3 sem.rdfGet 56
3.2.6 Loading Triples Using the REST API 56
3.2.6.1 Preparation 57
3.2.6.2 Addressing the Graph Store 57
3.2.6.3 Specifying Parameters 58
3.2.6.4 Supported Verbs 58
3.2.6.5 Supported Media Formats 59
3.2.6.6 Loading Triples 59
3.2.6.7 Response Errors 60
3.2.7 Loading Triples Using the Java API 61
3.2.8 Loading Triples Using the Node.js API 61

4.0 Triple Index Overview 63
4.1 Understanding the Triple Index and How It's Used 63
4.1.1 Triple Data and Value Caches 63
4.1.1.1 Triple Cache and Triple Value Cache 64
4.1.2 Triple Values and Type Information 64
4.1.3 Triple Positions 64
4.1.4 Index Files 65
4.1.5 Permutations 66
4.2 Enabling the Triple Index 66
4.2.1 Using the Database Configuration Pages 66
4.2.2 Using the Admin API 68
4.3 Other Considerations 69
4.3.1 Sizing Caches 69
4.3.2 Unused Values and Types 70
4.3.3 Scaling and Monitoring 71

5.0 Unmanaged Triples 73
5.1 Uses for Triples in XML Documents 76

5.1.1 Context from the Document 76
5.1.2 Combination Queries 77
5.1.3 Security 78
5.2 Bitemporal Triples 78

6.0 Semantic Queries 81
6.1 Querying Triples with SPARQL 82
6.1.1 Types of SPARQL Queries 82
6.1.2 Executing a SPARQL Query in Query Console 83
6.1.3 Specifying Query Result Options 83
6.1.3.1 Auto vs. Raw Format 83
6.1.3.2 Selecting Results Rendering 86
6.1.4 Constructing a SPARQL Query 87
6.1.5 Prefix Declaration 87
6.1.6 Query Pattern 88
6.1.7 Target RDF Graph 91
6.1.7.1 The FROM Keyword 93
6.1.7.2 The FROM NAMED Keywords 94
6.1.7.3 The GRAPH Keyword 95
6.1.8 Result Clauses 95
6.1.8.1 SELECT Queries 96
6.1.8.2 CONSTRUCT Queries 96
6.1.8.3 DESCRIBE Queries 98
6.1.8.4 ASK Queries 99
6.1.9 Query Clauses 99
6.1.9.1 The OPTIONAL Keyword 100
6.1.9.2 The UNION Keyword 100
6.1.9.3 The FILTER Keyword 102
6.1.9.4 Using Built-in Functions in a SPARQL Query 104
6.1.9.5 Comparison Operators 105
6.1.10 Negation in Filter Expressions 105
6.1.10.1 EXISTS 106
6.1.10.2 NOT EXISTS 106
6.1.10.3 MINUS 107
6.1.10.4 Differences Between NOT EXISTS and MINUS 108
6.1.10.5 Combination Queries with Negation 110
6.1.10.6 BIND Keyword 111
6.1.10.7 Values Sections 111
6.1.11 Solution Modifiers 112
6.1.11.1 The DISTINCT Keyword 112
6.1.11.2 The LIMIT Keyword 113
6.1.11.3 ORDER BY Keyword 113
6.1.11.4 The OFFSET Keyword 115
6.1.11.5 Subqueries 115
6.1.11.6 Projected Expressions 116
6.1.12 De-Duplication of SPARQL Results 117

6.1.13 Property Path Expressions 118
6.1.13.1 Enumerated Property Paths 118
6.1.13.2 Unenumerated Property Paths 120
6.1.13.3 Inference 123
6.1.14 SPARQL Aggregates 124
6.1.15 Using the Results of sem:sparql 127
6.1.16 SPARQL Resources 127
6.2 Querying Triples with XQuery or JavaScript 128
6.2.1 Preparing to Run the Examples 129
6.2.2 Using Semantic Functions to Query 130
6.2.2.1 sem:sparql 131
6.2.2.2 sem:sparql-values 133
6.2.2.3 sem:store 134
6.2.2.4 Querying Triples in Memory 134
6.2.3 Using Bindings for Variables 135
6.2.4 Viewing Results as XML and RDF 137
6.2.5 Working with CURIEs 139
6.2.6 Using Semantics with cts Searches 142
6.2.6.1 cts:triples 142
6.2.6.2 cts:triple-range-query 143
6.2.6.3 cts:search 143
6.2.6.4 cts:contains 144
6.3 Querying Triples with the Optic API 145
6.4 Serialization 145
6.4.1 Setting the Output Method 146
6.5 Security 146

7.0 Inference 147
7.1 Automatic Inference 147
7.1.1 Ontologies 148
7.1.2 Rulesets 149
7.1.2.1 Pre-Defined Rulesets 150
7.1.2.2 Specifying Rulesets for Queries 151
7.1.2.3 Using the Admin UI to Specify a Default Ruleset for a Database 153
7.1.2.4 Overriding the Default Ruleset 155
7.1.2.5 Creating a New Ruleset 156
7.1.2.6 Ruleset Grammar 157
7.1.2.7 Example Rulesets 158
7.1.3 Memory Available for Inference 160
7.1.4 A More Complex Use Case 161
7.2 Other Ways to Achieve Inference 161
7.2.1 Using Paths 162
7.2.2 Materialization 163
7.3 Performance Considerations 163
7.3.1 Partial Materialization 163

7.4 Using Inference with the REST API 163
7.5 Summary of APIs Used for Inference 165
7.5.1 Semantic APIs 165
7.5.2 Database Ruleset APIs 166
7.5.3 Management APIs 166

8.0 SPARQL Update 169
8.1 Using SPARQL Update 170
8.2 Graph Operations with SPARQL Update 170
8.2.1 CREATE 171
8.2.2 DROP 172
8.2.3 COPY 172
8.2.4 MOVE 173
8.2.5 ADD 174
8.3 Graph-Level Security 175
8.4 Data Operations with SPARQL Update 177
8.4.1 INSERT DATA 178
8.4.2 DELETE DATA 180
8.4.3 DELETE..INSERT WHERE 181
8.4.4 DELETE WHERE 182
8.4.5 INSERT WHERE 182
8.4.6 CLEAR 183
8.5 Bindings for Variables 184
8.6 Using SPARQL Update with Query Console 185
8.7 Using SPARQL Update with XQuery or Server-Side JavaScript 186
8.8 Using SPARQL Update with REST 187

9.0 Using Semantics with the REST Client API 189
9.1 Assumptions 191
9.2 Specifying Parameters 191
9.2.1 SPARQL Query Parameters 191
9.2.2 SPARQL Update Parameters 193
9.3 Supported Operations for the REST Client API 194
9.4 Serialization 196
9.4.1 Unsupported Serialization 197
9.5 Examples Using curl and REST 197
9.6 Response Output Formats 199
9.6.1 SPARQL Query Types and Output Formats 200
9.6.2 Example: Returning Results as XML 201
9.6.3 Example: Returning Results as JSON 202
9.6.4 Example: Returning Results as HTML 203
9.6.5 Example: Returning Results as CSV 204
9.6.6 Example: Returning Results as N-triples 205
9.6.7 Example: Returning a Boolean as XML or JSON 206
9.7 SPARQL Query with the REST Client API 207

9.7.1 SPARQL Queries in a POST Request 207
9.7.2 SPARQL Queries in a GET Request 210
9.8 SPARQL Update with the REST Client API 211
9.8.1 SPARQL Update in a POST Request 212
9.8.2 SPARQL Update via POST with URL-encoded Parameters 214
9.9 Listing Graph Names with the REST Client API 214
9.10 Exploring Triples with the REST Client API 215
9.11 Managing Graph Permissions 217
9.11.1 Default Permissions and Required Privileges 218
9.11.2 Setting Permissions as Part of Another Operation 218
9.11.3 Setting Permissions Standalone 219
9.11.4 Retrieving Graph Permissions 221

10.0 XQuery and JavaScript Semantics APIs 223
10.1 XQuery Library Module for Semantics 223
10.1.1 Importing the Semantics Library Module with XQuery 223
10.1.2 Importing the Semantics Library Module with JavaScript 224
10.2 Generating Triples 224
10.3 Extracting Triples from Content 225
10.4 Parsing Triples 228
10.5 Exploring Data 230
10.5.1 sem:triple Functions 231
10.5.2 Transitive Closure 231
10.5.2.1 Understanding Transitive Closure 231
10.5.2.2 sem:transitive-closure 232

11.0 Client-Side APIs for Semantics 237
11.1 Java Client API 237
11.2 Node.js Client API 237
11.3 Queries Using Optic API 238

12.0 Inserting, Deleting, and Modifying Triples with XQuery and Server-Side JavaScript 239
12.1 Updating Triples 239
12.2 Deleting Triples 241
12.2.1 Deleting Triples with XQuery or Server-Side JavaScript 241
12.2.1.1 sem:graph-delete 242
12.2.1.2 xdmp:node-delete 243
12.2.1.3 xdmp:document-delete 243
12.2.2 Deleting Triples with REST API 244

13.0 Using a Template to Identify Triples in a Document 247
13.1 Creating a Template 247
13.2 Template Elements 248

13.2.1 Reindexing Triggered by Templates 250
13.3 Examples 250
13.3.1 Validate and Insert a Template 250
13.3.2 Validate and Insert in One Step 253
13.3.3 Use a JSON Template 255
13.3.4 Identify Potential Triples 257
13.4 Triples Generated With TDE and SQL 259

14.0 Execution Plan 261
14.1 Generating an Execution Plan 261
14.2 Parsing an Execution Plan 262

15.0 Technical Support 265

16.0 Copyright 267
16.0 COPYRIGHT 267

1.0 Introduction to Semantics in MarkLogic

Semantics enables the discovery of facts and relationships in data, and provides context for those facts. Semantic technologies refer to a family of specific W3C standards that allow the exchange of information about relationships in data in machine readable form, whether it resides on the Web or within organizations. MarkLogic Semantics, using RDF (Resource Description Framework), allows you to natively store, search, and manage RDF triples using SPARQL query, SPARQL Update, and JavaScript, XQuery, or REST.

Semantics requires a flexible data model (RDF), query tool (SPARQL), a graph and triple data management tool (SPARQL Update), and a common markup language (for example RDFa, Turtle, N-Triples). MarkLogic lets you natively store, manage, and search triples using SPARQL and SPARQL Update.

RDF is one of the core technologies of linked open data. The framework provides standards for disambiguating data, integrating, and interacting with data that may come from disparate sources, both machine-readable and human-readable. It makes use of W3C recommendations and formal, defined vocabularies for data to be published and shared across the Semantic Web.

SPARQL (SPARQL Protocol and RDF Query Language) is used to query data in RDF serialization. SPARQL Update is used to create, delete, and update (delete/insert) triple data and graphs.

You can derive additional semantic information from your data using inference. You can also enrich your data using Linked Open Data (LOD), an extension of the World Wide Web created from the additional semantic metadata embedded in data.

> **Note:** Semantics is a separately licensed product. To use SPARQL features, a license that includes the Semantics Option is required. Use of APIs *leveraging* Semantics without using SPARQL, such as the Optic API or SQL API, does not require a Semantics Option license.

For more information, see the following resources:

- http://www.w3.org/standards/semanticweb
- http://www.w3.org/RDF
- http://www.w3.org/TR/rdf-sparql-query
- http://www.w3.org/TR/sparql11-update

This document describes how to load, query, and work with semantic data in MarkLogic Server. This chapter provides an overview of Semantics in MarkLogic Server. This chapter includes the following sections:

- Terminology
- Linked Open Data

- RDF Implementation in MarkLogic
- Example Datasets

1.1 Terminology

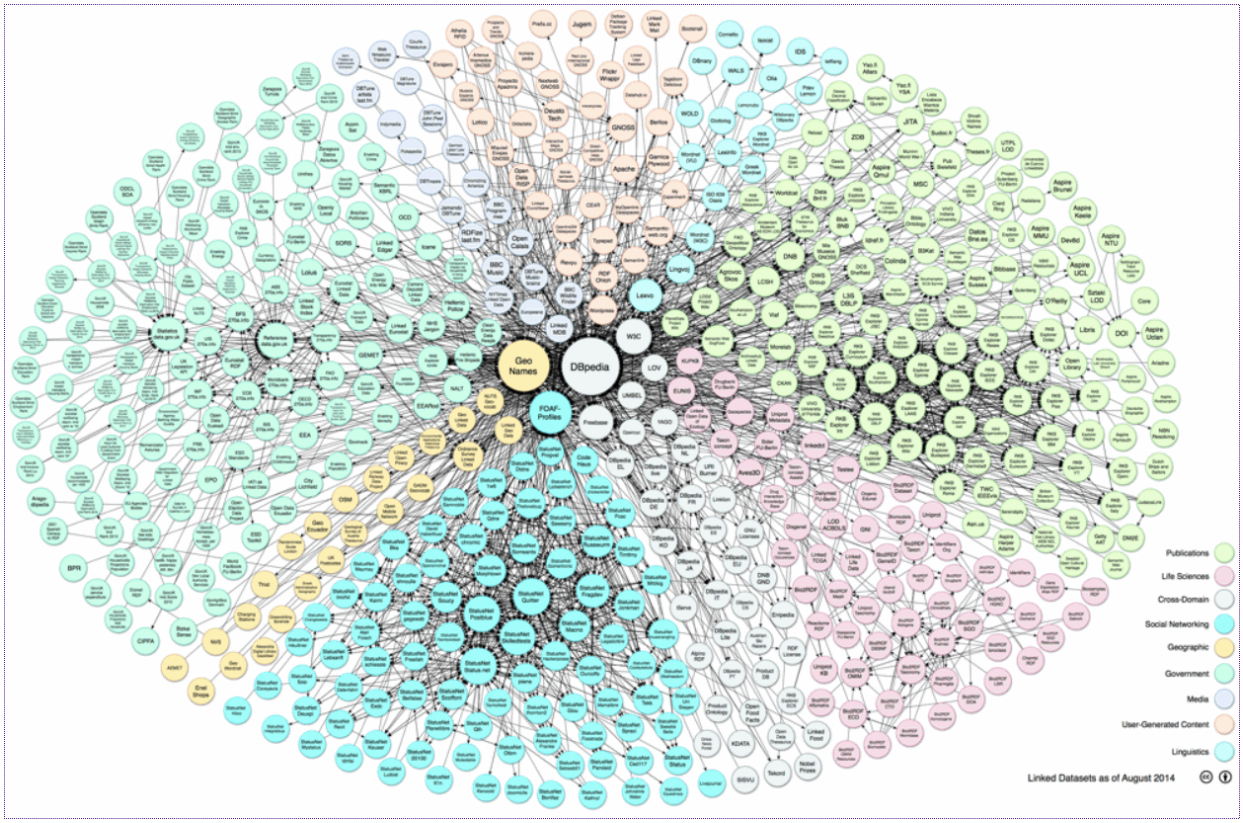
Terms used in this guide:

Term	Definition
RDF	RDF (Resource Description Framework) is a data model used to represent facts as a triple made up of a subject, predicate, and an object. The framework is W3C specification with a defined vocabulary.
RDF Triple	An RDF statement containing atomic values representing a subject, predicate, object, and optionally a graph. Each triple represents a single fact.
Subject	A representation of a resource such as a person or an entity. A node in an graph or triple.
Predicate	A representation of a property or characteristics of the subject or of the relationship between the subject and the object. The predicate is also known as an *arc* or *edge*.
Object	A node representing a property value, which in turn may be the subject in a triple or graph. An object may be a typed literal. See “RDF Datatypes” on page 21.
Graph	A set of RDF triple statements or patterns. In a graph-based RDF model, nodes represent subject or object resources, with the predicate providing the connection between those nodes. Graphs that are assigned a name are referred to as *Named Graphs*.
Quad	A representation of a subject, predicate, object, and an additional resource node for the context of the triple.
Vocabularies	A standard format for classifying terms. Vocabularies such as FOAF (Friend of a Friend) and Dublin Core (DC) define the concepts and relationships used to describe and represent facts. For example, OWL is a Web Ontology Language for publishing and sharing ontologies across the World Wide Web.
Triple Index	An index that indexes triples ingested into MarkLogic to facilitate the execution of SPARQL queries.
RDF Triple Store	A storage tool for the persistent storage, indexing, and query access to RDF graphs.

Term	Definition
IRI	An IRI (Internationalized Resource Identifier) is a compact string that is used for uniquely identifying resources in an RDF triple. IRIs may contain characters from the Universal Character Set (Unicode/ISO 10646), including Chinese or Japanese Kanji, Korean, Cyrillic characters, and so on.
CURIE	Compact URI Expression.
SPARQL	A recursive acronym for SPARQL Protocol and RDF Query Language (SPARQL), a query language designed for querying data in RDF serialization. SPARQL 1.1 syntax and functions are available in MarkLogic.
SPARQL Protocol	A means of conveying SPARQL queries from query clients to query processors, consisting of an abstract interface with bindings to HTTP (Hypertext Transfer Protocol) and SOAP (Simple Object Access Protocol).
SPARQL Update	An update language for RDF graphs that uses a syntax derived from the SPARQL Query language.
RDFa	Resource Description Framework in Attributes (RDFa) is a W3C Recommendation that adds a set of attribute-level extensions to HTML, XHTML, and various XML-based document types for embedding rich metadata within Web documents.
Blank node	A node in an RDF graph representing a resource for which a IRI or literal is not provided. The term *bnode* is used interchangeably with blank node.

1.2 Linked Open Data

Linked Open Data enables sharing of metadata and data across the Web. The World Wide Web provides access to resources of structured and unstructured data as human-readable web pages and hyperlinks. Linked Open Data extends this by inserting machine-readable metadata about pages and how they are related to each other to present semantically structured knowledge. The Linked Open Data Cloud gives some sense of the variety of open data sets available on the Web.



For more about Linked Open Data, see http://linkeddata.org/.

1.3 RDF Implementation in MarkLogic

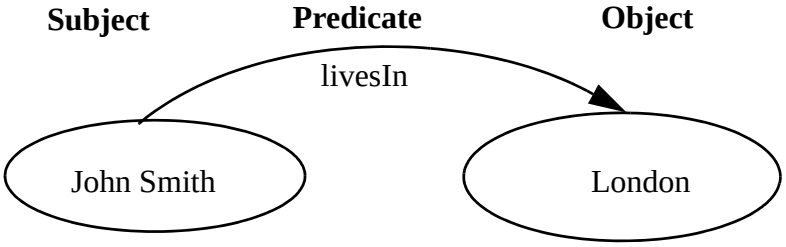
This section describes the semantic technologies using RDF that are implemented in MarkLogic Server and includes the following concepts:

- Using RDF in MarkLogic
- RDF Data Model
- RDF Datatypes
- RDF Vocabulary

1.3.1 Using RDF in MarkLogic

RDF is implemented in MarkLogic to store and search RDF triples. Specifically, each triple is an RDF triple statement containing a subject, predicate, object, and optionally a graph.

For example:



The subject node is a resource named John Smith, the object node is London, and the predicate, shown as an edge linking the two nodes, describes the relationship. From the example, the statement “John Smith lives in London” can be derived.

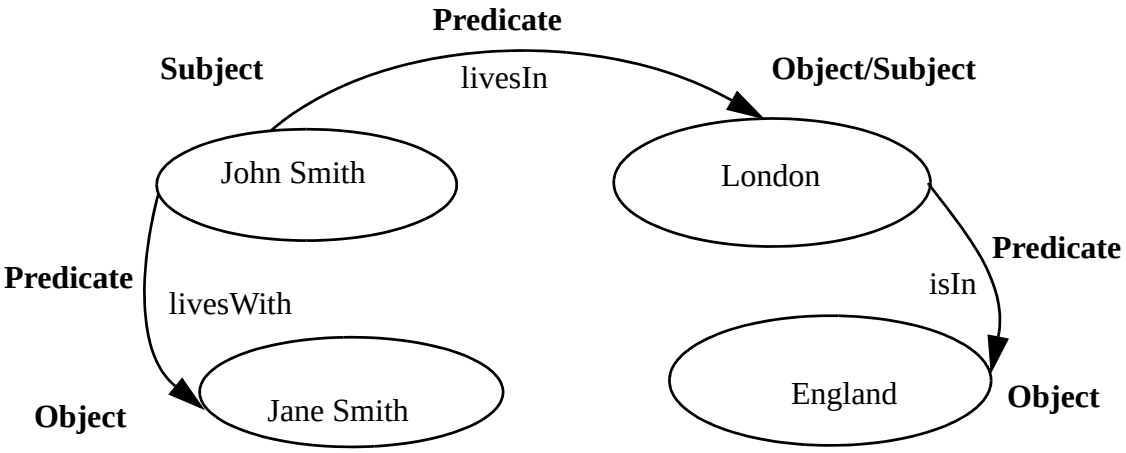
This triple looks like this in XML (with a second triple added):

```
<sem:triples xmlns:sem="http://marklogic.com/semantics">
  <sem:triple>
    <sem:subject> http://xmlns.com/foaf/0.1/name/"John
Smith"</sem:subject>
    <sem:predicate> http://example.org/livesIn</sem:predicate>
    <sem:object
datatype="http://www.w3.org/2001/XMLSchema#string">"London"</sem:objec
t>
  </sem:triple>
</sem:triples>
```

In JSON this same triple would look like:

```
{
  "my" : "data",
    "triple" : {
       "subject": "http://xmlns.com/foaf/0.1/name/John Smith",
       "predicate": "http://example.org/livesIn",
       "object": { "value": "London", "datatype": "xs:string" }
  }
}
```

Sets of triples are stored as RDF graphs. In MarkLogic, the graphs are stored as collections. The following image is an example of a simple RDF graph model that contains three triples. For more information about graphs, see “RDF Data Model” on page 20.



The object node of a triple can in turn be a subject node of another triple. In the example, the following facts are represented “John Smith lives with Jane Smith”, “John Smith lives in London” and “London is in England”.

The graph can be represented in tabular format:

Subject	Predicate	Object
John Smith	livesIn	London
London	isIn	England
John Smith	livesWith	Jane Smith

In JSON, these triples would look like this:

```
{
  "my" : "data",
    "triple" : [{
      "subject": "http://xmlns.com/foaf/0.1/name/John Smith",
      "predicate": "http://example.org/livesIn",
      "object": { "value": "London", "datatype": "xs:string" }
  },{
      "subject": "http://xmlns.com/foaf/0.1/name/London",
      "predicate": "http://example.org/isIn",
      "object": { "value": "England", "datatype": "xs:string" }
  },{
      "subject": "http://xmlns.com/foaf/0.1/name/John Smith",
      "predicate": "http://example.org/livesWith",
      "object": { "value": "Jane Smith", "datatype": "xs:string" }
}
]}
```

1.3.1.1 Storing RDF Triples in MarkLogic

When you load RDF triples into MarkLogic, the triples are stored in MarkLogic-managed XML documents. You can load triples from a document using an RDF serialization, such as Turtle or N-Triple. For example:

```
<http://example.org/dir/js> <http://xmlns.com/foaf/0.1/firstname>
"John" .
<http://example.org/dir/js> <http://xmlns.com/foaf/0.1/lastname>
"Smith" .
<http://example.org/dir/js> <http://xmlns.com/foaf/0.1/knows> "Jane
Smith" .
```

For more examples of RDF formats, see “Example RDF Formats” on page 39.

The triples in this example are stored in MarkLogic as XML documents, with `sem:triples` as the document root. These are managed triples because they have a document root element of `sem:triples`.

```
<?xml version="1.0" encoding="UTF-8"?>
<sem:triples xmlns:sem="http://marklogic.com/semantics">
 <sem:triple>
  <sem:subject>http://example.org/dir/js</sem:subject>
  <sem:predicate>http://xmlns.com/foaf/0.1/firstname</sem:predicate>
  <sem:object datatype="http://www.w3.org/2001/XMLSchema#string">John
  </sem:object>
 </sem:triple>
 <sem:triple>
  <sem:subject>http://example.org/dir/js</sem:subject>
  <sem:predicate>http://xmlns.com/foaf/0.1/lastname</sem:predicate>
  <sem:object datatype="http://www.w3.org/2001/XMLSchema#string">
      Smith</sem:object>
 </sem:triple>
 <sem:triple>
  <sem:subject>http://example.org/dir/js</sem:subject>
  <sem:predicate>http://xmlns.com/foaf/0.1/knows</sem:predicate>
  <sem:object datatype="http://www.w3.org/2001/XMLSchema#string">
      Jane Smith</sem:object>
 </sem:triple>
</sem:triples>
```

You can also embed triples within XML documents and load them into MarkLogic as-is. These are unmanaged triples, with a element node of `sem:triple`. You do not need the outer `sem:triples` element for unmanaged triples, but you do need the subject, predicate, and object elements within the `sem:triple` element.

Here is an embedded triple, contained in an XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<article>
  <info>
```

```
      <title>News for April 9, 2013</title>
       <sem:triples xmlns:sem="http://marklogic.com/semantics">
         <sem:triple>
          <sem:subject>http://example.org/article</sem:subject>
          <sem:predicate>http://example.org/mentions</sem:predicate>
          <sem:object>http://example.org/London</sem:object>
         <sem:triple>
       </sem:triples>
...
  </info>
</article>
```

The loaded triples are automatically indexed with a special-purpose index called a triple index. The triple index allows you to immediately search the RDF data for which you have the required privileges.

1.3.1.2 Querying Triples

You can write native SPARQL queries in Query Console to retrieve information from RDF triples stored in MarkLogic or in memory. When queried with SPARQL, the question of "who lives in England?" is answered with "John and Jane Smith". This is based on the assertion of facts from the above graph model. This is an example of a simple SPARQL `SELECT` query:

```
SELECT ?person ?place
WHERE
{
  ?person <http://example.org/livesIn> ?place .
  ?place <http://example.org/isIn>
http://xmlns.com/foaf/0.1/name/London.
}
```

You can also use XQuery to execute SPARQL queries with `sem:sparql`. For example:

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

sem:sparql("
PREFIX kennedy:<http://example.org/kennedy>
SELECT *
WHERE
{
?s ?p ?o .
FILTER (regex(?o, 'Joseph', 'i'))
}
")
```

For more information about using SPARQL and `sem:sparql` to query triples, see "Semantic Queries" on page 81.

Using XQuery, you can query across triples, documents, and values with `cts:triples` or `cts:triple-range-query`.

Here is an example using a `cts:triples` query:

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

let $r :=
  cts:triples(sem:iri("http://example.org/people/dir"),
    sem:iri("http://xmlns.com/foaf/0.1/knows"),
    sem:iri("person1"))

return <result>{$r}</result>
```

The following is an example of a query that uses `cts:triple-range-query`:

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

let $query := cts:triple-range-query(
  sem:iri("http://example.org/people/dir"),
  sem:iri("http://xmlns.com/foaf/0.1/knows"), ("person2"), "sameTerm")

return cts:search(fn:collection()//sem:triple, $query)
```

You can create combination queries with `cts:query` functions such as `cts:or-query` or `cts:and-query`.

For example:

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

declare namespace dc = "http://purl.org/dc/elements/1.1/";

  cts:search(collection()//sem:triple, cts:or-query((
    cts:triple-range-query((), sem:curie-expand("foaf:name"),
      "Lamar Alexander", "="),
    cts:triple-range-query(sem:iri("http://www.rdfabout.com/rdf/usgov
      /congress/people/A000360"), sem:curie-expand("foaf:img"), (),
  "=")))
```

For more information about `cts:triples` and the `cts:triple-range-query` queries, see “Semantic Queries” on page 81.

You can also use the results of a SPARQL query with an XQuery search to create combination queries.

For example:

```
xquery version "1.0-ml";
import module namespace sem = "http://marklogic.com/semantics"
  at "/MarkLogic/semantics.xqy";
import module namespace semi = "http://marklogic.com/semantics/impl"
  at "/MarkLogic/semantics/sem-impl.xqy";

declare namespace sr = "http://www.w3.org/2005/sparql-results";

let $results := sem:sparql( "prefix k: <http://example.org/kennedy>
select * { ?s k:latitude ?lat . ?s k:longitude ?lon }" )
let $xml := sem:sparql($results)

return
for $sol in $xml/sr:results/sr:result
let $point := cts:point(xs:float($sol/sr:binding[@name eq
'lat']/sr:literal), xs:float($sol/sr:binding[@name eq
'lon']/sr:literal))
return <place name="{$sol/sr:binding[@name eq 's']/*}"
point="{$point}"/>
```

For more information about combination queries, see “Querying Triples with XQuery or JavaScript” on page 128.

1.3.2 RDF Data Model

RDF triples are a convenient way to represent facts: facts about the world, facts about a domain, facts about a document. Each RDF triple is a fact (or assertion) represented by a subject, predicate, and object, such as “John livesIn London”. The subject and predicate of a triple must be an IRI (Internationalized Resource Identifier), which is a compact string used to uniquely identify resources. The object may be either an IRI or a literal, such as a number or string.

- Subjects and predicates are IRI references with an optional fragment identifier. For example:

  ```
  <http://xmlns.com/foaf/0.1/Person>
  foaf:person
  ```

- Literals are strings with an optional language tag or a number. These are used as objects in RDF triples. For example:

  ```
  "Bob"
  "chat" @fr
  ```

- Typed literals may be strings, integers, dates and so on, that are assigned to a datatype. These literals are typed with a “^^” operator “”. For example:

  ```
  "Bob"^^xs:string
  "3"^^xs:integer
  "26.2"^^xs:decimal
  ```

In addition, a subject or object may be a blank node (*bnode* or anonymous node), which is a node in a graph without a name. Blank nodes are represented with an underscore, followed by a colon (:) and then an identifier. For example:

```
_:a
_:jane
```

For more information about IRIs, see “IRIs and Prefixes” on page 22.

Often the object of one triple is the subject of another, so a collection of triples forms a graph. In this document we represent graphs using these conventions:

- Subjects and objects are shown as ovals.
- Predicates are shown as edges (labeled arrows).
- Typed literals are shown as boxes.

1.3.3 Blank Node Identifiers

In MarkLogic, a blank node is assigned a blank node identifier. This internal identifier is maintained across multiple invocations. In a triple, a blank node can be used for the subject or object and is specified by an underscore (_). For example:

```
_:jane <http://xmlns.com/foaf/0.1/name> "Jane Doe".
<http://example.org/people/about> <http://xmlns.com/foaf/0.1/knows>
_:jane
```

Given two blank nodes, you can determine whether or not they are the same. The first node "`_:jane`" will refer to the same node as the second invocation that also mentions `"_:jane"`. Blank nodes are represented as skolemized IRIs: blank nodes where existential variables are replaced with unique constants. Each blank node has a prefix of "`http://marklogic.com/semantics/blank`".

1.3.4 RDF Datatypes

RDF uses the XML schema datatypes. These include `xs:string`, `xs:float`, `xs:double`, `xs:integer`, and `xs:date` and so on, as described in the specification, *XML Schema Part 2: Datatypes Second Edition*:

http://www.w3.org/TR/xmlschema-2

All XML schema simple types are supported, along with all types derived from them, except for `xs:QName` and `xs:NOTATION`.

RDF can also contain custom datatypes that are named with a IRI. For example, a supported MarkLogic-specific datatype is `cts:point`.

Note: Use of an unsupported datatype such as `xs:QName`, `xs:NOTATION`, or types derived from these will generate an XDMP-BADRDFVAL exception.

If you omit a datatype declaration, it is considered to be of type `xs:string`. A typed literal is denoted by the presence of the `datatype` attribute, or by an `xml:lang` attribute to give the language encoding of the literal, for example, “en” for English.

Datatypes in the MarkLogic Semantics data model allow for values with a datatype that has no schema. These are identified as `xs:untypedAtomic`.

1.3.5 IRIs and Prefixes

This section describes meaning and role of IRIs and prefixes, and includes the following concepts:

- IRIs
- Prefixes

1.3.5.1 IRIs

IRIs (Internationalized Resource Identifiers) are internationalized versions of URIs (Uniform Resource Identifiers). URIs use a subset of ASCII characters and are limited to this set. IRIs use characters beyond ASCII, making them more useful in an international context. IRIs (and URIs) are unique resource identifiers that enable you to fetch a resource. A URN (Uniform Resource Name) can also be used to uniquely identify a resource.

An IRI may appear similar a URL and may or may not be an actual website. For example:

```
<http://example.org/addressbook/d>
```

IRIs are used instead of URIs, where appropriate, to identify resources. Since SPARQL specifically refers to IRIs, later chapters in this guide reference IRIs and not URIs.

IRIs are required to eliminate ambiguity in facts, particularly if data is received from different data sources. For example, if you are receiving information about books from different sources, one publisher may refer to the name of the book as “title”, another source may refer to the position of the author as “title”. Similarly, one domain may refer to the writer of the book as the “author” and another as “creator”.

Presenting the information with IRIs (and URNs), we see a clearer presentation of what the facts mean. The following examples are three sets of N-Triples:

```
<http://example.org/people/title/sh1999>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#label>
"Lucasian Professor of Mathematics"

<urn:isbn:9780553380163>
<http://purl.org/dc/elements/1.1/title>
"A Brief History of Time"

<urn:isbn:9780553380163>
```

```
<http://purl.org/dc/elements/1.1/creator>
"Stephen Hawking"
```

Note: Line breaks have been inserted for the purposes of formatting, which make this RDF N-Triple syntax invalid. Each triple would normally be on one line. (Turtle syntax allows for single triples to wrap across multiple lines.)

The IRI is a key component of RDF, however IRIs are usually long and are difficult to maintain. Compact URI Expressions (CURIEs) are supported as a mechanism for abbreviating IRIs. These are specified in the CURIE Syntax Definition:

http://www.w3.org/TR/rdfa-syntax/#s_curies

1.3.5.2 Prefixes

Prefixes are identified by IRIs and often begin with the name of an organization or company. For example:

```
PREFIX js: <http://example.org/people/about/js/>
```

A prefix is a shorthand string used to identify a name. The designated prefix binds a prefix IRI to the specified string. The prefix can then be used instead of writing the full IRI each time it is referenced. When you use prefixes to write RDF, the prefix is followed by a colon. You can choose any prefix for resources that you define. For example, here is a SPARQL declaration:

```
PREFIX dir: <http://example.org/people/about/>
```

You can also use standard and agreed upon prefixes that are a part of a specification. This is a SPARQL declaration for rdf:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns/>
```

The prefix depends on the serialization that you use. The Turtle prefix declaration would be:

```
@prefix dir: <http://example.org/people/about/> .
```

Note: All `PREFIX` declarations must end with a forward slash (“/”) or a hashtag (“#”). These separate the prefix from the final part of the IRI.

1.3.6 RDF Vocabulary

RDF vocabularies are defined using RDF Schema (RDFS) or Web Ontology Language (OWL) to provide a standard serialization for classifying terms. The vocabulary is essentially the set of IRIs for the arcs that form RDF graphs. For example, the FOAF vocabulary describes people and relationships.

The existence of a shared standard vocabulary is helpful, but not essential since it is possible to combine vocabularies or create a new one. Use the following prefix lookup to help decide which vocabulary to use:

http://prefix.cc/about

There is an increasingly large number of vocabularies. Common RDF prefixes that are widely used and agreed upon include the following:

Prefix	Prefix IRI	
cc	http://web.resource.org/cc#ns	Creative Commons
dc	http://purl.org/dc/elements/1.1/	Dublin Core vocabulary
dcterms	http://purl.org/dc/terms	Dublin Core terms
rdfs	http://www.w3.org/2000/01/rdf-schema#	RDF schema
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#	RDF vocabulary
owl	http://www.w3.org/2002/07/owl#	Web Ontology Language
foaf	http://xmlns.com/foaf/0.1/	FOAF (Friend of a Friend)
skos	http://www.w3.org/2004/02/skos/core	SKOS (Simple Knowledge Organization System)
vcard	http://www.w3.org/2001/vcard-rdf/3.0	VCard vocabulary
void	http://rdfs.org/ns/void	Vocabulary of Interlinked Datasets
xml	http://www.w3.org/XML/1998/namespace	XML namespace
xhtml	http://www.w3.org/1999/xhtml	XHTML namespace
xs	http://www.w3.org/2001/XMLSchema#	XML Schema
fn	http://www.w3.org/2005/xpath-functions	XQuery function and operators

Note: For these vocabularies, the IRIs are also URLs.

1.4 Example Datasets

There is a growing body of data from domains such as Government and governing agencies, Healthcare, Finance, Social Media and so on, available as triples, often accessible via SPARQL for the purpose of:

- Semantic search

- Dynamic Semantic Publishing
- Aggregating diverse datasets

There are a large number of datasets available for public consumption.

For example:

- FOAF: http://www.foaf-project.org - a project that provides a standard RDF vocabulary for describing people, what they do, and relationships to other people or entities.
- DBPedia: http://wiki.dbpedia.org/develop/datasets/ - data derived from Wikipedia with many external links to RDF datasets.
- Semantic Web: http://data.semanticweb.org - a database of thousands of unique triples about conference data.

2.0 Getting Started with Semantics in MarkLogic

This chapter includes the following sections:

- Setting up MarkLogic Server
- Loading Triples
- Querying Triples

2.1 Setting up MarkLogic Server

When you install MarkLogic Server, a database, REST instance, and XDBC server (for loading content) are created automatically for you. The default Documents database is available on port 8000 as soon as you install MarkLogic Server, with a REST instance attached to it.

The examples in this guide use this pre-configured database, XDBC server, and REST API instance on port 8000. This section focuses on setting up MarkLogic Server to store triples. To do this, you will need to configure the database to store, search, and manage triples.

> **Note:** You must have admin privileges for MarkLogic Server to complete the procedures described in this section.

Install MarkLogic Server on the database server, as described in the *Installation Guide for All Platforms*, and then perform the following tasks:

- Configuring the Database to Work with Triples
- Setting Up Additional Servers

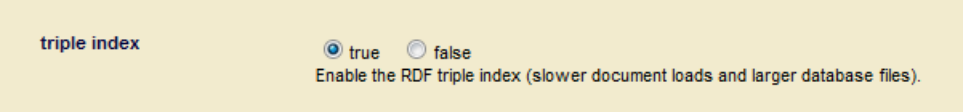
2.1.1 Configuring the Database to Work with Triples

In MarkLogic 9 and later, the Documents database has the triple index and the collection lexicon enabled by default. These options are also enabled by default for any new databases you create. You can skip the rest of these steps if you are using a new install of MarkLogic 9.

If you have an *existing* database that you want to use for triples, you need to enable the triple index and the collection lexicon. You can also use these steps to verify that a database is correctly set up. Follow these steps to configure an existing database for triples:

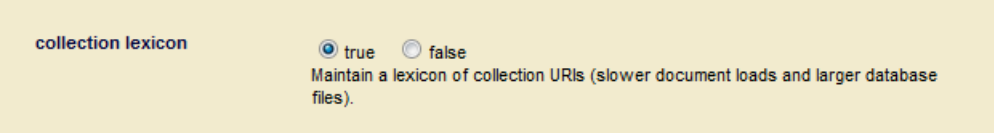
1. Navigate to the Admin Interface (`localhost:8001`). Click the Documents database, and then click the Configure tab.

2. Scroll down the Admin Configure page to see the status of triple index.



Set this to true if it is not already configured. The triple index is used for semantics.

3. Scroll down a bit further and set the collection lexicon to true.



The collection lexicon index is required and used by the REST endpoint. You will only need the collection lexicon if you are querying a named graph.

4. Click ok when you are done.

This is all you need to do before loading triples into your default database (the Documents database).

Note: For all new installations of MarkLogic 9 and later, the triple index and collection lexicon are enabled by default. Any new databases will also have the triple index and collection lexicon enabled.

2.1.2 Setting Up Additional Servers

In a production enviroment, you will want to create additional app servers, REST instances, and XDBC servers. Use these links to find out more:

- Application servers: The process to create additional app servers is described in Creating and Configuring App Servers in the *Administrator's Guide*.
- REST instances: To create a different REST instance on another port, see Administering REST Client API Instances in the *REST Application Developer's Guide*.
- XDBC servers: The process to create an XDBC server is described in detail in Creating a New XDBC Server in the *Administrator's Guide*.

2.2 Loading Triples

This section covers loading triples into the triples database. It includes the following topics:

- Downloading the Dataset
- Importing Triples with mlcp
- Verifying the Import

2.2.1 Downloading the Dataset

Use the full sample of Persondata from DBPedia (in English and Turtle serialization) for the examples, or use a different subset of Persondata if you prefer.

1. Download the Persondata example dataset from DBPedia at http://downloads.dbpedia.org/2016-10/core-i18n/en/persondata_en.ttl.bz2. You will use this dataset for the steps in the rest of this chapter.

   Note: DBPedia datasets are licensed under the terms of the of the Creative Commons Attribution-ShareAlike License and the GNU Free Documentation License. The data is available in localized languages and in N-Triple and N-Quad serialized formats.

2. Extract the data from the compressed file to a local directory, for example, `C:\space`.

2.2.2 Importing Triples with mlcp

There are multiple ways to load triples into MarkLogic, including MarkLogic Content Pump (mlcp), REST endpoints, and XQuery. The recommended way to bulk-load triples is with mlcp. These examples use mlcp on a stand-alone Windows environment.

1. Install and configure MarkLogic Pump as described in Installation and Configuration in the *mlcp User Guide*.

2. In the Windows command interpreter, `cmd.exe`, navigate to the mlcp bin directory for your mlcp installation. For example:

   ```
   cd C:\space\marklogic-contentpump-1.3-1\bin
   ```

3. Assuming that the Persondata is saved locally under `C:\space`, enter the following single-line command at the prompt:

   ```
   mlcp.bat import -host localhost -port 8000 -username admin ^
   -password password -input_file_path c:\space\persondata_en.ttl ^
   -mode local -input_file_type RDF -output_uri_prefix /people/
   ```

   For clarity the long command line is broken into multiple lines using the Windows line continuation character "^". Remove the line continuation characters to use the command.

   The modified command for UNIX is:

   ```
   mlcp.sh import -host localhost -port 8000 -username admin -password\
   password -input_file_path /space/persondata_en.ttl -mode local\
   -input_file_type RDF -output_uri_prefix /people/
   ```

   For clarity, the long command line is broken into multiple lines using the UNIX line continuation character "\". Remove the line continuation characters to use the command.

   The triples will be imported and stored in the default Documents database.

4. Lots of lines of text will display in your command line window, perhaps with what appear to be warning messages. This is normal. The successful triples data import (UNIX output) looks like this when it is complete:

```
14/09/15 14:35:38 INFO contentpump.ContentPump: Hadoop library version:
2.0.0-alpha
14/09/15 14:35:38 INFO contentpump.LocalJobRunner: Content type: XML
14/09/15 14:35:38 INFO input.FileInputFormat: Total input paths to
process : 1
O:file:///home/persondata_en.ttl : persondata_en.ttl
14/09/15 14:35:40 INFO contentpump.LocalJobRunner:  completed 0%
14/09/15 14:40:27 INFO contentpump.LocalJobRunner:  completed 100%
14/09/15 14:40:28 INFO contentpump.LocalJobRunner:
com.marklogic.contentpump.ContentPumpStats:
14/09/15 14:40:28 INFO contentpump.LocalJobRunner:
ATTEMPTED_INPUT_RECORD_COUNT: 59595
14/09/15 14:40:28 INFO contentpump.LocalJobRunner:
SKIPPED_INPUT_RECORD_COUNT: 0
14/09/15 14:40:28 INFO contentpump.LocalJobRunner: Total execution
time: 289 sec
```

2.2.3 Verifying the Import

To verify that the RDF triples are successfully loaded into the triples database, do the following.

1. Navigate to the REST Server with a Web browser:

   `http://`*`hostname`*`:8000`

   where *hostname* is the name of your MarkLogic Server host machine, and `8000` is the default port number for the REST instance that was created when you installed MarkLogic Server.

2. Append "`/`*`latest`*`/graphs/things`" to the end of the web address URL where *`latest`* is the latest version of the REST API. For example:

   ```
   http://hostname:8000/v1/graphs/things
   ```

The first one thousand subjects are displayed:



3. Click on a subject link to view the triples. Subject and object IRIs are presented as links.



2.3 Querying Triples

You can run SPARQL queries in Query Console or via an HTTP endpoint using the `/v1/graphs/sparql` endpoint (`GET:/v1/graphs/sparql` and `POST:/v1/graphs/sparql`). This section includes the following topics:

- Querying with Native SPARQL
- Querying with the sem:sparql Functions

Note: This section assumes you loaded the sample dataset as described in “Downloading the Dataset” on page 29.

2.3.1 Querying with Native SPARQL

You can run queries in Query Console using native SPARQL or the built-in function `sem:sparql`.

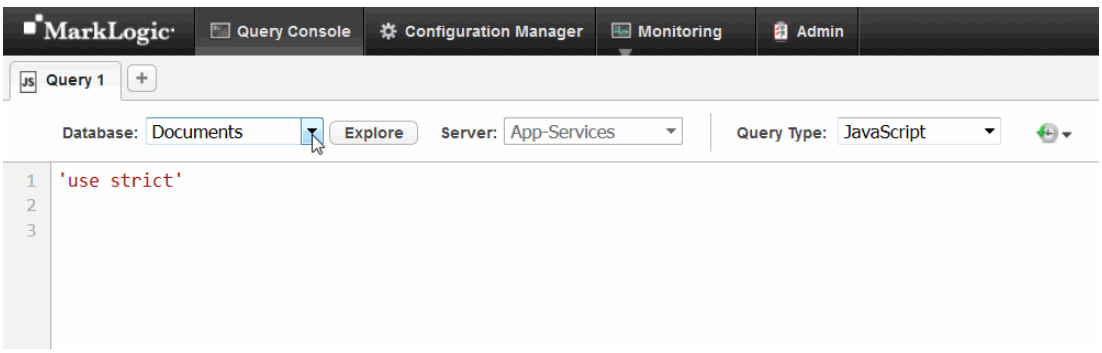
To run queries:

1. Navigate to Query Console with a Web browser:

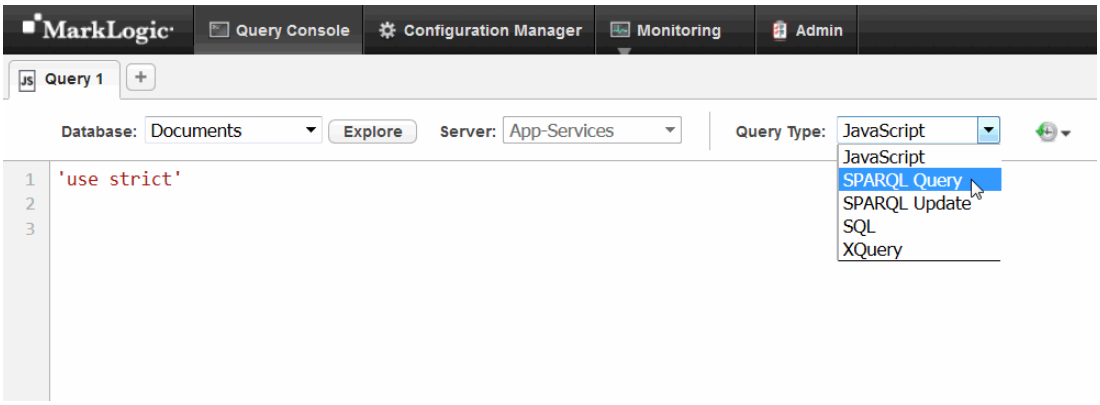
   ```
   http://hostname:8000/qconsole
   ```

   where *hostname* is the name of your MarkLogic Server host.

2. From the Database drop-down list, select the Documents database.



3. From the Query Type drop-down list, select SPARQLQuery.

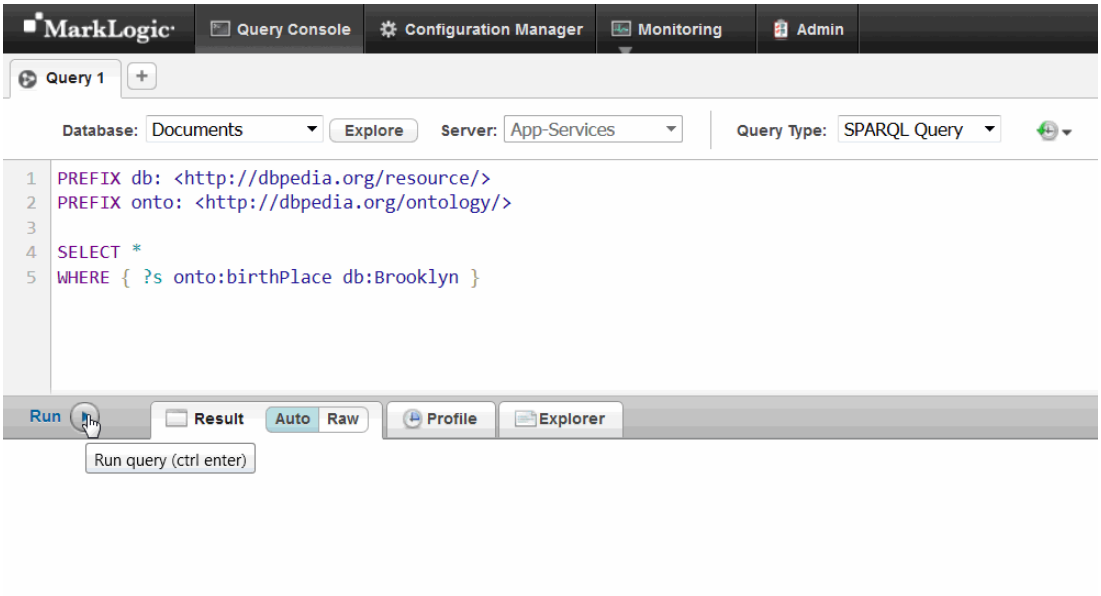


4. In the query window, replace the default query text with this SPARQL query:

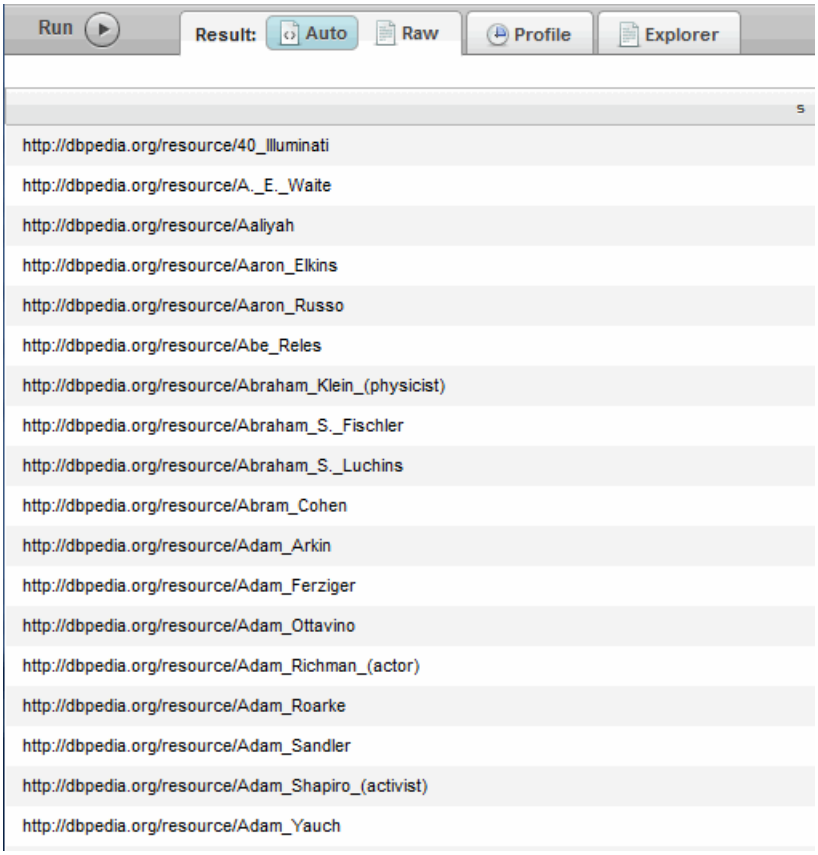
```
PREFIX db: <http://dbpedia.org/resource/>
PREFIX onto: <http://dbpedia.org/ontology/>

SELECT *
WHERE { ?s onto:birthPlace db:Brooklyn }
```

5. Click Run.



The results show people born in Brooklyn as IRIs.



2.3.2 Querying with the sem:sparql Functions

Use the built-in XQuery function `sem:sparql` in Query Console to run the same query.

1. From the Database drop-down list, select the Documents database.

2. From the Query Type drop-down list, select “XQuery”.

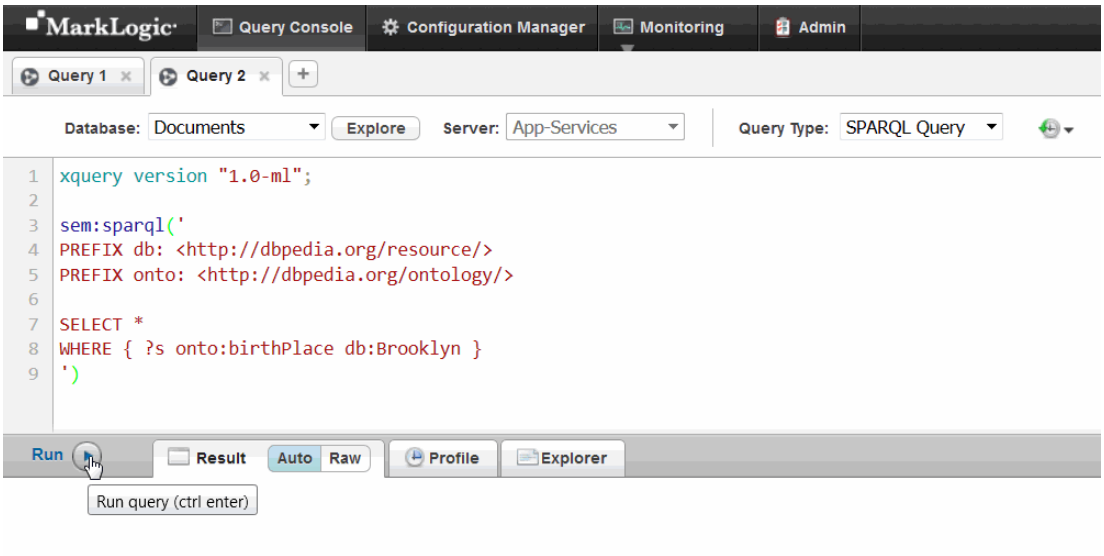
3. In the query window, enter this query:

```
xquery version "1.0-ml";

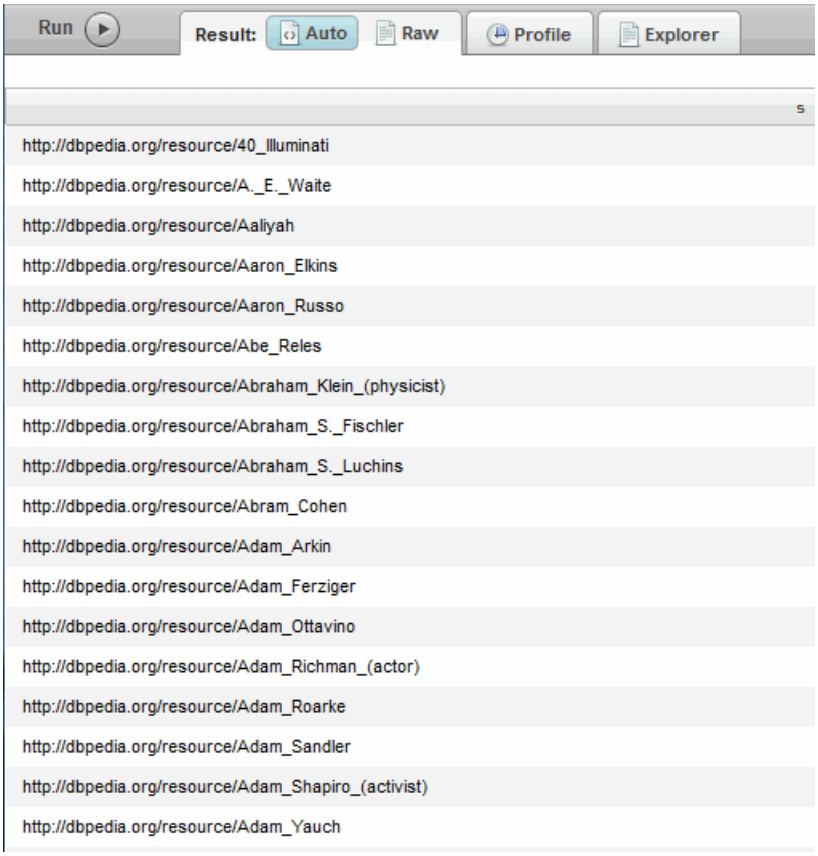
sem:sparql('
PREFIX db: <http://dbpedia.org/resource/>
PREFIX onto: <http://dbpedia.org/ontology/>

SELECT *
WHERE { ?s onto:birthPlace db:Brooklyn }
')
```

4. Click Run.



5. The results contain IRIS showing people born in Brooklyn, the same as in “Querying with Native SPARQL” on page 32.



For more information and examples of SPARQL queries, see “Semantic Queries” on page 81.

3.0 Loading Semantic Triples

You can load triples into a MarkLogic database from an XML document or JSON file that contains embedded triples elements, or from triples files containing serialized RDF data. This chapter includes the following sections:

- Loading Embedded RDF Triples
- Loading Triples

You can also use SPARQL Update to load triples. See “SPARQL Update” on page 169 for more information.

3.1 Loading Embedded RDF Triples

Load documents that contain embedded triples in XML documents or JSON documents with any of the ingestion tools described in Available Content Loading Interfaces in the *Loading Content Into MarkLogic Server Guide*.

> **Note:** The embedded triples must be in the MarkLogic XML format defined in the schema for `sem:triple` (`semantics.xsd`).

Triples ingested into a MarkLogic database are indexed by the triples index and stored for access and query by SPARQL. See “Storing RDF Triples in MarkLogic” on page 17 for details.

3.2 Loading Triples

There are multiple ways to load documents containing triples serialized in a supported RDF serialization into MarkLogic. “Supported RDF Triple Formats” on page 38 describes these RDF formats.

When you load one or more groups of triples, they are parsed into generated XML documents. A unique IRI is generated for every XML document. Each document can contain multiple triples.

> **Note:** The setting for the number of triples stored in documents is defined by MarkLogic Server and is not a user configuration.

Ingested triples are indexed with the triples index to provide access and the ability to query the triples with SPARQL, XQuery, or a combination of both. You can also use a REST endpoint to execute SPARQL queries and return RDF data.

If you do not provide a graph for the triple, the triples will be stored in a default graph that uses a MarkLogic Server feature called a collection. MarkLogic Server tracks the default graph with the collection IRI `http://marklogic.com/semantics#default-graph`.

You can specify a different collection during the load process and load triples into a named graph. For more information about collections, see Collections in the *Search Developer’s Guide*.

Note: If you insert triples into a database without specifying a graph name, the triples will be inserted into the default graph (`http://marklogic.com/semantics#default-graph`). If you insert triples into a super database and run `fn:count(fn:collection())` in the super database, you will get a DUPURI exception for duplicate URIs.

The generated XML documents containing the triple data are loaded into a default directory named `/triplestore`. Some loading tools let you specify a different directory. For example, when you load triples using mlcp, you can specify the graph and the directory as part of the import options. For more information, see “Loading Triples with mlcp” on page 44.

This section includes the following topics:

- Supported RDF Triple Formats
- Example RDF Formats
- Loading Triples with mlcp
- Loading Triples with XQuery
- Loading Triples with JavaScript
- Loading Triples Using the REST API
- Loading Triples Using the Java API
- Loading Triples Using the Node.js API

3.2.1 Supported RDF Triple Formats

MarkLogic Server supports loading these RDF data formats:

Format	Description	File Type	MIME Type
RDF/XML	A syntax used to serialize an RDF graph as an XML document. For an example, see “RDF/XML” on page 39.	.rdf	application/rdf+xml
Turtle	Terse RDF Triple Language (Turtle) serialization is a simplified subset of Notation 3 (N3), used for expressing data in the lowest common denominator of serialization. For an example, see “Turtle” on page 40.	.ttl	text/turtle
RDF/JSON	A syntax used to serialize RDF data as JSON objects. For an example, see “RDF/JSON” on page 40.	.json	application/rdf+json

Format	Description	File Type	MIME Type
N3	Notation3 (N3) serialization is a non-XML syntax used to serialize RDF data. For an example, see “N3” on page 41.	.n3	text/n3
N-Triples	A plain text serialization for RDF graphs. N-Triples is a subset of Turtle and Notation3 (N3). For an example, see “N-Triples” on page 41.	.nt	application/n-triples
N-Quads	A superset serialization that extends N-Triples with an optional context value. For an example, see “N-Quads” on page 43.	.nq	application/n-quads
TriG	A plain text serialization for RDF-named graphs and RDF datasets. For an example, see “TriG” on page 44.	.trig	application/trig

3.2.2 Example RDF Formats

This section includes examples for the following RDF formats:

- RDF/XML
- Turtle
- RDF/JSON
- N3
- N-Triples
- N-Quads
- TriG

3.2.2.1 RDF/XML

RDF/XML is the original standard for writing unique RDF syntax as XML. It is used to serialize an RDF graph as an XML document.

This example defines three prefixes: “rdf”, “xsd”, and “d”.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:d="http://example.org/data/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
  <rdf:Description rdf:about="http://example.org/data#item22">
    <d:shipped rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
      2013-05-14</d:shipped>
    <d:quantity
```

```
rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
      27</d:quantity>
    <d:invoiced
rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">
      true</d:invoiced>
    <d:costPerItem
rdf:datatype="http://www.w3.org/2001/XMLSchema#decimal">
      10.50</d:costPerItem>
  </rdf:Description>
</rdf:RDF>
```

3.2.2.2 Turtle

Terse RDF Triple Language (or Turtle) serialization expresses data in the RDF data model using a syntax similar to SPARQL. Turtle syntax expresses triples in the RDF data model in groups of three IRIs.

For example:

```
<http://example.org/item/item22>
<http://example.org/details/shipped>
"2013-05-14"^^<http://www.w3.org/2001/XMLSchema#dateTime> .
```

This triple states that item 22 was shipped on May 14th, 2013.

Turtle syntax provides a way to abbreviate information for multiple statements using `@prefix` to factor out the common portions of IRIs. This makes it quicker to write RDF Turtle statements. The syntax resembles RDF/XML, however unlike RDF/XML, it does not rely on XML. Turtle syntax is also valid Notation3 (N3) since Turtle is a subset of N3.

> **Note:** Turtle can only serialize valid RDF graphs.

In this example, four triples describe a transaction. The “shipped” object is assigned a “date” datatype, making it a typed literal enclosed in quotes. There are three untyped literals for the “quantity”, “invoiced”, and “costPerItem” objects.

```
@prefix i: <http://example.org/item> .
@prefix dt: <http://example.org/details#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
  i:item22 dt:shipped "2013-05-14"^^xsd:date .
  i:item22 dt:quantity 100 .
  i:item22 dt:invoiced true .
  i:item22 dt:costPerItem 10.50 .
```

3.2.2.3 RDF/JSON

RDF/JSON is a textual syntax for RDF that allows an RDF graph to be written in a form compatible with JavaScript Object Notation (JSON).

For example:

```
{ "http://example.com/directory#m":
  { "http://example.com/ns/person#firstName":
    [ { "value": "Michelle",
        "type": "literal",
        "datatype": "http://www.w3.org/2001/XMLSchema#string" }
    ]
  }
}
```

3.2.2.4 N3

Notation3 (N3) is a non-XML syntax used to serialize RDF graphs in a more compact and readable form than XML RDF notation. N3 includes support for RDF-based rules.

When you have several statements about the same subject in N3, you can use a semicolon (;) to introduce another property of the same subject. You can also use a comma to introduce another object with the same predicate and subject.

For example:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix foafcorp: <http://xmlns.com/foaf/corp/> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0> .
@prefix sec: <http://www.rdfabout.com/rdf/schema/ussec> .
@prefix id: <http://www.rdfabout.com/rdf/usgov/sec/id> .

id:cik0001265081 sec:hasRelation [
  dc:date "2008-06-05";
  sec:corporation id:cik0001000045;
  rdf:type sec:OfficerRelation;
  sec:officerTitle "Senior Vice President, CFO"] .
id:cik0001000180 sec:cik "0001000180";
  foaf:name "SANDISK CORP";
  sec:tradingSymbol "SNDK";
  rdf:type foafcorp:Company.
id:cik0001009165 sec:cik "0001009165";
  rdf:type foaf:Person;
  foaf:name "HARARI ELIYAHOU ET AL";
  vcard:ADR [ vcard:Street "601 MCCARTHY BLVD.; ";
vcard:Locality "MILPITAS, CA"; vcard:Pcode "95035" ] .
```

3.2.2.5 N-Triples

N-Triples is a plain text serialization for RDF graphs. It is a subset of Turtle, designed to be simpler to use than Turtle or N3. Each line in N-Triples syntax encodes one RDF triple statement and consists of the following:

- Subject (an IRI or a blank node identifier), followed by one or more characters of whitespace

- Predicate (an IRI), followed by one or more characters of whitespace
- Object (an IRI, blank node identifier, or literal) followed by a period (.) and a new line.

Typed literals may include language tags to indicate the language. In this N-Triples example, `@en-US` indicates that title of the resource is in US English.

```
<http://www.w3.org/2001/sw/RDFCore/ntriples>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://xmlns.com/foaf/0.1/Document> .
<http://www.w3.org/2001/sw/RDFCore/ntriples/>
<http://purl.org/dc/terms/title> "Example Doc"@en-US .
<http://www.w3.org/2001/sw/RDFCore/ntriples/>
<http://xmlns.com/foaf/0.1/maker> _:jane .
<http://www.w3.org/2001/sw/RDFCore/ntriples/>
<http://xmlns.com/foaf/0.1/maker> _:joe .
  _:jane <http://www.w3.org/1999/02/22-rdf-syntax-ns>
  <http://xmlns.com/foaf/0.1/Person> .
  _:jane <http://xmlns.com/foaf/0.1/name> "Jane Doe".

  _:joe <http://www.w3.org/1999/02/22-rdf-syntax-ns>
  <http://xmlns.com/foaf/0.1/Person> .
  _:joe <http://xmlns.com/foaf/0.1/name> "Joe Bloggs".
```

Note: Each line breaks after the end period. For clarity, additional line breaks have been added.

3.2.2.6 N-Quads

N-Quads is a line-based, plain text serialization for encoding an RDF dataset. N-Quads syntax is a superset of N-Triples, extending N-Triples with an optional context value. The simplest statement is a sequence of terms (subject, predicate, object) forming an RDF triple, and an optional IRI labeling the graph in a dataset to which the triple belongs. All of these are separated by a whitespace and terminated by a period (.) at the end of each statement.

This example uses the relationship vocabulary. The class or property in the vocabulary has a IRI constructed by appending a term name “acquaintanceOf” to the vocabulary IRI.

```
<http://example.org/#Jane>
<http://http://purl.org/vocab.org/relationship/#acquaintanceOf>
<http://example.org/#Joe>
<http://example.org/graphs/directory> .
```

3.2.2.7 TriG

TriG is a plain text serialization for serializing RDF graphs. TriG is similar to Turtle, but is extended with curly braces ({) and (}) to group triples into multiple graphs and precede named graphs with their names. An optional equals operator (=) can be used to assign graph names and an optional end period (.) is included for Notation3 compatibility.

Characteristics of TriG serialization include:

- Graph names must be unique within a TriG document, with one unnamed graph per TriG document.
- TriG content is stored in files with an '.trig' suffix. The MIME type of TriG is application/trig and the content encoding is UTF-8.

This example contains a default graph and two named graphs.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

# default graph is http://marklogic.com/semantics#default-graph
    {
    <http://example.org/joe> dc:publisher "Joe" .
    <http://example.org/jane> dc:publisher "Jane" .
    }
# first named graph
    <http://example.org/joe>
    {
      _:a foaf:name "Joe" .
      _:a foaf:mbox <mailto:joe@jbloggs.example.org> .
    }
# second named graph
  <http://example.org/jane>
    {
      _:a foaf:name "Jane" .
      _:a foaf:mbox <mailto:jane@jdoe.example.org> .
    }
```

3.2.3 Loading Triples with mlcp

MarkLogic Content Pump (mlcp) is a command line tool for importing into, exporting from, and copying content to MarkLogic from a local file system or Hadoop distributed file system (HDFS).

Using mlcp, you can bulk load billions of triples and quads into a MarkLogic database and specify options for the import. For example, you can specify the directory into which the triples or quads are loaded. It is the recommended tool for bulk loading triples. For more detailed information about mlcp, see Loading Content Using MarkLogic Content Pump in the *Loading Content Into MarkLogic Server Guide*.

This section discusses loading triples into MarkLogic Server with mlcp and includes the following topics:

- Preparation
- Import Command Syntax
- Loading Triples and Quads
- Specifying Collections and a Directory

3.2.3.1 Preparation

Use these procedures to load content with mlcp:

1. Download and extract the mlcp binary files from developer.marklogic.com. Be sure that you have the latest version of mlcp. For more information about installing and using mlcp and system requirements, see Installation and Configuration in the *mlcp User Guide*.

   Note: Although the extracted mlcp binary files do not need to be on the same MarkLogic host machine, you must have access and permissions for the host machine into which you are loading the triples.

2. For these examples we will use the default database (Documents) and forest (Documents). To create your own database see Creating a New Database in the *Administrator's Guide*.

3. Verify that the triple index is enabled by checking the Documents database configuration page of the Admin Interface, or using the Admin API. See "Enabling the Triple Index" on page 66 for details.

   Note: The collection lexicon index is required for the Graph Store HTTP Protocol used by REST API instances and for use of the GRAPH "?g" construct in SPARQL queries. See "Configuring the Database to Work with Triples" on page 27 for information on the collection lexicon.

4. You can use mlcp with the default server on port 8000, which includes an XDBC server. To create your own XDBC server, see Creating a New XDBC Server in the *Administrator's Guide*.

5. (Optional) Put the mlcp bin directory in your path. For example:

   ```
   $ export PATH=${PATH}:/space/marklogic/directory-name/bin
   ```

   where *directory-name* is derived from the version of mlcp that you downloaded.

6. Use a command-line interpreter or interface to enter the import command as a single-line command.

3.2.3.2 Import Command Syntax

The mlcp import command syntax required for loading triples and quads into MarkLogic is:

```
mlcp_command import -host hostname -port port number \
-username username -password password
-output_graph graphname\
-input_file_path filepath -input_file_type filetype
```

Note: Long command lines in this section are broken into multiple lines using the line continuation characters "\" or "^". Remove the line continuation characters when you use the import command.

The *mlcp_command* you use depends on your environment. Use the mlcp shell script `mclp.sh` for UNIX systems and the batch script `mlcp.bat` for Windows systems. The `-host` and `-port` values specify the MarkLogic host machine into which you are loading the triples. Your user credentials, `-username` and `-password` are followed by the path to the content , the `-input_file_path` value. If you use your own database, be sure to add the `-database` parameter for your database. If no database parameter is specified, the content will be put into the default Documents database.

The `-input_file_path` may point to a directory, file, or compressed file in `.zip` or `.gzip` format. The `-input_file_type` is the type of content to be loaded. For triples, the `-input_file_type` must be RDF.

Note: The file extension of the file found in the `-input_file_path` is used by mlcp to identify the type of content being loaded. The type of RDF serialization is determined by the file extension (`.rdf`, `.ttl`, `.nt`, and so on).

A document with a file extension of `.nq` or `.trig` is identified as quad data, all other file extensions are identified as triple data. For more information about file extensions, see "Supported RDF Triple Formats" on page 38.

Note: You must have sufficient MarkLogic privileges to import to the specified host. See Security Considerations in the *mlcp User Guide*.

3.2.3.3 Loading Triples and Quads

In addition to the required import options, you can specify several input and output options. See "Import Options" on page 47 for more details about these options. For example, you can load triples and quads by specifying RDF as the `-input_file_type` option:

```
$ mlcp.sh import -host localhost -port 8000 -username user \
-password passwd -input_file_path /space/tripledata/example.nt \
-output_graph /my/graph -mode local -input_file_type RDF
```

This example uses the shell script to load triples from a single N-Triples file `example.nt`, from a local file system directory `/space/tripledata` into a MarkLogic host on port `8000`.

On a Windows environment, the command would look like this:

```
> mlcp.bat import -host localhost -port 8000 ^
-username admin -password passwd ^
-input_file_path c:\space\tripledata\example.nt -mode local^
-input_file_type RDF -output_graph /my/graph
```

Note: For clarity, these long command lines are broken into multiple lines using the line continuation characters "\" or "^". Remove the line continuation characters when you use the import command.

When you specify RDF as `-input_file_type` the mlcp RDFReader parses the triples and generates XML documents with `sem:triple` as the root element of the document.

3.2.3.4 Import Options

These options can be used with the `import` command to load triples or quads.

Options	Description
`-input_file_type` *string*	Specifies the input file type. Default: document. For triples, use RDF.
`-input_compressed` *boolean*	When set to "true" this option enables decompression on import. Default: false
`-fastload` *boolean*	When set to "true" this option forces optimal performance with a direct forest update. This may result in duplicate document IRIs. See Time vs. Correctness: Understanding -fastload Tradeoffs in the *mlcp User Guide*.
`-output_directory`	Specifies the destination database directory in which to create the loaded documents. Using this option enables `-fastload` by default, which can cause duplicate IRIs to be created. See Time vs. Correctness: Understanding -fastload Tradeoffs in the *mlcp User Guide*. Default: `/triplestore`
`-output_graph`	The graph value to assign to quads with no explicit graph specified in the data. Cannot be used with `-output_override_graph`.
`-output_override_graph`	The graph value to assign to every quad, whether a quad is specified in the data or not. Cannot be used with `-output_graph`.

Options	Description
`-output_collections`	Creates a comma-separated list of collections. Default: `http://marklogic.com/semantics#default-graph` If `-output_collections` is used with `-output_graph` and `-output_override_graph`, the collections specified will be added to the documents loaded.
`-database string`	(optional) The name of the destination database. Default: The database associated with the destination App Server identified by `-host` and `-port`.

Note: When you load triples using mlcp, the `-output_permissions` option is ignored - triples (and, under the covers, triples documents) inherit the permissions of the graph that you're loading into.

If `-output_collections` and `-output_override_graph` are set at the same time, a graph document will be created for the graph specified by `-output_override_graph`, and triples documents will be loaded into collections specified by `-output_collections` and `-output_override_graph`.

If `-output_collections` and `-output_graph` are set at the same time, a graph document will be created for the graph specified by `-output_graph` (where there is no explicit graph specified in the data). Quads with no explicit graph specified in the data will be loaded into collections specified by `-output_collections` and the graph specified by `-output_graph`, while those quads that contain explicit graph data will be loaded into the collections specified by `-output_collections` and the graph(s) specified.

You can split large triples documents into smaller documents to parallelize loading with mlcp and load all the files in a directory that you specify with `-input_file_path`.

For more information about import and output options for mlcp, see Import Command Line Options in the *mlcp User Guide*.

For example:

```
# Windows users, see Modifying the Example Commands for Windows

$ mlcp.sh import -host localhost -port 8000 -username user \
-password passwd -input_file_path /space/tripledata \
-mode local -input_file_type RDF -output_graph /my/graph
```

3.2.3.5 Specifying Collections and a Directory

To load triples into a named graph, specify a collection by using the `-output_collections` option.

> **Note:** To create a new graph, you need to have the `sparql-update-user` role. For more information about roles, see Understanding Roles in the *Security Guide*.

For example:

```
# Windows users, see Modifying the Example Commands for Windows

$ mlcp.sh import -host localhost -port 8000 -username user \
-password passwd -input_file_path /space/tripledata \
-mode local -input_file_type RDF -output_graph /my/graph\
-output_collections /my/collection
```

This command puts all the triples in the `tripledata` directory into a named graph and overwrites the graph IRI to `/my/collection`.

> **Note:** Use `-output_collections` and not `-filename_as_collection` to overwrite the default graph IRI.

For triples data, the documents go in the default collection (`http://marklogic.com/semantics#default-graph`) if you do not specify any collections.

For quad data, if you do not specify any collections, the triples are parsed, serialized, and stored in documents with the fourth part of the quad as the collection.

For example with this quad, the fourth part is an IRI that identifies the homepage of the subject.

```
<http://dbpedia.org/resource/London_Heathrow_Airport>
<http://xmlns.com/foaf/0.1/homepage>
<http://www.heathrowairport.com/>
<http://en.wikipedia.org/wiki/London_Heathrow_Airport?oldid=495283228#
absolute-line=26/> .
```

When the quad is loaded into the database, the collection is generated as a named graph, `http://en.wikipedia.org/wiki/London_Heathrow_Airport?oldid=495283228#absolute-line=26`.

> **Note:** If the `-output_collections` import option specifies a named graph, the fourth element of the quad is ignored and the named graph is used.

If you are using a variety of loading methods, consider putting all of the triples documents in a common directory. Since the `sem:rdf-insert` and `sem:rdf-load` functions put triples documents in the `/triplestore` directory, use `-output_uri_prefix /triplestore` to put mlcp-generated triples documents there as well.

For example:

```
$ mlcp.sh import -host localhost -port 8000 -username user \
-password passwd -input_file_path /space/tripledata/example.zip \
-mode local -input_file_type RDF -input_compressed true
-output_collections /my/collection -output_uri_prefix '/triplestore' \
-output_graph /my/graph
```

When you load triples or quads into a specified named graph from a compressed `.zip` or `.gzip` file, mlcp extracts and serializes the content based on the serialization. For example, a compressed file containing Turtle documents (`.ttl`) will be identified and parsed as triples.

When the content is loaded into MarkLogic with mlcp, the triples are parsed as they are ingested as XML documents with a unique IRI. These unique IRIs are random numbers expressed in hexadecimal. This example shows triples loaded with mlcp from the `persondata.ttl` file, with the `-output_uri_prefix` specified as `/triplestore`:

```
/triplestore/d2a0b25bda81bb58-0-10024.xml
/triplestore/d2a0b25bda81bb58-0-12280.xml
/triplestore/d2a0b25bda81bb58-0-13724.xml
/triplestore/d2a0b25bda81bb58-0-14456.xml
```

Carefully consider the method you choose for loading triples. The algorithm for generating the document IRIs with mlcp differs from other loading methods such as loading from a system file directory with `sem:rdf-load`.

For example, loading the same `persondata.ttl` file with `sem:rdf-load` results in IRIs that appear to have no relation to each other:

```
/triplestore/11b53cf4db02080a.xml
/triplestore/19b3a986fcd71a5c.xml
/triplestore/215710576ebe4328.xml
/triplestore/25ec5ded9bfdb7c2.xml
```

When you load triples with `sem:rdf-load`, the triples are bound to the `http://marklogic.com/semantics` prefix in the resulting documents.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<sem:triples xmlns:sem="http://marklogic.com/semantics">
  <sem:triple>
    <sem:subject>http://dbpedia.org/resource/Wayne_Stenehjem
    </sem:subject>
    <sem:predicate>http://purl.org/dc/elements/1.1/description
    </sem:predicate>
    <sem:object datatype="http://www.w3.org/2001/XMLSchema#string"
    xml:lang="en">American politician
    </sem:object>
  </sem:triple>
  <sem:triple>
    <sem:subject>http://dbpedia.org/resource/Wayne_Stenehjem
    </sem:subject>
    <sem:predicate>http://dbpedia.org/ontology/birthDate
    </sem:predicate>
    <sem:object datatype="http://www.w3.org/2001/XMLSchema#date">
    1953-02-05
    </sem:object>
    </sem:triple>
  </sem:triples>
```

Note: You can leave out the `sem:triples` tag, but you cannot leave out the `sem:triple` tags.

3.2.4 Loading Triples with XQuery

Triples are typically created outside MarkLogic Server and loaded via Query Console by using the following `sem:` functions:

- sem:rdf-insert
- sem:rdf-load
- sem:rdf-get

The `sem:rdf-insert` and `sem:rdf-load` functions are update functions. The `sem:rdf-get` function is a return function that loads triples in memory. These functions are included in the XQuery Semantics API that is implemented as an XQuery library module.

To use `sem:` functions in XQuery, import the module with the following XQuery prolog statement in Query Console:

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

Note: If this module is already imported, you will get an error message.

For more details about semantic functions in XQuery, see the Semantics (`sem:`) documentation in the *MarkLogic XQuery and XSLT Function Reference*.

3.2.4.1 sem:rdf-insert

The `sem:rdf-insert` function inserts triples into the database as triples documents. The triple is created in-memory by using the `sem:triple` and `sem:iri` constructors. The IRIs of the inserted documents are returned on execution.

For example:-*

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

  sem:rdf-insert(
    sem:triple(
      sem:iri("http://example.org/people#m"),
      sem:iri("http://example.com/person#firstName"),
      "Michael"))
```

This returns the document IRI:

```
/triplestore/70eb0b7139816fe3.xml
```

By default, `sem:rdf-insert` puts the documents into the directory `/triplestore/` and assigns the default graph. You can specify a named graph as a collection in the fourth parameter.

For example:

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

  sem:rdf-insert(sem:triple(
        sem:iri("http://example.com/ns/directory#jp"),
        sem:iri("http://example.com/ns/person#firstName"),
        "John-Paul"), null, null,  "mygraph")
```

When you run this example, the document is inserted into both the default graph and mygraph.



Note: If you insert quads or triples in TriG serialization, the graph name comes from the value in the “fourth position” in the quads/trig file.

3.2.4.2 sem:rdf-load

The `sem:rdf-load` function loads and parses triples from files in a specified location into the database and returns the IRIs of the triples documents. You can specify the serialization of the triples, such as `turtle` for Turtle files or `rdfxml` for RDF files.

For example:

```
sem:rdf-load('C:\rdfdata\example.rdf', "rdfxml")

=>
/triplestore/fbd28af1471b39e9.xml
```

As with `sem:rdf-insert`, this function also puts the triples documents into the default graph and `/triplestore/` directory unless a directory or named graph is specified in the options. This example specifies `mynewgraph` as a named graph in the parameters:

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

   sem:rdf-load("C:\turtledata\example.ttl", "turtle", (), (),
        "mynewgraph")
```

The document is inserted:

/triplestore/91b14a8e61b07c11.xml	E sem:triples	(no properties)	http://marklogic.com/semantics#default-graph, mygraph
/triplestore/9f2cfe9ecf2f87c9.xml	E sem:triples	(no properties)	http://marklogic.com/semantics#default-graph, mynewgraph

Note: To use `sem:rdf-load` you need the `xdmp:document-get` privilege.

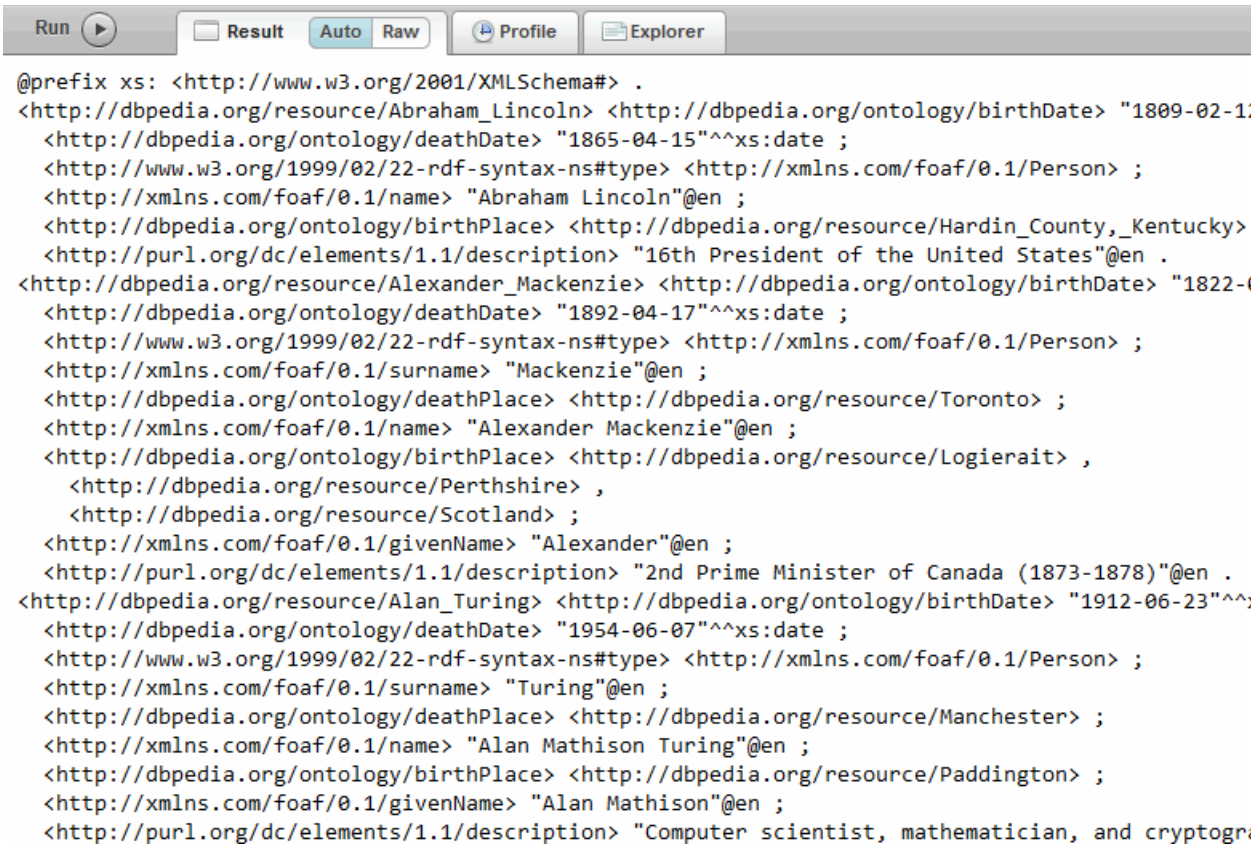
3.2.4.3 sem:rdf-get

The `sem:ref-get` function returns triples in triples files from a specified location. The following example retrieves triples serialized in Turtle serialization from the local filesystem:

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

sem:rdf-get('C:\turtledata\people.ttl', "turtle")
```

The triples are returned as triples in Turtle serialization with one triple per line. Each triple ends with a period.



This Query Console display format allows for easy copying from the Result pane.

3.2.5 Loading Triples with JavaScript

Triples can be loaded via Query Console by using the following `sem.` functions:

- sem.rdfInsert
- sem.rdfLoad
- sem.rdfGet

The `sem.rdfInsert` and `sem.rdfLoad` functions are update functions. The `sem.rdfGet` function is a return function that loads triples in memory. These functions are included in the JavaScript Semantics API.

To use `sem.` functions in JavaScript, import the module with the following JavaScript statements in Query Console:

```
declareUpdate();
const sem = require("/MarkLogic/semantics.xqy");
```

Note: If this module is already imported, you will get an error message.

For more details about semantic functions in JavaScript, see the Semantics (`sem.`) documentation in the *MarkLogic Server-Side JavaScript Function Reference*.

3.2.5.1 sem.rdfInsert

The `sem.rdfInsert` function inserts triples into the database as triples documents. The triple is created in-memory by using the `sem.triple` and `sem.iri` constructors. The IRIs of the inserted documents are returned on execution.

For example:-*

```
declareUpdate();
const sem = require("/MarkLogic/semantics.xqy");

sem.rdfInsert(
  sem.triple(
      sem.iri("http://example.com/ns/directory#m"),
      sem.iri("http://example.com/ns/person#firstName"), "Michael"));
```

This returns the document IRI:

```
/triplestore/74521a908ece2074.xml
```

By default, `sem.rdfInsert` puts the documents into the directory `/triplestore/` and assigns the default graph. You can specify a named graph as a collection in the fourth parameter.

For example:

```
declareUpdate();
const sem = require("/MarkLogic/semantics.xqy");

sem.rdfInsert(
  sem.triple(
      sem.iri("http://example.com/ns/directory#m"),
      sem.iri("http://example.com/ns/person#firstName"),
      "John-Paul"), (), (), "mygraph");
```

When you run this example, the document is inserted into both the default graph and mygraph.

Note: If you insert quads or triples in TriG serialization, the graph name comes from the value in the “fourth position” in the quads/trig file.

3.2.5.2 sem.rdfLoad

The `sem.rdfLoad` function loads and parses triples from files in a specified location into the database and returns the IRIs of the triples documents. You can specify the serialization of the triples, such as `turtle` for Turtle files or `rdfxml` for RDF files.

For example:

```
declareUpdate();
var sem = require("/MarkLogic/semantics.xqy");

sem.rdfLoad('C:/data/example.rdf', "rdfxml")
=>
/triplestore/fbd28af1471b39e9.xml
```

As with `sem.rdf-Insert`, this function also puts the triples documents into the default graph and `/triplestore/` directory unless a directory or named graph is specified in the options. This example specifies `mynewgraph` as a named graph in the parameters:

```
declareUpdate();
var sem = require("/MarkLogic/semantics.xqy");

  sem.rdfLoad('C:/turtledata/example.ttl', "turtle", (), (),
        "mynewgraph"))
=>
/triplestore/fbd28af1471b39e9.xml
```

The document is inserted.

> **Note:** To use `sem.rdfLoad` you need the `xdmp.documentGet` privilege.

3.2.5.3 sem.rdfGet

The `sem.refGet` function returns triples in triples files from a specified location. The following example retrieves triples serialized in Turtle serialization from the local filesystem:

```
var sem = require("/MarkLogic/semantics.xqy");

sem.rdfGet('C:/turtledata/people.ttl', "turtle");
```

The triples are returned as triples in Turtle serialization with one triple per line. Each triple ends with a period.

3.2.6 Loading Triples Using the REST API

A REST endpoint is an XQuery module on MarkLogic Server that routes and responds to an HTTP request. An HTTP client invokes endpoints to create, read, update, or delete content in MarkLogic. This section discusses using the REST API to load triples with a REST endpoint. It covers the following topics:

- Preparation
- Addressing the Graph Store
- Specifying Parameters
- Supported Verbs
- Supported Media Formats
- Loading Triples
- Response Errors

3.2.6.1 Preparation

If you are unfamiliar with the REST API and endpoints, see Introduction to the MarkLogic REST API in the *REST Application Developer's Guide*.

Use the following procedures to make requests with REST endpoints:

1. Install MarkLogic Server version 8.0-4 or later.

2. Install `curl` or an equivalent command line tool for issuing HTTP requests.

3. You can use the default database and forest (Documents) on port 8000 or create your own. To create a new database and forest, see Creating a New Database in the *Administrator's Guide*.

4. Verify that the triple index and the collection lexicon are enabled on the Documents database by checking the configuration page of the Admin Interface or by using the Admin API. See "Enabling the Triple Index" on page 66.

   Note: The collection lexicon is required for the Graph Store HTTP Protocol of REST API instances.

5. You can use the default REST API instance associated with port 8000. If you want to create a new REST API instance, see Creating an Instance in the *REST Application Developer's Guide.*

3.2.6.2 Addressing the Graph Store

The graph endpoint is an implementation of the W3C Graph Store HTTP Protocol as specified in the SPARQL 1.1 Graph Store HTTP Protocol:

http://www.w3.org/TR/2013/REC-sparql11-http-rdf-update-20130321/

The base URL for the graph store is:

```
http://hostname:port/vversion/graphs
```

Where *hostname* is the MarkLogic Server host machine, *port* is the port on which the REST API instance is running, and *version* is the version number of the API. The Graph Store HTTP Protocol is a mapping from RESTful HTTP requests to the corresponding SPARQL 1.1 Update operations. See Summary of the /graphs Service in the *REST Application Developer's Guide*.

3.2.6.3 Specifying Parameters

The graph endpoint accepts an optional parameter for a particular named graph. For example:

```
http://localhost:8000/v1/graphs?graph=http://named-graph
```

If omitted, the default graph must be specified as a default parameter with no value.

For example:

```
http://localhost:8000/v1/graphs?default
```

When a `GET` request is issued with no parameters, the list of graphs will be given in list format. See `GET:/v1/graphs` for more details.

3.2.6.4 Supported Verbs

A REST client uses HTTP verbs such as `GET` and `PUT` to interact with MarkLogic Server. This table lists the supported verbs and the role required to use each:

Verb	Description	Role
`GET`	Retrieves a named graph.	rest-reader
`POST`	Merges triples into a named graph or adds triples to an empty graph.	rest-writer
`PUT`	Replaces triples in a named graph or adds triples to an empty graph. Functionally equivalent to DELETE followed by `POST`. For an example, see "Loading Triples" on page 59.	rest-writer
`DELETE`	Removes triples in a named graph.	rest-writer
`HEAD`	Test for the existence of a graph. Retrieves a named graph, without the body.	rest-reader

The role you use to make a MarkLogic REST API request must have appropriate privileges for the content accessed by the HTTP call; for example, permission to read or update documents in the target database. For more information about REST API roles and privileges, see Security Requirements in the *REST Application Developer's Guide*.

Note: This endpoint will only update documents with the element `sem:triple` as the root.

3.2.6.5 Supported Media Formats

For a list of supported media formats for the `Content-type` HTTP header, see "Supported RDF Triple Formats" on page 38.

3.2.6.6 Loading Triples

To insert triples, make a `PUT` or `POST` request to a URL of the form:

```
http://host:port/v1/graphs?graph=graphname
```

When constructing the request:

1. Specify the graph in which to load the triples.
   - To specify the default graph, set the `graph` parameter to the default graph.
   - To specify a named graph, set the `graph` parameter to the named graph.
2. Place the content in the request body.
3. Specify the MIME type of the content in the `Content-type` HTTP header. See "Supported RDF Triple Formats" on page 38.
4. Specify the user credentials.

The triples are loaded into the default directory, `/triplestore`.

This is an example of a `curl` command for a UNIX or Cygwin command line interpreter. The command sends a `PUT` HTTP request to insert the contents of the file `example.nt` into the database as XML documents in the default graph:

```
# Windows users, see  Modifying the Example Commands for Windows

$ curl -s -X PUT --data-binary '@example.nt' \
    -H "Content-type: application/n-triples" \
    --digest --user "admin:password" \
    "http://localhost:8000/v1/graphs?default"
```

Note: When you load triples with the REST endpoint using `PUT` or `POST`, you must specify the default graph or a named graph.

These `curl` command options are used in the preceding example:

Option	Description
`-s`	Specifies silent mode, so that the `curl` output does not include the HTTP response headers in the output. The alternative is `-i` to include the response headers.
`-X` *`http_method`*	The type of HTTP request (`PUT`) that `curl` will send. Other supported requests are `GET`, `POST` and `DELETE`. See “Supported Verbs” on page 58.
`--data-binary` *`data`*	Data to include in the request body. Data may be placed directly on the command line as an argument to `--data-binary`, or read from a file by using `@`*`filename`*. If you are using Windows, a Windows version of `curl` that supports the "`@`" operator is required.
`-H` *`headers`*	The HTTP header to include in the request. The examples in this guide use `Content-type`.
`--digest`	The authentication method specified encrypts the user’s password.
`--user` *`user:password`*	Username and password used to authenticate the request. Use a MarkLogic Server user that has sufficient privileges to carry out the requested operation. For details, see Security Requirements in the *REST Application Developer’s Guide*.

For more information about the REST API, see the Semantics documentation in the *REST Client API*. For more about REST and Semantics see “Using Semantics with the REST Client API” on page 189.

3.2.6.7 Response Errors

This section covers the error reporting conventions followed by the MarkLogic REST API.

If a request to a MarkLogic REST API Instance fails, an error response code is returned and additional information is detailed in the response body.

These response errors may be returned:

- `400 Bad Request` returns for `PUT` or `POST` requests that have no parameters at all.
- `400 Bad Request` returns for `PUT` or `POST` requests for payloads that fails to parse.
- `404 Not Found` returns for `GET` requests to a graph that does not exist (the IRI is not present in the collection lexicon).
- `406 Not Acceptable` returns for `GET` requests for triples in an unsupported serialization.

- `415 Unsupported Media Type` returns for `POST` or `PUT` request in an unsupported format.

  Note: The `repair` parameter for `POST` and `PUT` requests can be set to `true` or `false`. By default this is `false`. If set to `true`, a payload that does not properly parse will still insert any triples that do parse. If set to false, any payload errors whatsoever will result in a `400 Bad Request` response.

3.2.7 Loading Triples Using the Java API

For an example of loading triples using the MarkLogic Java API, see Example: Loading, Managing, and Querying Triples in the *Java Application Developer's Guide*.

3.2.8 Loading Triples Using the Node.js API

For an example of loading triples using the MarkLogic Node.js API, see Loading Triples in the *Node.js Application Developer's Guide*.

4.0 Triple Index Overview

This chapter provides an overview of the triple index in MarkLogic Server and includes the following sections:

- Understanding the Triple Index and How It's Used
- Enabling the Triple Index
- Other Considerations

4.1 Understanding the Triple Index and How It's Used

The triple index is used to index schema-valid `sem:triple` elements found anywhere in a document. The indexing of triples is performed when documents containing triples are ingested into MarkLogic or during a database reindex. The triple index stores each unique value only once, in the dictionary. The dictionary gives each value an ID, and the triple data then uses that ID to reference the value in the dictionary.

The validity of `sem:triple` elements is determined by checking elements and attributes in the documents against the `sem:triple` schema (`/MarkLogic/Config/semantics.xsd`). If the `sem:triple` element is valid, an entry is created in the triple index, otherwise the element is skipped. Unlike range indexes, triple indexes do not have to fit in memory, so there is little up-front memory allocation.

> **Note:** For all new installations of MarkLogic 9 and later, the triple index and collection lexicon are enabled by default. Any new databases will also have the triple index and collection lexicon enabled.

This section covers the following topics:

- Triple Data and Value Caches
- Triple Values and Type Information
- Triple Positions
- Index Files
- Permutations

4.1.1 Triple Data and Value Caches

Internally, MarkLogic stores triples in two ways: *triple values* and *triple data*. The *triple values* are the individual values from every triple, including all typed literal, IRIs, and blank nodes. The *triple data* holds the triples in different permutations, along with a document ID and position information. The triple data refer to the triple values by ID, making for very efficient lookup. Triple data is stored compressed on disk, and triple values are stored in a separate compressed value store. Both the triple index and the value store are stored in compressed four-kilobyte (4k) blocks.

When triple data is needed (for example, during a lookup), the relevant block is cached in either the triple cache or the triple value cache. Unlike other MarkLogic caches, the triple cache and triple value cache shrinks and grows, only taking up memory when it needs to add to the caches.

Note: You can configure the size of the triple cache and the triple value cache for the host of your triple store, as described in “Sizing Caches” on page 69.

4.1.1.1 Triple Cache and Triple Value Cache

The triple cache holds blocks of compressed triples from disk which are flushed using a least recently used (LRU) algorithm. Blocks in the triple cache refer to values from a dictionary. The triple value cache holds uncompressed values from the triple index dictionary. The triple value cache is also an LRU cache.

Triples in the triple index are filtered out depending on the timestamps of the query and of the document from which they came. The triple cache holds information generated before the filtering happens, so deleting a triple has no effect on triple caches. However, after a merge, old stands may be deleted. When a stand is deleted, all its blocks are flushed from the triple caches.

Cache timeout controls how long MarkLogic Server will keep triple index blocks in the cache after the last time it was used (when it has not been flushed to make room for another block). Increasing the cache timeout might be good for keeping the cache hot for queries that are run at infrequent periods. Other more frequent queries may push the information out of the cache before the infrequent query is re-run.

4.1.2 Triple Values and Type Information

Values are stored in a separate value store on disk in “value equality” sorted order, so in a given stand, the value ID order is equivalent to value equality order.

Strings in the values are stored in the range index string storage. Anything not relevant to value equality is removed from the stored values, such as timezone and derived type information.

Since type information is stored separately, triples can be returned directly from the triple index. This information is also used for RDF-specific “sameTerm” comparison required by SPARQL simple entailment.

4.1.3 Triple Positions

The triple positions index is used to accurately resolve queries that use `cts:triple-range-query` and the `item-frequency` option of `cts:triples`. The triple positions index is also used to accurately resolve searches that use the `cts:near-query` and `cts:element-query` constructors. The triple positions index stores locations within a fragment of the relative positions of triples within that fragment (typically, a fragment is a document). Enabling the triple positions index increases index sizes and somewhat slows document loads, but it increases the accuracy of queries that need those positions.

For example:

```
xquery version "1.0-ml";

cts:search(doc(),
  cts:near-query((
  cts:triple-range-query(sem:iri("http://www.rdfabout.com/rdf/
    usgov/sec/id/cik0001075285"), (), ()),

  cts:triple-range-query(sem:iri("http://www.rdfabout.com/rdf/
    usgov/sec/id/cik0001317036"), (), ())
    ),11), "unfiltered")
```

The `cts:near-query` returns a sequence of queries to match, where the matches occur within the specified distance from each other. The distance specified is in the number of words between any two matching queries.

The unfiltered search selects fragments from the indexes that are candidates to satisfy the specified `cts:query` and returns the document.

4.1.4 Index Files

To efficiently make use of memory, the index files for triple and value stores are directly mapped into memory. The type store is entirely mapped into memory.

Both the triple and value stores have index files consisting of 64-byte segments. The first segment in each is a header containing checksums, version number, and counts (of triples or values). This is followed by:

- Triples index: After the header segment, the triples index contains an index of the first two values and permutation of the first triple in each block, arranged into 64-byte segments. This is used to find the blocks needed to answer a given lookup based on values from the triple. Currently triples are not accessed by ordinal, so an ordinal index is not required.
- Values Index: After the header segment, the values index contains an index of the first value in each block, arranged into 64-byte segments. The values index is used to find the blocks needed to answer a given lookup based on value. This is followed by an index of the starting ordinal for each block, which is used to find the block needed to answer a given lookup based on a value ID.

  Note: The triple index stores positions if the `triple positions` is enabled. See “Enabling the Triple Index” on page 66.

The type store has an index file that stores the offset into the type data file for each stored type. This is also mapped into memory.

This table describes the memory-mapped index files that store information used by the triple indexes and values stores.

Index File	Description
`TripleIndex` `TripleValueIndex`	Block indexes for TripleData and TripleValueData
`TripleTypeData` `TripleTypeIndex`	Type information for the triple values

`StringData`
`StringIndex`
`AtomData`
`AtomIndex`
	Also used by the string-based range indexes

`TripleValueFreqs` `TripleValueFreqsIndex`	Statistical information about the triples. The triple index keeps statistics on the triples for every value kept in the database.

4.1.5 Permutations

The permutation enumeration details the role each value plays in the original triple. Three permutations are stored in order to provide access to different sort orders, and to be able to efficiently look up different parts of the triple. The permutations are acronyms made up from the initials of the three RDF elements (subject, predicate, and object), for example:{ `SOP, PSO, OPS` }.

Use the `cts:triples` function to specify one of these sort orders in the options:

- `order-pso` - Returns results ordered by predicate, then subject, then object
- `order-sop` - Returns results ordered by subject, then object, then predicate
- `order-ops` - Returns results ordered by object, then predicate, then subject

4.2 Enabling the Triple Index

By default, the triple index is enabled for databases in MarkLogic 9 or later. This section discusses how to enable the triple index or verify that it is enabled. It also discusses related indexes and configuration settings. It includes the following topics:

- Using the Database Configuration Pages
- Using the Admin API

4.2.1 Using the Database Configuration Pages

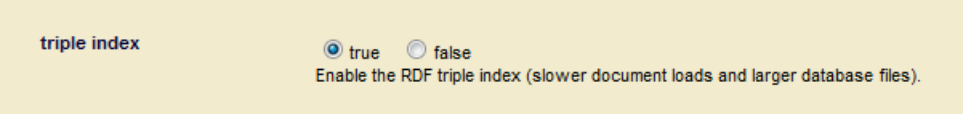
The triple index can be enabled or disabled on the Admin Interface (`http://hostname:8001`) database configuration page. The *hostname* is the MarkLogic Server host for which the triple index is to be enabled.

For more information about index settings, see Index Settings that Affect Documents of the *Administrator's Guide* and "Configuring the Database to Work with Triples" on page 27.

> **Note:** For all new installations of MarkLogic 9 and later, the triple index is enabled by default. Any new databases will also have the triple index enabled. You may want to verify that existing databases have the triple index enabled.

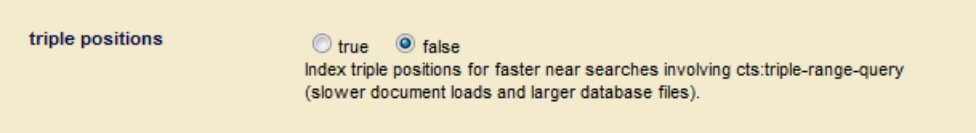
Use the following procedures to verify or configure the triple index and related settings. To enable the triple positions index, the in-memory triple index size, and collection lexicon, use the Admin interface (`http://hostname:8001`) or the Admin API. See "Using the Admin API" on page 68 for details.

- In the Admin Interface, scroll down to the triple index setting and set it to true.



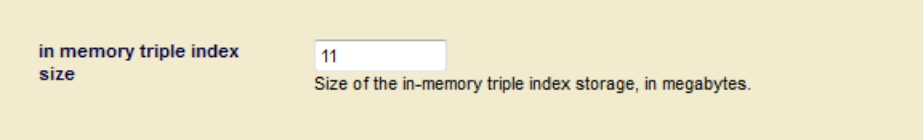
  When you enable the triples index for the first time, or if you are reindexing your database after enabling the triple index, only documents containing valid `sem:triple` elements are indexed.

- You can enable the triple positions index for faster near searches using `cts:triple-range-query`.



  It is not necessary to enable the triple position index for querying with native SPARQL.

- You can set the size of cache and buffer memory to be allocated for managing triple index data for an in-memory stand.



> **Note:** When you change any index settings for a database, the new settings will take effect based on whether reindexing is enabled (`reindexer enable` set to `true`).

4.2.2 Using the Admin API

Use these Admin API functions to enable the triple index, triple index positions, and configure the in-memory triple index size for your database:

- `admin:database-set-triple-index`
- `admin:database-set-triple-positions`
- `admin:database-set-in-memory-triple-index-size`

This example sets the triple index of "Sample-Database" to `true` using the Admin API:

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

(: Get the configuration :)
let $config := admin:get-configuration()

(: Obtain the database ID of 'Sample-Database' :)
let $Sample-Database := admin:database-get-id(
  $config, "Sample-Database")
let $c := admin:database-set-triple-index($config, $Sample-Database,
fn:true())
return admin:save-configuration($c)
```

This example uses the Admin API to set the triple positions of the database to `true`:

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

let $config := admin:get-configuration()
let $Sample-Database := admin:database-get-id(
  $config, "Sample-Database")
let $c := admin:database-set-triple-positions($config,
  $Sample-Database, fn:true())
return admin:save-configuration($c)
```

This example sets the in-memory triple index size of the database to 256MB:

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

let $config := admin:get-configuration()
let $Sample-Database := admin:database-get-id(
  $config, "Sample-Database")
let $c := admin:database-set-in-memory-triple-index-size($config,
  $Sample-Database, 256)
return admin:save-configuration($c)
```

Note: For details about the function signatures and descriptions, see the `admin:database` functions (database) in the *XQuery and XSLT Reference Guide*.

4.3 Other Considerations

This section includes the following topics:

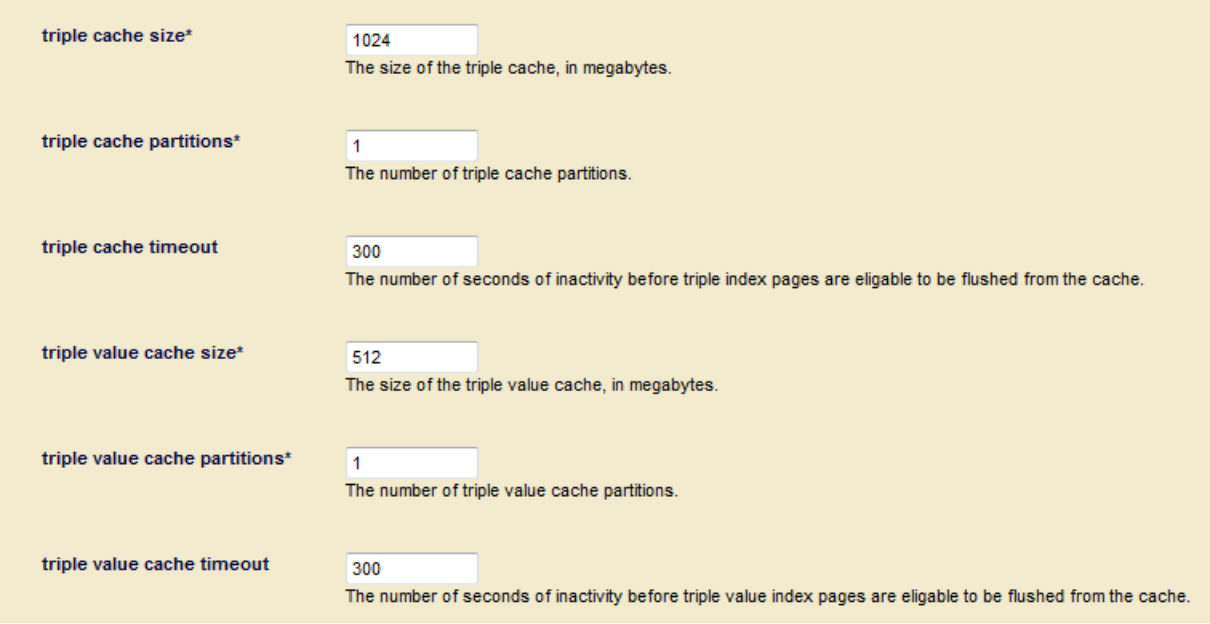
- Sizing Caches
- Unused Values and Types
- Scaling and Monitoring

4.3.1 Sizing Caches

The triple cache and the triple value cache are d-node caches, which are partitioned for lock contention. This partitioning enables parallelism and speeds up processing.

The maximum sizes of the caches and number of partitions are configurable. To change the triple or triple value cache sizes for the host, you can use the Groups configuration page in the Admin Interface or use the Admin API.

In the Admin Interface (`http://hostname:8001`) on the Groups configuration page, specify values for caches sizes, partitions, and timeouts:



This table describes the Admin API functions for group cache configurations:

Function	Description
`admin:group-set-triple-cache-size`	Changes the triple cache size setting of the group with the specified ID to the specified value
`admin:group-set-triple-cache-partitions`	Changes the triple cache partitions setting of the group with the specified ID to the specified value
`admin:group-set-triple-cache-timeout`	Changes the number of seconds a triple block can be unused before being flushed from caches
`admin:group-set-triple-value-cache-timeout`	Changes the number of seconds a triple value block can be unused before being flushed from caches
`admin:group-set-triple-value-cache-size`	Changes the triple value cache size setting of the group with the specified ID to the specified value
`admin:group-set-triple-value-cache-partitions`	Changes the triple value cache partitions setting of the group with the specified ID to the specified value

4.3.2 Unused Values and Types

During a merge, triple values and types may become unused by the triple index. To merge the triple index in a single streaming pass, type and value stores are merged before the triples. Unused values and types are identified during the merge of the triples. During the next merge, the unused types and values identified are be removed, releasing the space they previously used.

> **Note:** For best compaction, two merges are needed. This is not an issue in normal operations because MarkLogic Server is designed to periodically merge.

Since the type store is ordered by frequency, it is merged entirely in memory. The value and triple stores are merged in a streaming fashion, from and to disk directly.

For more information about merging, see Understanding and Controlling Database Merges in the *Administrator's Guide*.

4.3.3 Scaling and Monitoring

Since SPARQL execution does not fetch fragments, there is the potential to scale back on expanded and compressed tree caches on triple-only deployments. You can configure tree caches from the Group configuration page in the Admin Interface, or by using these functions:

```
admin:group-set-expanded-tree-cache-size
admin:group-set-compressed-tree-cache-size
```

You can monitor the status of the database and forest from the database Status page in the Admin Interface:

```
http://hostname:8001/
```

You can also use the MarkLogic monitoring tools, Monitoring Dashboard and Monitoring History:

```
http://hostname:8002/dashboard
http://hostname:8002/history
```

For more information, see Using the MarkLogic Server Monitoring Dashboard in the *Monitoring MarkLogic Guide*.

You can also use these functions for query metrics and to monitor the status of forests and caches:

- `xdmp:query-meters` - Cache hits or misses for a query
- `xdmp:forest-status` - Cache hits or misses, hit rate, and miss rate for each stand
- `xdmp:cache-status` - Percentage busy, used, and free by cache partition

5.0 Unmanaged Triples

Triples that included as part of an XML or a JSON document and have an element node of `sem:triple` are called unmanaged triples, sometimes referred to as *embedded triples*. These unmanaged triples must be in the MarkLogic XML or JSON format defined in the schema for `sem:triple` (`semantics.xsd`).

> **Note:** Unmanaged triples cannot be modified with SPARQL Update. Use XQuery or JavaScript to modify these triples. See “Updating Triples” on page 239 for more details.

With unmanaged triples, MarkLogic works like a triple store and a document store. You have the functionality of a triple store and a document store for your data.

This example inserts an unmanaged triple into an XML document (`Article.xml`):

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

xdmp:document-insert("Article.xml",
<article>
  <info>
   <title>News for April 9, 2013</title>
    <sem:triples xmlns:sem="http://marklogic.com/semantics">
      <sem:triple>
       <sem:subject>http://example.com/article</sem:subject>
       <sem:predicate>http://example.com/mentions</sem:predicate>
       <sem:object
datatype="http://www.w3.org/2001/XMLSchema#string">London</sem:object>
      </sem:triple>
    </sem:triples>
  </info>
</article>)
```

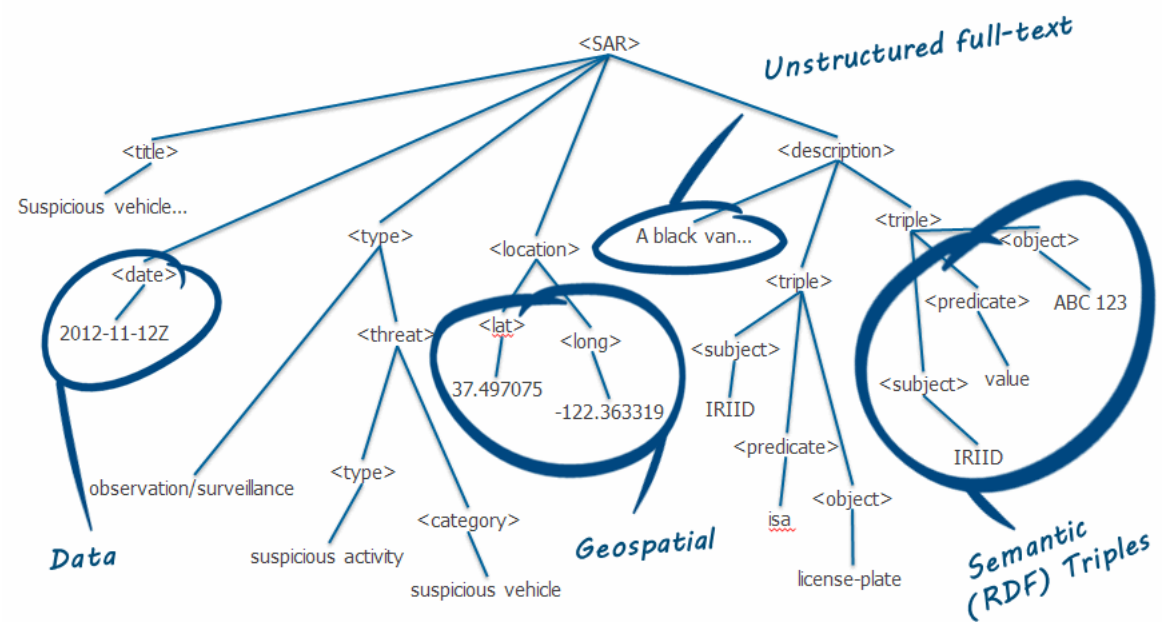
> **Note:** You can leave out the `sem:triples` tag, but you cannot leave out the `sem:triple` tags.

An XML or JSON document can contain many kinds of information, along with the triples.

This example shows a suspicious activity report document that contains both XML and triples:

```
<SAR>
  <title>Suspicious vehicle...Suspicious vehicle near airport</title>
  <date>2014-11-12Z</date>
  <type>observation/surveillance</type>
  <threat>
    <type>suspicious activity</type>
    <category>suspicious vehicle</category>
  </threat>
  <location>
    <lat>37.497075</lat>
    <long>-122.363319</long>
  </location>
  <description>A blue van with license plate ABC 123 was observed
parked behind the airport sign...
    <sem:triple>
      <sem:subject>IRIID</sem:subject>
      <sem:predicate>isa</sem:predicate>
      <sem:object
datatype="http://www.w3.org/2001/XMLSchema#string">license-
plate</sem:object>
    </sem:triple>
    <sem:triple>
      <sem:subject>IRIID</sem:subject>
      <sem:predicate>value</sem:predicate>
      <sem:object
datatype="http://www.w3.org/2001/XMLSchema#string">ABC
123</sem:object>
    </sem:triple>
  </description>
</SAR>
```

Unmanaged triples ingested into a MarkLogic database are indexed by the triple index and stored for access and query by SPARQL. Here is another representation of the same information:



You can also embed triples into JSON documents. Here is how you would insert a triple using JavaScript:

```
declareUpdate();
var sem = require("/MarkLogic/semantics.xqy");
xdmp.documentInsert(
 "testDoc.json", {
  "my": "data","triple":{
    "subject": "http://example.org/ns/dir/js/",
    "predicate": "http://xmlns.com/foaf/0.1/firstname/",
    "object": {"datatype" : "http://www.w3.org/2001/XMLSchema#string",
     "value": "John"
      }
    }
  }
)
```

Here is the triple embedded in a JSON document:

```
{
"my": "data",
"triple":{
  "subject": "http://example.org/ns/dir/js/",
  "predicate": "http://xmlns.com/foaf/0.1/firstname/",
  "object": {
    "datatype" : "http://www.w3.org/2001/XMLSchema#string",
      "value":"John"
    }
  }
}
```

You can do the same document insert with XQuery:

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

xdmp:document-insert("myData.xml",
 <sem:triples xmlns:sem="http://marklogic.com/semantics">
   <sem:triple>
    <sem:subject>http://example.org/ns/dir/js/</sem:subject>
    <sem:predicate>http://xmlns.com/foaf/0.1/firstname/</sem:predicate
>
    <sem:object
datatype="http://www.w3.org/2001/XMLSchema#string">John</sem:object>
   </sem:triple>
  </sem:triples>
)
```

When triples are embedded in an XML or JSON document as unmanaged triples, they can include additional information about the triple along with additional metadata (time/date information, bitemporal information, source of the triple). You can add useful information to the XML or JSON file (like the provenance of the triple). When you update the triple, you update the document and the triple together.

In addition to adding triples to a document, you can use a template to identify content to be indexed as triples. See “Using a Template to Identify Triples in a Document” on page 247 for more information about templates.

5.1 Uses for Triples in XML Documents

With unmanaged triples you can do combination queries on both the document and the triples they contain. The triples stay “in context” with the other information in the document in which they are embedded and have the security and permissions associated with that document. These triples are updated with the document and deleted when the document is deleted.

5.1.1 Context from the Document

When you have triples in a document, the document can provide context for the data described by the triples. The source of the triples and more information about when the document and triples were created can be included as part of the document.

```
<article>
 <info>AP Newswire - Nixon went to China</info>
 <triples-context>
  <confidence>80</confidence>
  <pub-date>2011-10-14</pub-date>
  <source>AP Newswire</source>
 </triples-context>
 <sem:triple xmlns:sem="http://marklogic.com/semantics">
  <sem:subject>http://example.org/news/Nixon</sem:subject>
  <sem:predicate>http://example.org/wentTo</sem:predicate>
```

```
  <sem:object
datatype="http://www.w3.org/2001/XMLSchema#string">China</sem:object>
 </sem:triple>
</article>
```

You can annotate the triples to provide even more information, such as the level of confidence in the reliability of the information.

5.1.2 Combination Queries

A combination query operates on both the document and any triples. Here is a complex query for the information in the AP newswire document :

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

sem:sparql('
  SELECT ?country
  WHERE {
    <http://example.org/news/Nixon> <http://example.org/wentTo>
?country
  }
  ',
  (),
  (),
  cts:and-query( (
    cts:path-range-query( "//triples-context/confidence", ">=", 80) ,
    cts:path-range-query( "//triples-context/pub-date", ">",
xs:date("1974-01-01")),
    cts:or-query( (
      cts:element-value-query( xs:QName("source"), "AP Newswire" ),
      cts:element-value-query( xs:QName("source"), "BBC" )
   ) )
 ) )
)
```

The cts query in this example identifies a set of fragments. Any triples in those fragments are used to build a semantic store and the SPARQL query is then run against that store. This means that the query says, “Find countries in triples that are in fragments identified by the cts query; which is any fragment that has a `sem:triple/@confidence > 80` and a `sem:triple/@date` earlier than 1974, and has a source element with either “`AP Newswire`” or “`BBC`”.

5.1.3 Security

For unmanaged triples, the security permissions for the document also apply to the triples. You will need to have the appropriate permissions to modify or add triples to the document. To find the current permissions for a document, use `xdmp:document-get-permissions`:

```
xquery version "1.0-ml";
xdmp:document-get-permissions("/example.json")

=>
<sec:permission xmlns:sec="http://marklogic.com/xdmp/security">
  <sec:capability>read</sec:capability>
  <sec:role-id>11180836995942796002</sec:role-id>
</sec:permission>
<sec:permission xmlns:sec="http://marklogic.com/xdmp/security">
  <sec:capability>update</sec:capability>
  <sec:role-id>11180836995942796002</sec:role-id>
</sec:permission>
```

To set the permssions on a document, you can use `xdmp:document-set-permissions`:

```
xdmp:document-set-permissions(
  "/example.json",
  (xdmp:permission("sparql-update-user","update"),
  xdmp:permission("sparql-update-user","read"))
)
```

See Document Permissions in the *Security Guide* for more information about document permissions.

5.2 Bitemporal Triples

You can use SPARQL to perform bitemporal search queries with unmanaged triples. In this example, the bitemporal query is wrapped inside the SPARQL query as a `cts:period-range-query`.

```
let $q := '
SELECT
  ?derivation
WHERE {
  <http://example.com/prov/trader/>
   <http://www.w3.org/ns/prov#wasDerivedFrom/> ?derivation
  }
  '
return
  sem:sparql(
    $q,
    (),
    (),
    sem:store(
      (),
      cts:period-range-query(
```

```
            "valid",
            "ISO_CONTAINS",
          cts:period(
            xs:dateTime("2014-04-01T16:10:00"),
            xs:dateTime("2014-04-01T16:12:00")) )
        )
      )
```

This bitemporal SPARQL query searches for events between `2014-04-01T16:10:00` and `2014-04-01T16:12:00`. See Understanding Temporal Documents in the *Temporal Developer's Guide* for more infomation about temporal documents.

6.0 Semantic Queries

This chapter discusses the principal techniques and tools used for performing semantic queries on RDF triples. Just as with loading and deleting triples, you can select your preferred method for querying RDF triples in MarkLogic. You can query triples in several ways, though the main focus in this chapter is using SPARQL to query triples.

MarkLogic supports the syntax and capabilities in SPARQL 1.1. SPARQL is a query language specification for querying over RDF triples. The SPARQL language is a formal W3C recommendation from the RDF Data Access Working Group. It is described in the SPARQL Query Language for RDF recommendation:

http://www.w3.org/TR/rdf-sparql-query/

SPARQL queries are executed natively in MarkLogic to query either in-memory triples or triples stored in a database. When querying triples stored in a database, SPARQL queries execute entirely against the triple index. For examples of running SPARQL queries, see “Querying Triples” on page 32.

You can combine SPARQL with XQuery or JavaScript. For example, you can restrict a SPARQL query by passing in a `cts:query` (XQuery) or `cts.query` (JavaScript) and you can call built-in functions (including `cts:contains` or `cts.contains` for full-text search) as part of your SPARQL query. For more details, see “Using Built-in Functions in a SPARQL Query” on page 104.

You can use the following methods to query triples:

- SPARQL mode in Query Console. For details, see “Querying Triples with SPARQL” on page 82
- XQuery using the semantics functions, and Search API, or a combination of XQuery and SPARQL. For details, see “Querying Triples with XQuery or JavaScript” on page 128.
- HTTP via a SPARQL endpoint. For details, see “Using Semantics with the REST Client API” on page 189.

  Note: SPARQL keywords are shown in uppercase in this chapter, however SPARQL keywords are not case sensitive.

This chapter includes the following sections:

- Querying Triples with SPARQL
- Querying Triples with XQuery or JavaScript
- Querying Triples with the Optic API
- Serialization
- Security

6.1 Querying Triples with SPARQL

This section is a high-level overview of the SPARQL query capabilities in MarkLogic and includes the following topics:

- Types of SPARQL Queries
- Executing a SPARQL Query in Query Console
- Specifying Query Result Options
- Selecting Results Rendering
- Constructing a SPARQL Query
- Prefix Declaration
- Query Pattern
- Target RDF Graph
- Result Clauses
- Query Clauses
- Solution Modifiers
- Property Path Expressions
- SPARQL Aggregates
- SPARQL Resources

Note: The examples in this section use the `persondata-en.ttl` dataset from http://downloads.dbpedia.org/3.8/en/persondata_en.ttl.bz2. See “Downloading the Dataset” on page 29.

6.1.1 Types of SPARQL Queries

You can query an RDF dataset using any of these SPARQL query forms:

- SELECT Queries - A SPARQL `SELECT` query returns a *solution*, which is a set of bindings of variables and values.
- CONSTRUCT Queries - A SPARQL `CONSTRUCT` query returns *triples* as a sequence of `sem:triple` values in an RDF graph. These triples are constructed by substituting variables in a set of triple templates to create new triples from existing triples.
- DESCRIBE Queries - A SPARQL `DESCRIBE` query returns a sequence of `sem:triple` values as an RDF graph that describes the resources found.
- ASK Queries - A SPARQL `ASK` query returns a boolean (`true` or `false`) indicating whether a query pattern matches the dataset.

6.1.2 Executing a SPARQL Query in Query Console

To execute a SPARQL query:

1. In a Web browser, navigate to the Query Console:

   ```
   http://hostname:8000/qconsole
   ```

   where *hostname* is the name of your MarkLogic Server host.

2. From the Query Type drop-down list, select SPARQL Query.

   The Query Console supports syntax highlighting for SPARQL keywords.

   Note: Select SPARQL Update when you are working with SPARQL Update. See "SPARQL Update" on page 169 for more information.

3. Construct your SPARQL query. See "Constructing a SPARQL Query" on page 87.

   You can add comments prefaced with the hash symbol (#).

4. From the Content Source drop-down list, select the target database.

5. In the control bar below the query window, click Run.

   Note: If the triple index is not enabled for the target database, an XDMP-TRPLIDXNOTFOUND exception is thrown. See "Enabling the Triple Index" on page 66 for details.

6.1.3 Specifying Query Result Options

In the Query Console, SPARQL results are returned as a sequence of `json:object` values in the case of a `SELECT` query, a sequence of `sem:triple` values in the case of a `CONSTRUCT` or `DESCRIBE` query, or a single `xs:boolean` value in the case of an `ASK` query. The results for each will look different in Query Console.

This section discusses the following topics:

- Auto vs. Raw Format
- Selecting Results Rendering

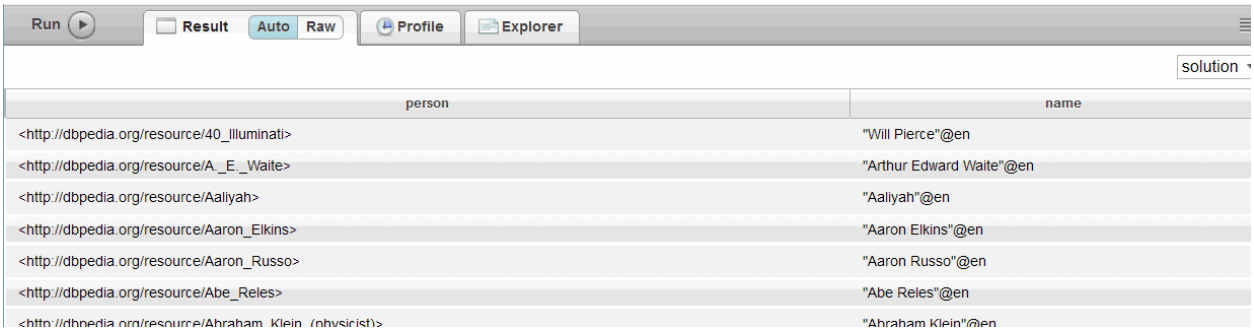
6.1.3.1 Auto vs. Raw Format

The results of a SPARQL query displays triples or `SELECT` solutions. Solution objects show a mapping from variable names to typed values. Each heterogeneous item in the result sequence will have specific rendering, which is by default shown in Auto format.

For example, this `SELECT` query returns a solution:

```
PREFIX db: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX onto: <http://dbpedia.org/ontology/>

SELECT ?person ?name
  WHERE { ?person onto:birthPlace db:Brooklyn;
    foaf:name ?name .}
```



To change the display format to Raw, click Raw on the Result tab. In Raw format, the results for the same query are displayed in RDF/JSON serialization:

```
[
 {
  "person":"<http://dbpedia.org/resource/40_Illuminati>",
  "name":"\"Will Pierce\"@en"
 },
 {
  "person":"<http://dbpedia.org/resource/A._E._Waite>",
  "name":"\"Arthur Edward Waite\"@en"
 },
 {
  "person":"<http://dbpedia.org/resource/Aaliyah>",
  "name":"\"Aaliyah\"@en"
 },
 {
  "person":"<http://dbpedia.org/resource/Aaron_Elkins>",
  "name":"\"Aaron Elkins\"@en"
 },
 {
  "person":"<http://dbpedia.org/resource/Aaron_Russo>",
  "name":"\"Aaron Russo\"@en"
 },
 {
  "person":"<http://dbpedia.org/resource/Abe_Reles>",
  "name":"\"Abe Reles\"@en"
 },
 {
  "person":"<http://dbpedia.org/resource/Abraham_Klein_(physicist)>",
  "name":"\"Abraham Klein\"@en"
```

```
 },
 {
  "person":"<http://dbpedia.org/resource/Abraham_S._Fischler>",
  "name":"\"Abraham S.Fischler\"@en"
 },
 {
  "person":"<http://dbpedia.org/resource/Abraham_S._Luchins>",
  "name":"\"Abraham S.Luchins\"@en"
 },
 {
  "person":"<http://dbpedia.org/resource/Abram_Cohen>",
  "name":"\"Abram Cohen\"@en"
 }
]
```

If you run a similar DESCRIBE query, the output is returned in Query Console in triples format:

```
PREFIX db: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX onto: <http://dbpedia.org/ontology/>

DESCRIBE ?person ?name
WHERE { ?person onto:birthPlace db:Brooklyn;
foaf:name ?name .}

=>

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/40_Illuminati>
<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Brooklyn> ,
<http://dbpedia.org/resource/New_York> ;
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type/>
<http://xmlns.com/foaf/0.1/Person> ;
<http://xmlns.com/foaf/0.1/surname> "Pierce"@en ;
<http://purl.org/dc/elements/1.1/description> "Rapper"@en ;
<http://xmlns.com/foaf/0.1/givenName> "Will"@en ;
<http://xmlns.com/foaf/0.1/name> "Will Pierce"@en .
<http://dbpedia.org/resource/A._E._Waite>
<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Brooklyn> ;
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type/>
<http://xmlns.com/foaf/0.1/Person> ;
<http://xmlns.com/foaf/0.1/givenName> "Arthur Edward"@en ;
<http://xmlns.com/foaf/0.1/name> "Arthur Edward Waite"@en ;
<http://purl.org/dc/elements/1.1/description> "English writer"@en ;
<http://xmlns.com/foaf/0.1/surname> "Waite"@en .
<http://dbpedia.org/resource/Aaliyah>
<http://dbpedia.org/ontology/deathPlace>
<http://dbpedia.org/resource/Abaco_Islands> ,
<http://dbpedia.org/resource/Marsh_Harbour> ,
<http://dbpedia.org/resource/The_Bahamas> ;
<http://dbpedia.org/ontology/birthPlace>
```

```
<http://dbpedia.org/resource/Brooklyn> ,
<http://dbpedia.org/resource/New_York_City> ;
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type/>
<http://xmlns.com/foaf/0.1/Person> ;
<http://xmlns.com/foaf/0.1/name> "Aaliyah"@en ;
<http://purl.org/dc/elements/1.1/description> "Singer, dancer,
actress, model"@en ;
<http://dbpedia.org/ontology/birthDate> "1979-01-16"^^xs:date ;
<http://dbpedia.org/ontology/deathDate> "2001-08-25"^^xs:date .
. . . .
```

Note: When you run a query that returns triples as a subgraph, the default output serialization is Turtle.

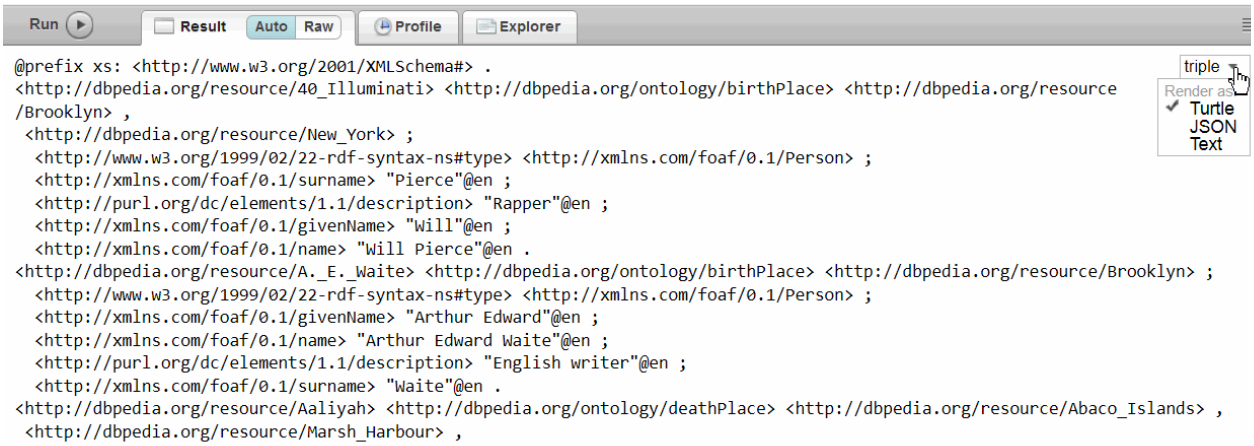
The `DESCRIBE` clause has a limit of 9999 triples in the server. If a query includes a `DESCRIBE` clause with one IRI or few IRIs that total more than 9999 triples, triples will be truncated from the results. The server does not provide any warning or message that this has occured.

6.1.3.2 Selecting Results Rendering

Use the solution as: drop-down list options to choose the display for query results. For example, this `DESCRIBE` query returns triples in Turtle serialization:

```
PREFIX db: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX onto: <http://dbpedia.org/ontology/>

DESCRIBE ?person ?name
WHERE { ?person onto:birthPlace db:Brooklyn;
foaf:name ?name .}
```



Or you can select JSON or text as the format for the results.

Note: For a `DESCRIBE` query, the rendering options are Turtle, JSON, or Text. Rendering options may be different for queries that use `cts:search`, a combination of SPARQL and `cts:` queries, or use query results that are serialized by a serialization function.

6.1.4 Constructing a SPARQL Query

You can construct a SPARQL query to ask specific questions about triples or to create new triples from triples in your triple store. A SPARQL query typically contains the following (in order):

- Prefix Declaration - abbreviates prefix IRIs
- Query Pattern - specifies what to query in the RDF graph, compares and matches query patterns
- Target RDF Graph - identifies the dataset to query
- Result Clauses - specifies the information to return from the graph
- Query Clauses - extends or restricts the scope of your query
- Solution Modifiers - specifies the order in which to return the results and the number of results

The query pattern and a result clause are the minimum required components for a query. The prefix declaration, target RDF graph, query clauses, and solution modifiers are optional components that structure and define your query.

The following example is a simple SPARQL `SELECT` query that contains a query pattern to find people whose birthplace is Paris:

```
SELECT ?s
WHERE {?s <http://dbpedia.org/ontology/birthPlace/>
<http://dbpedia.org/resource/Paris>
}
```

The following sections discuss the components of the SPARQL query in more detail, and how to compose simple and complex queries.

6.1.5 Prefix Declaration

IRIs can be long and unwieldy, and the same IRI may be used many times in a query. To make queries concise, SPARQL allows the definition of prefixes and base IRIs. Defining prefixes saves time, makes the query more readable, and can reduce errors. The prefix for a commonly used vocabulary is also known as a CURIE (Compact URI Expression).

In this example, the prefix definitions are declared and the query pattern is written with abbreviated prefixes:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX db: <http://dbpedia.org/resource/>
PREFIX onto: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT *
WHERE {
 ?s  dc:description "Physicist"@en ;
  rdf:type foaf:Person ;
 onto:birthPlace db:England .
  }
```

The query results returns the people described as “Physicist” who were born in England. The “@en” language tag means that you are searching for the English word “Physicist”. The query will match only triples with “Physicist” and an English language tag.

6.1.6 Query Pattern

At the heart of a SPARQL query is a set of triple patterns called a graph pattern. Triple patterns are like RDF triples except the subject, predicate, and object nodes may be a variable.

A graph pattern matches a subgraph of the RDF data when RDF terms from that subgraph may be substituted for the variables, and the result is an RDF graph equivalent to the subgraph.

The graph pattern is one or more triple patterns contained within curly braces ({ }). The following types of graph patterns for the query pattern are discussed in this chapter:

- Basic graph pattern - a set of triple patterns must match against triples in the triple store
- Group graph pattern - a set of graph patterns must all match using the same variable substitution
- Optional graph pattern - additional patterns may extend the solution
- Union graph pattern - where two or more possible patterns are tried
- Graph graph pattern - where patterns are matched against named graphs

SPARQL variables are denoted with a question mark (`?`) or a dollar symbol (`$`). The variables can be positioned to match any subject, predicate, or object node, and match any value in that position. Thus, the variable may be bound to an IRI or a literal (string, boolean, date, and so on). Each time a triple pattern matches a triple in the triple store, it produces a binding for each variable.

This example shows a basic graph pattern with variables to match the subject (`?s`) and predicate (`?p`) of triples where the object is “db:Paris” - to find subjects who were born or died in Paris. The query consists of two parts; the `SELECT` clause specifies what is in the query results (subject and predicate) and the `WHERE` clause provides the basic graph pattern to match against the data graph:

```
PREFIX db: <http://dbpedia.org/resource/>
PREFIX onto: <http://dbpedia.org/ontology/>

SELECT ?s ?p
WHERE {?s ?p db:Paris }
```

This query will return every person in your dataset who was born or died in Paris. You may want to limit the number of results by adding “LIMIT 10” to the end of the query. See“The LIMIT Keyword” on page 113 for details.

Note: A variable may only be bound once. The `?s` and `?p` in the `SELECT` clause are the same variables as in the `WHERE` clause.

The results of the query include the subject and predicate IRIs (for birthPlace and deathPlace) where “db:Paris” is in the object position of the triple:

<http://dbpedia.org/resource/Étienne-Denis_Pasquier>	<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Étienne-Louis_Malus>	<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/A._Kingsley_Macomber>	<http://dbpedia.org/ontology/deathPlace>
<http://dbpedia.org/resource/Abdul_Rasul_(Iraqi_scientist)>	<http://dbpedia.org/ontology/deathPlace>
<http://dbpedia.org/resource/Abdülmecid_II>	<http://dbpedia.org/ontology/deathPlace>
<http://dbpedia.org/resource/Abel_Decaux>	<http://dbpedia.org/ontology/deathPlace>

A SPARQL `SELECT` query returns a *solution*, which is a set of bindings of variables and values. By default, the results of `SELECT` queries are returned in *Auto* format, a formatted view made for easy viewing. You can change the output display. For details, see “Specifying Query Result Options” on page 83.

The previous example is a single triple pattern match (the basic graph pattern). You can query with SPARQL using multiple triple pattern matching. SPARQL uses a syntax similar to Turtle for expressing query patterns, where each triple pattern ends with a period.

Similar to an `AND` clause in SQL queries, every triple in the query pattern must be matched exactly. For example, consider place names in our dataset that can be found in different countries such as Paris, Texas or Paris, France.

The following example returns the IRIs for all resources born in Paris, France that are described as “Footballers”:

```
PREFIX db: <http://dbpedia.org/resource/>
PREFIX onto: <http://dbpedia.org/ontology/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?s ?p
WHERE {?s onto:birthPlace db:Paris .
       ?s onto:birthPlace db:France .
       ?s dc:description "Footballer"@en .
  }
```

s	p
<http://dbpedia.org/resource/Abdoulaye_Baldé_(footballer)>	<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Abdoulaye_Keita_(footballer_born_1990)>	<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Abdoulaye_Méïté>	<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Aboubacar_Sankhare>	<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Aboubacar_Tandia>	<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Ahmed_Soukouna>	<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Alain_de_Martigny>	<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Albert_Jourda>	<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Alexandre_Raineau>	<http://dbpedia.org/ontology/birthPlace>

An alternative way to write the query pattern above is to use a semicolon (`;`) in the `WHERE` clause to separate triple patterns that share the same subject.

For example:

```
PREFIX db: <http://dbpedia.org/resource/>
PREFIX onto: <http://dbpedia.org/ontology/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?s
WHERE {?s onto:birthPlace db:Paris ;
          onto:birthPlace db:France ;
          dc:description "Footballer"@en .
  }
```

The SPARQL specification allows you to use a blank node as subject and object of a triple pattern in a query.

For example:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT  ?desc
WHERE { _:p rdf:type foaf:Person ;
            dc:description ?desc .
      }
```

The query returns the role or title for resources as defined in the triples in the dataset:

desc
"Ukrainian musician"
"Competitive eater"
"American guitarist"
"American martial artist"
"Boxer"
"American guitarist"

Note: If there are blank nodes in the queried graph, blank node identifiers may be returned in the results.

6.1.7 Target RDF Graph

A SPARQL query is executed against an RDF dataset that contains graphs. These graphs can be:

- A single default graph - a set of triples with no name attached to them
- One or more named graphs - where inside a `GRAPH` clause, each named graph is a pair, made up of a name and a set of triples

For example, this query will be executed on the graph named `http://my_collections`:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>

SELECT ?name ?mbox ?date
FROM <http://my_collections>
WHERE { ?g dc:publisher ?name ;
      dc:date ?date .
     GRAPH ?g { ?person foaf:name ?name ;
     foaf:mbox ?mbox }
     }
```

“The GRAPH Keyword” on page 95 describes the use of `GRAPH` in a query.

The FROM and FROM NAMED keywords are used to specify an RDF dataset in a SPARQL query, as described in the W3C SPARQL Query Language for RDF:

http://www.w3.org/TR/rdf-sparql-query/#specifyingDataset

In the absence of FROM or FROM NAMED keywords, a SPARQL query executes against all graphs that exist in the database. In other words, if you don't specify a graph name with a query, the UNION of all graphs will be queried.

Using XQuery, REST, or Javascript you can also specify one or more graphs to be queried by using:

- a `default-graph-uri*` - Selects the graph name(s) to query, usually a subset of the available graphs.
- a `named-graph-uri*` - Used with FROM NAMED and GRAPH to specify the IRI(s) to be substituted for a name within particular kinds of queries. You can have one or more `named-graph-uri*` parameters specified as part of a query.

If you specify `default-graph-uri*`, one or more graph names that you specify will be queried. The "`*`" indicates that one or more `default-graph-uri` or `named-graph-uri` parameters can be specified.

> **Note:** This `default-graph-uri` is not the "default" graph that contains unnamed triples - `http://marklogic.com/semantics#default-graph`.

In this example a SPARQL query is wrapped in XQuery, to search the data set in the `http://example.org/bob/foaf.rdf` graph:

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

sem:sparql('
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?g ?name
WHERE {graph ?g { ?alum foaf:schoolHomepage <http://www.ucsb.edu/> .
    ?alum foaf:knows ?person .
    ?person foaf:name ?name }
      }
      '
      ()
      ("default-graph-uri=http://example.org/bob/foaf.rdf")
```

The FROM in a SPARQL query functions the same as `default-graph-uri`, and the FROM NAMED functions the same as `named-graph-uri`. These two clauses function in the same way as part of the SPARQL query, except that one is written into queries (wrapped in the query), while the other is specified outside of the query.

This section discusses the following topics:

- The FROM Keyword
- The FROM NAMED Keywords
- The GRAPH Keyword

6.1.7.1 The FROM Keyword

The `FROM` clause in a SPARQL query tells SPARQL where to get data to query, which graph to query. To use `FROM` as part of a query, there has to be a graph with the name in the `FROM` clause. Graph names in MarkLogic are implemented as collections, which you can view using Explore or the `cts:collections` function in the Query Console.

This SPARQL query uses the `FROM` keyword to search data in the `info:govtrack/people` graph:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

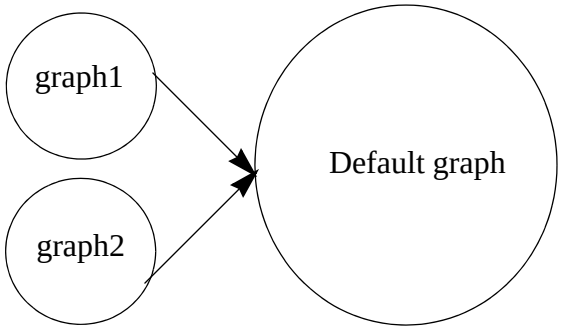
SELECT  ?name
FROM    <http://marklogic.com/semantics#info:govtrack/people/>
WHERE   { ?x foaf:name ?name }
LIMIT 10
```

See “Preparing to Run the Examples” on page 129 for information about the GovTrack dataset.

The default graph is the result of an RDF merge of the graphs (a union of graphs) referred to in one or more `FROM` clauses. Each `FROM` clause contains an IRI that indicates a graph to be used to form the default graph.

For example, graph1 and graph2 are merged to form the default graph:

```
FROM graph1
FROM graph2
```



Note: When we talk about the default graph in this sense, it is not the same as the default collections, `http://marklogic.com/semantics#default-graph`.

This example shows a SPARQL SELECT query that returns all triples where "Alice" is in the object position. The RDF dataset contains a single default graph and no named graphs:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?s ?p ?o
FROM <http://example.org/foaf/alice>
WHERE {?s foaf:name "Alice";
?p ?o .}
```

Note: The FROM keyword must be placed before the WHERE clause. Placing the FROM keyword after the WHERE clause causes a syntax error.

6.1.7.2 The FROM NAMED Keywords

A query can supply IRIs for the named graphs in the dataset using the FROM NAMED clause. Each IRI is used to provide one named graph in the dataset. Having multiple FROM NAMED clauses causes multiple graphs to be added to the dataset. With FROM NAMED, every graph name you use in the query will be matched only to the graph provided in the clause.

You can set the named-graph at load time using mlcp with the collection parameter -output_collections http://www.example.org/my_graph. See "Specifying Collections and a Directory" on page 49. You can also set the named-graph using the REST client with PUT:/v1/graphs.

Note: A named graph is typically created when you load RDF data. See "Loading Triples" on page 37.

In a query, FROM NAMED is used to identify a named graph that is queried from the WHERE clause by using the GRAPH keyword.

For example:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?who ?g ?mbox
FROM <http://example.org/foaf/aliceFoaf>
FROM NAMED <http://example.org/alice>
FROM NAMED <http://example.org/bob>
WHERE
{
   ?g dc:publisher ?who .
   GRAPH ?g { ?x foaf:mbox ?mbox }
}
```

In the example, the FROM and FROM NAMED keywords are used together. The FROM NAMED is used to scope the graphs that are considered during query evaluation, and the GRAPH construct specifies one of the named graphs.

Note: When FROM or FROM NAMED keywords are used, the graphs you can use in a GRAPH clause potentially become restricted.

6.1.7.3 The GRAPH Keyword

The GRAPH keyword instructs the query engine to evaluate part of the query against the named graphs in the dataset. A variable used in the GRAPH clause may also be used in another GRAPH clause or in a graph pattern matched against the default graph in the dataset.

For example:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>

SELECT ?name ?mbox ?date
WHERE { ?g dc:publisher ?name ;
       dc:date ?date .
      GRAPH ?g { ?person foaf:name ?name ;
      foaf:mbox ?mbox }
      }
```

Note: You must enable the collection lexicon when you use a GRAPH construct in a SPARQL query. You can enable the collection lexicon from the database configuration pages or the Admin Interface.

Triples inside of a GRAPH clause with an explicit IRI, such as `GRAPH <...uri...> { ...graph pattern... }`, are matched against the dataset using the IRI specified in the graph clause.

6.1.8 Result Clauses

Querying the dataset with different types of SPARQL queries returns different types of results. These SPARQL query forms return the following result clauses:

- SELECT Queries - returns a sequence of variable bindings
- CONSTRUCT Queries - returns an RDF graph constructed by substituting variables in a set of triple templates
- DESCRIBE Queries - returns an RDF graph that describes the resources found
- ASK Queries - returns a boolean indicating whether a query pattern matches

6.1.8.1 SELECT Queries

The SPARQL `SELECT` keyword indicates that you are requesting data from a dataset. This SPARQL query is the most widely used of the query forms. SPARQL `SELECT` queries return a sequence of bindings as a solution, that satisfies the query. Selected variables are separated by white spaces, not commas.

You can use the asterisk wildcard symbol (`*`) with SPARQL `SELECT` as shorthand for selecting all the variables identified in the query pattern.

For example:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT *
WHERE{?s foaf:givenName ?fn .
       ?s foaf:surname ?ln .
     }
```

Note: In single triple patterns, a period at the end is optional. In a query pattern with multiple triple patterns, the period at the end of final triple is also optional.

In the example, the `SELECT` query returns a sequence of bindings that includes the IRI for the subject variable (`?s`), along with the first name (`?fn`) and last name (`?ln`) of resources in the dataset.

SPARQL `SELECT` query results are serialized as XML, JSON, or passed to another function as a map. The results of a `SELECT` query may not always be triples.

6.1.8.2 CONSTRUCT Queries

You can create new triples from existing triples by using SPARQL `CONSTRUCT` queries. When you execute a construct query, the results are returned in a sequence of `sem:triple` values as triples in memory.

This example creates triples for Albert Einstein from the existing triples in the database:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

CONSTRUCT
{?person ?p ?o .}
WHERE {?person foaf:givenName "Albert"@en ;
       foaf:surname "Einstein"@en ;
       ?p ?o .}
```

The `CONSTRUCT` queries return an RDF graph created from variables in the query pattern.

These triples are created for Albert Einstein from the existing triples in the dataset:

```
@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Albert_Einstein>
<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Baden-Württemberg> .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Albert_Einstein>
<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/German_Empire> .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Albert_Einstein>
<http://dbpedia.org/ontology/deathPlace>
<http://dbpedia.org/resource/Princeton,_New_Jersey> .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Albert_Einstein>
<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Ulm> .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Albert_Einstein>
<http://dbpedia.org/ontology/deathPlace>
<http://dbpedia.org/resource/United_States> .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Albert_Einstein>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://xmlns.com/foaf/0.1/Person> .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Albert_Einstein>
<http://xmlns.com/foaf/0.1/givenName> "Albert"@en .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Albert_Einstein>
<http://xmlns.com/foaf/0.1/name> "Albert Einstein"@en .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Albert_Einstein>
<http://xmlns.com/foaf/0.1/surname> "Einstein"@en .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Albert_Einstein>
<http://purl.org/dc/elements/1.1/description> "Physicist"@en .
```

These triples are constructed in memory and not added to the database.

Note: The “@en” language tag means that this is an English word and will match differently than any other language tag.

6.1.8.3 DESCRIBE Queries

SPARQL `DESCRIBE` queries return a sequence of `sem:triple` values. The `DESCRIBE` query result returns RDF graphs that describe one or more of the given resources. The W3C specification leaves the details implementation dependent. In MarkLogic, we return a Concise Bounded Desciption of the IRIs identified, which includes all triples which have the IRI as a subject, and for each of those triples that has a blank node as an object, all triples with those blank nodes as a subject. This implementation does not provide any reified statements, and will return a maximum of 9999 triples.

For example, this query finds triples containing “Pascal Bedrossian”:

```
DESCRIBE <http://dbpedia.org/resource/Pascal_Bedrossian>
```

The triples found by the `DESCRIBE` query are returned in Turtle format. You can also select JSON or Text as the format.

```
@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/France> .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Marseille> .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://xmlns.com/foaf/0.1/Person> .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://xmlns.com/foaf/0.1/surname> "Bedrossian"@en .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://xmlns.com/foaf/0.1/givenName> "Pascal"@en .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://xmlns.com/foaf/0.1/name> "Pascal Bedrossian"@en .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://purl.org/dc/elements/1.1/description> "footballer"@en .

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://dbpedia.org/ontology/birthDate> "1974-11-28"^^xs:date .
```

Note: The `DESCRIBE` clause has a limit of 9999 triples in the server, which means if a query includes a `DESCRIBE` clause with one IRI or few IRIs that total more than 9999 triples, triples will be truncated from the results. The server does not provide any warning or message that this has occured.

6.1.8.4 ASK Queries

SPARQL `ASK` queries return a single `xs:boolean` value. The `ASK` clause returns `true` if the query pattern has any matches in the dataset and `false` if there is no pattern match.

For example, in the `persondata` dataset are the following facts about two members of the Kennedy family: Carolyn Bessette-Kennedy and Eunice Kennedy-Shriver:

- Eunice Kennedy-Shriver, the founder of the Special Olympics precursor and a sister of John F. Kennedy was born on 1921-07-10.
- Carolyn Bessette-Kennedy, a publicist, and wife of JFK Junior, was born on 1966-01-07.

This query asks if Carolyn was born after Eunice.

```
PREFIX db: <http://dbpedia.org/resource/>
PREFIX onto: <http://dbpedia.org/ontology/>

ASK
{
  db:Carolyn_Bessette-Kennedy onto:birthDate ?by .
  db:Eunice_Kennedy_Shriver onto:birthDate ?bd .
  FILTER (?by>?bd).
}
=>
true
```

The response is `true`.

Note: `ASK` queries check to see if there is at least one result.

6.1.9 Query Clauses

Add the following query clauses to extend or reduce the number of potential results returned:

- The OPTIONAL Keyword
- The UNION Keyword
- The FILTER Keyword
- Comparison Operators
- Negation in Filter Expressions
- BIND Keyword
- Values Sections

6.1.9.1 The OPTIONAL Keyword

The `OPTIONAL` keyword is used to return additional results if there is a match in an optional graph pattern. For example, this query pattern returns triples in the database consisting of the first name (`?fn`), last name (`?ln`) and mail address (`?mb`):

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?fn ?ln
WHERE{?x foaf:givenName ?fn .
      ?x foaf:surname ?ln .
      ?x foaf:email ?mb .
}
```

Only triples that match *all* the triple patterns are returned. In the `persondata` dataset there may be people with no email address. In this case, the Query Console will silently leave these people out of the result set.

You can use the optional graph pattern (also known as a *left join*) to return matching values of any variables in common, if they exist. Since the `OPTIONAL` keyword is also a graph pattern, it has its own set of curly braces (inside the curly braces of the `WHERE` clause).

This example extends the previous example to return one or more email addresses, and just the first name and last name if there is no email address:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?fn ?ln
WHERE {?x foaf:givenName ?fn .
       ?x foaf:surname ?ln .
OPTIONAL{?x foaf:email ?mb .}
}
```

Note: Optional patterns may yield unbound variables. See “ORDER BY Keyword” on page 113 for more about unbound variables.

6.1.9.2 The UNION Keyword

Use the `UNION` keyword to match multiple patterns from multiple different sets of data, and then combine them in the query result. The `UNION` keyword is placed inside the curly braces of the `WHERE` clause. The syntax is:

```
{ triple pattern } UNION { triple pattern }
```

The `UNION` pattern combines graph patterns; each alternative possibility can contain more than one triple pattern (*logical disjunction*).

This example finds people who are described as “Authors” or “Novelists” and their date of birth:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX onto: <http://dbpedia.org/ontology/>

SELECT  ?person ?desc ?date
WHERE { ?person rdf:type foaf:Person .
      ?person dc:description ?desc .
      ?person onto:birthDate ?date .

      { ?person dc:description "Novelist"@en . }
UNION
      { ?person  dc:description "Author"@en . }
      }
```

You can also group triple patterns into multiple graph patterns using a group graph pattern structure.

For example:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?person ?desc
WHERE {{?person rdf:type foaf:Person }
       {?person dc:description ?desc }

      {{?person dc:description "Author"@en }

 UNION
      { ?person  dc:description "Novelist"@en . } } }
```

Note that each set of braces contains a triple. This is semantically equivalent to this next query and would yield the same results.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?person
WHERE {?person rdf:type foaf:Person ;
               dc:description ?desc .

   {?person dc:description "Author"@en }

UNION
   {?person  dc:description "Novelist"@en . }
      }
```

Note: You can use multiple `UNION` patterns in a SPARQL query. The results from the `OPTIONAL` and `UNION` queries differ in that the `UNION` query allows a subgraph of another solution, while an `OPTIONAL` query explicitly does not.

6.1.9.3 The FILTER Keyword

There are multiple methods for limiting the results of a SPARQL query. You can use the `FILTER`, `DISTINCT`, or the `LIMIT` keywords to restrict the number of matching results that are returned.

You can use one or more SPARQL `FILTER` keywords to specify the variables by which to constrain results. The `FILTER` constraint is placed inside the curly braces of the `WHERE` clause and can contain symbols for logical, mathematical, or comparison operators such as greater than (`>`), less than(`<`), equal to (`=`), and so on. The `FILTER` constraints use boolean conditions to return matching query results. There are also a number of built-in SPARQL tests you can use such as `isURI`, `isBlank`, and so forth.

This table lists some of the SPARQL unary operators in `FILTER` constraints:

Operator	Type	Result Type
`!`	`xsd:boolean`	`xsd:boolean`
`+`	numeric	numeric
`-`	numeric	numeric
`BOUND()`	variable	`xsd:boolean`
`isURI()`	RDF term	`xsd:boolean`
`isBLANK()`	RDF term	`xsd:boolean`
`isLITERAL`	RDF term	`xsd:boolean`
`STR()`	literal/IRI	simple literal
`LANG()`	literal	simple literal
`DATATYPE()`	literal	IRI

Note: For a full list of operations, see *Operator Mapping* in the SPARQL Query Language for RDF.

This example is a query pattern that provides meaning to the variable `?bd` (a person's birth date). The `FILTER` clause of the query pattern compares the variable value to the date January 1st, 1999 and returns people born after the given date:

```
PREFIX onto: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>

SELECT ?s
WHERE {?s rdf:type foaf:Person .
       ?s onto:birthDate ?bd .
  FILTER (?bd > "1999-01-01"^^xsd:date)
}
```

The SPARQL keyword `a` is a shortcut for the common predicate `rdf:type`, giving the class of a resource. For example, the `WHERE` clause could be written as:

```
WHERE {?s a foaf:Person .
       ?s onto:birthDate ?bd .
```

You can express a `FILTER` clause with a regular expression pattern by using the `regex` function. For example:

```
SELECT ?s ?p ?o
WHERE {?s ?p ?o
  FILTER (regex (?o, "Lister", "i"))
}
```

The SPARQL query returns all matching results where the text in the object position contains the string `Lister` in any case. Regular expression matches are made case-insensitive with the `i` flag.

> **Note:** This type of `FILTER` query is the equivalent of the `fn:match` XQuery function. Regular expressions are not optimized in SPARQL. Use `cts:contains` for optimized full text searching.

The regular expression language is defined in *XQuery 1.0 and XPath 2.0 Functions and Operators*, section 7.6.1 Regular Expression Syntax.

6.1.9.4 Using Built-in Functions in a SPARQL Query

In addition to SPARQL functions, you can use XQuery or JavaScript built-in functions (for example, functions with the prefix `fn`, `cts`, `math`, or `xdmp`) in a SPARQL query where you can use a function, which includes `FILTER`, `BIND`, and the expressions in a `SELECT` statement.

A built-in function is one that can be called without using “import module” in XQuery or “var <module> = require” in JavaScript. These functions are called extension functions when used in a SPARQL query. You can find a list of built-in functions at http://docs.marklogic.com by selecting “Server-Side JavaScript APIs” (or “Server-Side XQuery APIs”). The built-ins listed are under “MarkLogic Built-In Functions” and “W3C-Standard Functions.” See “Using Semantic Functions to Query” on page 130 for more information.

Extension functions in SPARQL are identified by IRIs in the form of `http://www.w3.org/2005/xpath-functions#`*`name`* where *`name`* is the local name of the function and the string before the # is the prefix IRI of the function, for example `http://www.w3.org/2005/xpath-functions#starts-with`. For the prefix IRIs commonly associated with `fn`, `cts`, `math`, and `xdmp` (or any other prefix IRIs that do not end with a "`/`" or "`#`"), append a `#` to the prefix IRI and then the function local name, for example: `http://marklogic.com/cts#contains`.

You can access built-in functions like `cts` using `PREFIX` in the SPARQL query. In this example, `cts:contains` is added as using `PREFIX` and then included as part of the `FILTER` query:

```
PREFIX cts: <http://marklogic.com/cts#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT *
WHERE{ ?s ?p ?o .
  FILTER cts:contains(?o, cts:or-query(("Monarch", "Sovereign")))
  FILTER(?p IN (dc:description, rdfs:type))
}
```

This is full-text search for the words “Monarch” or “Sovereign” where the predicate is either a description or a type. In the second `FILTER` clause, the use of `IN` specifies the predicates to filter on. The results include people that have a title of “Monarch” (of a kingdom, state or sovereignty) and things that have a description of “Monarch” such as the Monarch butterfly or Monarch Islands.

In this example the XPath function `starts-with` is used in a SPARQL query to return the roles or titles of people whose description begins with "Chief". The function is imported by including the IRI as part of the `FILTER` query:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?desc
WHERE {?s dc:description ?desc
  FILTER (<http://www.w3.org/2005/xpath-functions#starts-with>( ?desc,
"Chief" ) )}
```

Note: You can use the `FILTER` keyword with the `OPTIONAL` and `UNION` keywords.

6.1.9.5 Comparison Operators

The `IN` and `NOT IN` comparison operators are used with the `FILTER` clause to return a boolean `true` if a matching term is in the set of expressions, or `false` if not. For example:

```
ASK {
  FILTER(2 IN (1, 2, 3))
}

=>
true

ASK {
  FILTER(2 NOT IN (1, 2, 3))
}

=>
false
```

6.1.10 Negation in Filter Expressions

Negation can be used with the `FILTER` expression to eliminate solutions from the query results. There are two types of negation - one type filters results depending on whether a graph pattern does or does not match in the context of the query solution being filtered, and the other type is based on removing solutions related to another pattern. MarkLogic supports SPARQL 1.1 Negation (using `EXISTS`, `NOT EXISTS`, and `MINUS`) for use with `FILTER`.

The examples for negation use this data:

```
PREFIX : <http://example.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

:alice  rdf:type   foaf:Person .
:alice  foaf:name  "Alice" .
:bob    rdf:type   foaf:Person .
```

This section contains these topics:

- EXISTS
- NOT EXISTS
- MINUS
- Differences Between NOT EXISTS and MINUS
- Combination Queries with Negation

6.1.10.1 EXISTS

The filter expression `EXISTS` checks to see whether the query pattern can be found in the data. For example, the `EXISTS` filter in this examples checks for the pattern `?person foaf:name ?name` in the data:

```
PREFIX  rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  foaf:   <http://xmlns.com/foaf/0.1/>

SELECT ?person
WHERE
{
    ?person rdf:type  foaf:Person .
    FILTER EXISTS { ?person foaf:name ?name }
}

=>
    person
<http://example.org/alice
```

The result of the query is Alice. The `EXISTS` filter does not generate any additional bindings.

6.1.10.2 NOT EXISTS

With the `NOT EXISTS` filter expression, the query tests whether a graph pattern *does not* match a dataset, given the values of variables in the group graph pattern in which the filter occurs. This query tests whether the `?person foaf:name ?name` does not occur in the data:

```
PREFIX  rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  foaf:   <http://xmlns.com/foaf/0.1/>

SELECT ?person
WHERE
{
    ?person rdf:type  foaf:Person .
    FILTER NOT EXISTS { ?person foaf:name ?name }
}

=>
    person
<http://example.org/bob>
```

The graph pattern for `<http://example.org/bob>` does not have a predicate `foaf:name` for `?person`, so the query returns Bob as the result for this query. The `NOT EXISTS` filter does not generate any additional bindings.

6.1.10.3 MINUS

The another type of SPARQL negation is `MINUS`, which evaluates both its arguments, then calculates solutions in the left-hand side that are not compatible with the solutions on the right-hand side of the pattern.

For this example we will add additional data:

```
PREFIX :       <http://example.org/>
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>

:alice  foaf:givenName "Alice" ;
        foaf:familyName "Smith" .

:bob    foaf:givenName "Bob" ;
        foaf:familyName "Jones" .

:carol  foaf:givenName "Carol" ;
        foaf:familyName "Smith" .
```

This query looks for patterns in the data that do not match `?s foaf:givenName "Bob"` and returns those results:

```
PREFIX :       <http://example.org/>
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>

SELECT DISTINCT ?s
WHERE {
   ?s ?p ?o .
   MINUS {
      ?s foaf:givenName "Bob" .
   }
}

=>
<http://example.org/carol>
<http://example.org/alice>
```

The results are Carol and Alice.

The filters `NOT EXISTS` and `MINUS` represent two ways of approaching negation. The `NOT EXISTS` approach tests whether a pattern exists in the data, based on the bindings determined by the query pattern. The `MINUS` approach removes matches based on the evaluation of two patterns. In some cases, they can produce different results. The `MINUS` filter does not generate any additional bindings.

6.1.10.4 Differences Between NOT EXISTS and MINUS

The filter expressions `NOT EXISTS` and `MINUS` represent two ways of using negation. The `NOT EXISTS` filter tests whether a pattern exists in the data, given the bindings already determined by the query pattern. The `MINUS` filter removes matches from the result set based on the evaluation of two patterns in the query. In some cases, these two approaches can produce different answers.

Example: Sharing of variables

If we have this dataset:

```
@prefix : <http://example.com/> .
:a :b :c .
```

And we use this query:

```
SELECT *
{
  ?s ?p ?o
  FILTER NOT EXISTS {?x ?y ?x}
}

=>
(This query has no results)
```

The result set will be empty because `{?x ?y ?x}` matches all triples in the data, which the `NOT EXISTS` filter eliminates from the results.

When we use `MINUS` in the same query, there is no shared variable between the first part (`?s ?p ?o`) and the second part (`?x ?y ?z`), so no bindings are eliminated:

```
SELECT *
{
  ?s ?p ?o
  FILTER MINUS {?x ?y ?x}
}

=>
s                         p                         o
<http://example.com/a> <http://example.com/b> <http://example.com/c>
```

Example: Fixed pattern

Another case where the results will be different for `NOT EXISTS` and `MINUS` is where there is a concrete pattern (no variables) in the example query.

This query uses `NOT EXISTS` as the filter for negation:

```
PREFIX : <http://example.com/>
SELECT *
{
 ?s ?p ?o
```

```
 FILTER NOT EXISTS {:a :b :c}
}

=>
(This query has no results)
```

This query uses `MINUS` as the filter:

```
PREFIX : <http://example.com/>
SELECT *
{
 ?s ?p ?o
 MINUS {:a :b :c}
}

=>
s                        p                          o
<http://example.com/a> <http://example.com/b> <http://example.com/c>
```

Since there is no match of bindings, no solutions are eliminated, and the solution includes a, b, and c.

Example: Inner FILTERs

Differences in results will also occur because in a filter, variables from the group are in scope. In this example, the `FILTER` inside the `NOT EXISTS` has access to the value of `?n` for the solution being considered. For this example, we will use this dataset:

```
PREFIX : <http://example.com/>
:a :p 1 .
:a :q 1 .
:a :q 2 .

:b :p 3.0 .
:b :q 4.0 .
:b :q 5.0 .
```

When using `FILTER NOT EXISTS`, the test is on each possible solution to `?x :p ?n` in this query:

```
PREFIX : <http://example.com/>
SELECT * WHERE {
 ?x :p ?n
 FILTER NOT EXISTS {
     ?x :q :m .
     FILTER (?n = ?m)
     }
 }

=>
x                       n
<http://example.com/b> 3.0
```

With `MINUS`, the `FILTER` inside the pattern does not have a value for `?n` and it is always unbound.

```
PREFIX : <http://example.com/>
SELECT * WHERE {
    ?x ?p  ?n
    MINUS {
          ?x :q ?m .
          FILTER (?n = ?m)
    }
 }

=>
x                     n
<http://example.com/b> 3.0
<http://example.com/a> 1
```

6.1.10.5 Combination Queries with Negation

A combination query operates on triples embedded in documents. The query searches both the document and any triples embedded in the document. You can add negation with the `FILTER` keyword to constrain the results of the query.

For example:

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

let $query := '
  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  SELECT ?country
  WHERE {
    <http://example.org/news/Nixon> <http://example.org/wentTo>
?country
  FILTER NOT EXISTS {?country foaf:isIn ?location .
                ?location foaf:isIn "Europe"} . }'
let $store := sem:store((),cts:and-query( (
    cts:path-range-query( "//triples-context/confidence", ">=", 80) ,
    cts:path-range-query( "//triples-context/pub-date", ">",
xs:date("1974-01-01")),
    cts:or-query( (
      cts:element-value-query( xs:QName("source"), "AP Newswire" ),
      cts:element-value-query( xs:QName("source"), "BBC" )
   )))))

let $result := sem:sparql($query, (), (), $store)
return <result>{$result}</result>
```

Note: The `cts:path-range-query` requires the path index to be configured to work correctly. See Understanding Range Indexes in the *Administrator's Guide*.

This is a modification of an earlier query that says “Find all of the documents containing triples that have information about countries that Nixon visited. From that group, return only those triples that have a confidence level of 80% or above and a publication date after January 1st, 1974. And only return triples with a source element of AP Newswire or BBC.” The `MINUS` filter removes any countries that are located in Europe from the results.

Note: SPARQL Update will not modify triples embedded in documents. SPARQL Update can be used to insert new triples into graphs as part of a combination query, or to modify managed triples. See “Unmanaged Triples” on page 73 for more information about triples in documents.

6.1.10.6 BIND Keyword

The `BIND` keyword allows a value to be assigned to a variable from a basic graph pattern or property path expression. The use of `BIND` ends the preceding basic graph pattern. The variable introduced by the `BIND` clause must not have been used in the group graph pattern up to the point of use in `BIND`. When you assign a computed value to a variable in the middle of a pattern, the computed value can then be used in other patterns, such as a `CONSTRUCT` query. The syntax is (`expression AS ?var`). For example:

```
PREFIX db: <http://dbpedia.org/resource/>
PREFIX onto: <http://dbpedia.org/ontology/>

SELECT ?person
  WHERE { BIND (db:London AS ?location)
          ?person onto:birthPlace ?location .
  }
LIMIT 10
```

6.1.10.7 Values Sections

You can use SPARQL `VALUES` sections to provide inline data as an unordered solution sequence that is joined with the results of the query evaluation. The `VALUES` section allows multiple variables to be specified in the data block. For example:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT *
WHERE { ?a foaf:name ?n .
VALUES ?n { "John" "Jane" }}
```

This query says “find subjects with `foaf:name` John or Jane” - supplying the values the `?n` can have instead of searching for `?n` in the dataset. This is the same as a query using the longer form where the parameter lists are contained in parentheses:

```
VALUES (?z) { ("John") ("Jane") }
```

Note: A `VALUES` block of data can appear in a query pattern or at the end of a `SELECT` query or subquery.

6.1.11 Solution Modifiers

A solution modifier modifies the result set for `SELECT` queries. This section discusses how you can modify what your query returns using the following solution modifiers:

- The DISTINCT Keyword
- The LIMIT Keyword
- ORDER BY Keyword
- The OFFSET Keyword
- Subqueries
- Projected Expressions

Note: With the exception of `DISTINCT`, modifiers appear after the `WHERE` clause.

6.1.11.1 The DISTINCT Keyword

Use the `DISTINCT` keyword to remove duplicate results from a results set.

For example:

```
SELECT DISTINCT ?p
WHERE {?s ?p ?o}
```

The query returns all of the predicates - just once - for all the triples in the `persondata` dataset.

p
<http://dbpedia.org/ontology/birthDate>
<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/ontology/deathDate>
<http://dbpedia.org/ontology/deathPlace>
<http://purl.org/dc/elements/1.1/description>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://xmlns.com/foaf/0.1/givenName>
<http://xmlns.com/foaf/0.1/name>
<http://xmlns.com/foaf/0.1/surname>

6.1.11.2 The LIMIT Keyword

Use the `LIMIT` keyword to further restrict the results of a SPARQL query that are displayed. For example, in the DBPedia dataset, there could be thousands of authors that match this query:

```
PREFIX dc:<http://purl.org/dc/elements/1.1/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>

SELECT ?x ?fn ?ln
WHERE{?x dc:description "Author"@en ;
      foaf:name ?fn ;
      foaf:surname ?ln.}
```

To specify the number matching results to display, add the `LIMIT` keyword after the curly braces of the `WHERE` clause with an integer (not a variable).

For example:

```
PREFIX dc:<http://purl.org/dc/elements/1.1/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>

SELECT ?x ?fn ?ln
WHERE{?x dc:description "Author"@en ;
      foaf:name ?fn ;
      foaf:surname ?ln.}
LIMIT 10
```

The results of the query are limited to the first ten matches:

x	fn	
<http://dbpedia.org/resource/A.S._King>	"A.S. King"@en	"King"@en
<http://dbpedia.org/resource/A._H._Almaas>	"A.H. Almaas"@en	"Almaas"@en
<http://dbpedia.org/resource/A._Lee_Martinez>	"A. Lee Martinez"@en	"Martinez"@en
<http://dbpedia.org/resource/A._M._Burrage>	"A. M. Burrage"@en	"Burrage"@en
<http://dbpedia.org/resource/A._Muthukrishnan>	"A. Muthukrishnan"@en	"Muthukrishnan"@
<http://dbpedia.org/resource/Abidemi_Sanusi>	"Abidemi Sanusi"@en	"Sanusi"@en
<http://dbpedia.org/resource/Ada_Albrecht>	"Ada Albrecht"@en	"Albrecht"@en
<http://dbpedia.org/resource/Adèle_Geras>	"Adele Geras"@en	"Geras"@en
<http://dbpedia.org/resource/Agnete_Friis>	"Agnete Friis"@en	"Friis"@en
<http://dbpedia.org/resource/Ahmad_Akbarpour>	"Ahmad Akbarpour"@en	"Akbarpour"@en

6.1.11.3 ORDER BY Keyword

Use the `ORDER BY` clause to specify the values of one or more variable by which to sort the query results. SPARQL provides an ordering for unbound variables, blank nodes, IRIs, or RDF literals as described in the SPARQL 1.1 Query Language recommendation:

http://www.w3.org/TR/sparql11-query/#modOrderBy

The default ordering is ascending order.

For example:

```
PREFIX dc:<http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?x ?fn ?ln
WHERE{?x dc:description "Author" ;
      foaf:name ?fn ;
      foaf:surname ?ln.}
ORDER BY ?ln ?fn
LIMIT 10
```

The results are ordered by the author's last name (`?ln`) and then by the author's first name (`?fn`):

x	fn	ln
<http://dbpedia.org/resource/Patience_Abbe>	"Patience Abbe"@en	"Abbe"@en
<http://dbpedia.org/resource/Lynn_Abbey>	"Lynn Abbey"@en	"Abbey"@en
<http://dbpedia.org/resource/George_Abbot_(author)>	"George Abbot"@en	"Abbot"@en
<http://dbpedia.org/resource/Eleanor_Hallowell_Abbott>	"Eleanor Hallowell Abbott"@en	"Abbott"@en
<http://dbpedia.org/resource/Hailey_Abbott>	"Hailey Abbott"@en	"Abbott"@en
<http://dbpedia.org/resource/Walid_Abdallah>	"Walid Abdallah"@en	"Abdallah"@en
<http://dbpedia.org/resource/Robert_Abernathy>	"Robert Abernathy"@en	"Abernathy"@en
<http://dbpedia.org/resource/Susan_Abulhawa>	"Susan Abulhawa"@en	"Abulhawa"@en
<http://dbpedia.org/resource/Rodolfo_Acevedo>	"Rodolfo Acevedo"@en	"Acevedo"@en
<http://dbpedia.org/resource/John_M._Ackerman>	"John M. Ackerman"@en	"Ackerman"@en

To change the order of results to descending order, use the `DESC` keyword and place the variable for the values to be returned in brackets. For example:

```
PREFIX dc:<http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?x ?fn ?ln
WHERE{?x dc:description "Author"@en ;
      foaf:name ?fn ;
      foaf:surname ?ln .}
ORDER BY DESC (?ln)
LIMIT 10
```

x	fn	ln
<http://dbpedia.org/resource/Joan_de_Hamel>	"Joan de Hamel"@en	"de Hamel"@en
<http://dbpedia.org/resource/Laila_al-Ouhaydib>	"Laila al-Ouhaydib"@en	"al-Ouhaydib"@en
<http://dbpedia.org/resource/Shahidul_Zahir>	"Shahidul Zahir"@en	"Zahir"@en
<http://dbpedia.org/resource/Anwer_Zahidi>	"Anwer Zahidi"@en	"Zahidi"@en
<http://dbpedia.org/resource/Helen_Zahavi>	"Helen Zahavi"@en	"Zahavi"@en
<http://dbpedia.org/resource/Rachel_Zadok>	"Rachel Zadok"@en	"Zadok"@en
<http://dbpedia.org/resource/Michele_Zackheim>	"Michele Zackheim"@en	"Zackheim"@en
<http://dbpedia.org/resource/Shan_Sa>	"Ni Yan"@en	"Yan"@en
<http://dbpedia.org/resource/Evie_Wyld>	"Evie Wyld"@en	"Wyld"@en
<http://dbpedia.org/resource/Patricia_Wrede>	"Patricia C. Wrede"@en	"Wrede"@en

6.1.11.4 The OFFSET Keyword

The `OFFSET` modifier is used for pagination, to skip a given number of matching query results before returning the remaining results. This keyword can be used with the `LIMIT` and `ORDER BY` keywords to retrieve different slices of data from a dataset. For example, you can create pages of results from different offsets.

This example queries for Authors in ascending order and limits the results to the first twenty, skipping the first eight matches and starting the list at position nine:

```
PREFIX dc:<http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?x ?fn ?ln
WHERE{?x dc:description "Author"@en ;
      foaf:name ?fn ;
      foaf:surname ?ln.}
ORDER BY ?x
OFFSET 8
LIMIT 20
```

The results are returned, skipping the first eight matches.

x	fn	ln
<http://dbpedia.org/resource/Agnete_Friis>	"Agnete Friis"@en	"Friis"@en
<http://dbpedia.org/resource/Ahmad_Akbarpour>	"Ahmad Akbarpour"@en	"Akbarpour"@en
<http://dbpedia.org/resource/Ajip_Rosidi>	"Ajip Rosidi"@en	"Rosidi"@en
<http://dbpedia.org/resource/Alauddin_Masood>	"Alauddin Masood"@en	"Masood"@en
<http://dbpedia.org/resource/Alberto_Fortis>	"Abbe Alberto Fortis"@en	"Fortis"@en
<http://dbpedia.org/resource/Alexandra_Hawkins>	"Alexandra Hawkins"@en	"Hawkins"@en
<http://dbpedia.org/resource/Alexandre_Beljame>	"Alexandre Beljame"@en	"Beljame"@en
<http://dbpedia.org/resource/Alexis_Jenni>	"Alexis Jenni"@en	"Jenni"@en
<http://dbpedia.org/resource/Alexis_Lecaye>	"Alexis Lecaye"@en	"Lecaye"@en
<http://dbpedia.org/resource/Alfred_Leland_Crabb>	"Alfred Leland Crabb"@en	"Crabb"@en

Note: SPARQL uses a 1-based index, meaning the first item is 1 and not 0, so an offset of 8 will skip items one through eight.

6.1.11.5 Subqueries

You can combine the results of several queries by using subqueries. You can nest one or more queries inside another query. Each subquery is enclosed in separate pairs of curly braces. Typically, subqueries are used with solution modifiers. This example queries for Politicians who were born in London and then limits the results to the first ten:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX db: <http://dbpedia.org/resource/>
PREFIX onto: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT  ?name ?location ?date
```

```
WHERE
   { ?person dc:description "Politician"@en .

    {SELECT ?location
    WHERE{?person onto:birthPlace db:London .
          ?person onto:birthPlace ?location }
    }
    {SELECT ?date
    WHERE{?person onto:birthDate ?date . }
    }
    {SELECT ?name
    WHERE{ ?person foaf:name ?name }
    }
   }
LIMIT 10
```

6.1.11.6 Projected Expressions

You can use projected expressions within SPARQL `SELECT` queries to project arbitrary SPARQL expressions, rather than only bound variables. This allows the creation of new values in a query.

This type of query uses values derived from a variable, constant IRIs, constant literal, function calls, or other expressions in the `SELECT` list for columns in a query result set.

> **Note:** Functions could include both SPARQL built-in functions and extension functions supported by an implementation.

Projected expressions must be in parentheses and must be given an alias using the `AS` keyword. The syntax is (`expression AS ?var`).

For example :

```
PREFIX ex: <http://example.org/>

SELECT ?Item (?price * ?qty AS ?total_price)
WHERE {
  ?Item ex:price ?price.
  ?Item ex:quantity ?qty
}
```

The query returns values for `?total_price` that do not occur in the graphs contained in the RDF dataset.

6.1.12 De-Duplication of SPARQL Results

MarkLogic has implemented `dedup=on` and `dedup=off` options to `sem:sparql()`. Here are some examples of how deduplication works, based on a simple `sem:sparql()` example.

First, insert the same triple twice:

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

(: load an rdf triple that will match the SPARQL query :)

sem:rdf-insert(
 sem:triple(sem:iri("http://www.example.org/dept/108/invoices/20963"),
  sem:iri("http://www.example.org/dept/108/invoices/paid"),
  "true") ,
  xdmp:default-permissions(),
  "test-dedup") ;

(: returns the URI of the document that contains the triple :)

import module namespace sem = "http://marklogic.com/semantics"
  at "/MarkLogic/semantics.xqy";

(: load an rdf triple that will match the SPARQL query :)

sem:rdf-insert(
  sem:triple(sem:iri("http://www.example.org/dept/108/invoices/20963")
,
  sem:iri("http://www.example.org/dept/108/invoices/paid"),
  "true") ,
  xdmp:default-permissions(),
  "test-dedup") ;

(: returns the URI of the document that contains the triple :)
```

Then use a SPARQL query with `dedup=off`:

```
sem:sparql('
PREFIX inv: <http://www.example.org/dept/108/invoices/>

SELECT ?predicate ?object
WHERE
{ inv:20963 ?predicate ?object }
' ,
(),
"dedup=off" )
=>
<http://www.example.org/dept/108/invoices/paid> "true"
<http://www.example.org/dept/108/invoices/paid> "true"
```

Two identical triples are returned.

This SPARQL query uses `dedup=on`, which is the default:

```
sem:sparql('
PREFIX inv: <http://www.example.org/dept/108/invoices/>

SELECT ?predicate ?object
WHERE { inv:20963 ?predicate ?object }
' ,
(),
 "dedup=on" )
=>
<http://www.example.org/dept/108/invoices/paid> "true"
```

Only one instance of the triple is returned.

The `dedup=on` option is the default, standards-compliant behavior. The `dedup=off` option for `sem:sparql` may well give the same results if you never insert duplicate triples, but it entails a considerable performance overhead (for example, with filtering in search), so it's important to consider using this option.

6.1.13 Property Path Expressions

Property paths enable you to traverse an RDF graph. You can follow possible routes through a graph between two graph nodes. You can use property paths to answer questions like "show me all of the people who are connected to John, and all the people who know people who know John." You can use property paths to query paths of any length in a dataset graph by using an XPath-like syntax. A property path query retrieves pairs of connecting nodes where the paths that link those nodes match the given property path. This makes it easier to follow and use relationships expressed as triples.

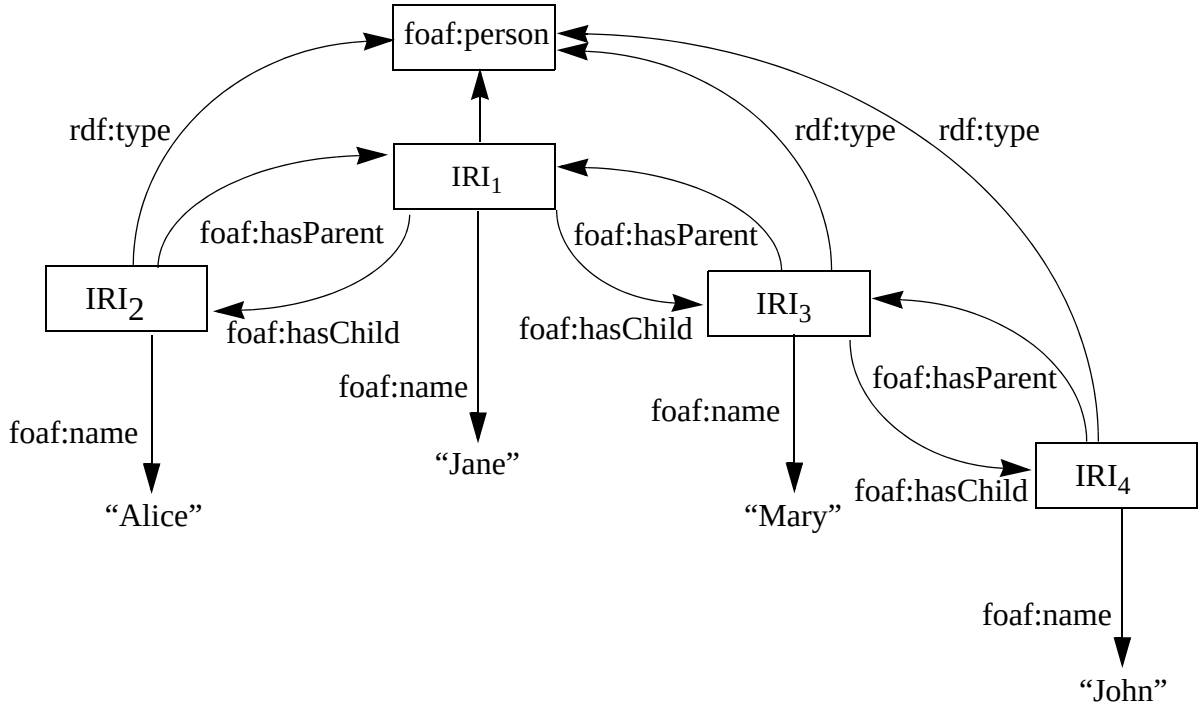
Query evaluation determines all matches of a path expression and binds subject or object as appropriate. Only one match per route through the graph is recorded - there are no duplicates for any given path expression.

6.1.13.1 Enumerated Property Paths

The following table describes the supported enumerated path operators (|, ^, and /) that can be combined with predicates in a property path:

Property Path	Construct	Description
Sequence	`path1/path2`	Forwards path from path1 to path2
Inverse	`^path`	Backwards path from object to subject
Alternative	`path1|path2`	Either path1 or path2
Group	`(path)`	A group path `path`, brackets control precedence

The following examples illustrate property paths using this simple graph model:



Here is that same graph model expressed as triples in Turtle format:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns> .
@prefix p0: <http://marklogic.com/semantics/> .

p0:alice foaf:hasParent p0:jane ;
  a foaf:Person ;
  foaf:name "Alice" .

p0:jane foaf:hasChild p0:alice,
    p0:mary;
  a foaf:Person ;
  foaf:name "Jane" .

p0:mary foaf:hasParent p0:jane ;
  a foaf:Person ;
  foaf:hasChild p0:john ;
  foaf:name "Mary" .

p0:john foaf:hasParent p0:mary ;
  a foaf:Person ;
  foaf:name "John".
```

This example query uses paths (the / operator) to find the name of Alice's parent:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?s ?name
  WHERE {?s foaf:name "Alice".
    ?s foaf:hasParent/foaf:name ?name .
  }

=>
  s       name
<http://marklogic.com/semantics/alice> "Jane"
```

This query finds the names of people two links away from John (his grandparent):

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?s ?name
  WHERE {?s foaf:name "John".
    ?s foaf:hasParent/foaf:hasParent/foaf:name ?name .
  }

=>
  s     name
<http://marklogic.com/semantics/john> "Jane"
```

This query reverses the property path direction (swaps the roles of subject and object using the ^ operator) to find the name of Mary's mother:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?s
  WHERE { <http://marklogic.com/semantics/mary> ^foaf:hasChild ?s }

=>
  s
<http://marklogic.com/semantics/Jane>
```

6.1.13.2 Unenumerated Property Paths

Unenumerated paths enable you to query triple paths and discover relationships, along with simple facts. This table describes the unenumerated path operators (+, *, or ?) that can be combined with predicates in a property path:

Property Path	Construct	Description
One or more	`path+`	A path that connects the subject and the object of the path by one or more matches of a path element.

Property Path	Construct	Description
Zero or more	`path*`	A path that connects the subject and the object of the path by zero or more matches of a path element.
Zero or one	`path?`	A path that connects the subject and the object of the path by zero or one matches of a path element

Note: A path element may itself be composed of path constructs.

The inverse operator (^) can be used with the enumerated path operators. Precedence of these operators is left-to-right within groups.

For these next examples, we can use `sem:rdf-insert` to add these triples to express the concept of `foaf:knows`:

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

let $string := '
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix p0: <http://marklogic.com/semantics/> .

p0:alice foaf:knows p0:jane .

p0:jane foaf:knows p0:mary,
  p0:alice .

p0:mary foaf:knows p0:john,
  p0:jane .

p0:john foaf:knows p0:mary .'

return sem:rdf-insert(sem:rdf-parse($string, "turtle"))
```

To find the names of all the people who are connected to Mary, use `foaf:knows` with the "+" path operator:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?s ?name
  WHERE {?s foaf:name "Mary" .
    ?s foaf:knows+/foaf:name ?name .}

=>
  s                                          name
<http://marklogic.com/semantics/mary>    "Jane"
<http://marklogic.com/semantics/mary>    "John"
```

```
<http://marklogic.com/semantics/mary>    "Mary"
<http://marklogic.com/semantics/mary>    "Alice"
```

This query will match all of the triples connected to Mary by `foaf:knows` where one or more paths exist. You can use `foaf:knows` with the "*" operator to find the names of anyone who is connected to Mary (including Mary) by zero or more paths.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?s ?name
  WHERE {?s foaf:name "Mary" .
    ?s foaf:knows*/foaf:name ?name .}
```

In this case the results will be same as in the previous example because the number of people connected to Mary by zero or more paths (the "*" path operator) is the same as the number connected by one or more paths.

Using the "?" operator finds the triples connected to Mary by one path element.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?s ?name
  WHERE {?s foaf:name "Mary" .
    ?s foaf:knows?/foaf:name ?name .}

=>
  s                                      name
<http://marklogic.com/semantics/mary>  "Jane"
<http://marklogic.com/semantics/mary>  "John"
<http://marklogic.com/semantics/mary>  "Mary"
```

You can also use a property path sequence to discover connections between triples.

For example, this query will find triples connected to Mary by three path elements:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?s ?name
WHERE {
?s foaf:name "Mary" .
?s foaf:knows/foaf:knows/foaf:knows/foaf:name ?name .
}

 s                                          name
<http://marklogic.com/semantics/mary>    "John"
<http://marklogic.com/semantics/mary>    "Jane"
<http://marklogic.com/semantics/mary>    "John"
<http://marklogic.com/semantics/mary>    "Jane"
```

The duplicate results are due to the different paths traversed by the query. You could add a `DISTINCT` keyword in the `SELECT` clause to return only one instance of each result and elimate the duplicates.

Note: The SPARQL modifier "`!`" has not been implemented in MarkLogic. Using this modifier to invert a property path value results in a syntax error.

You can combine SPARQL queries using property paths with a `cts:query` parameter to restrict results to only some documents (a combination query).

This combination query will find all the people connected to Alice who have children:

```
PREFIX cts: <http://marklogic.com/cts#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?s ?name
WHERE {
  ?s foaf:name "Mary" .
  ?s foaf:knows+/foaf:name ?name .
    ?s ?p ?o .
FILTER cts:contains(?p, cts:word-
query("http://xmlns.com/foaf/0.1/hasChild"))
       }
=>
<http://marklogic.com/semantics/mary> "Alice"
<http://marklogic.com/semantics/mary> "Jane"
<http://marklogic.com/semantics/mary> "John"
<http://marklogic.com/semantics/mary> "Mary"
```

You could also use a `cts:query` parameter to restrict the query to a collection or directory.

6.1.13.3 Inference

You can use unenumerated paths to do simple inference using thesaural relationships. (A thesaural relationship is a simple ontology).

For example, you can infer all the possible types of a resource, including supertypes of resources using this pattern:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?x ?type
{
   ?x rdf:type/rdfs:subClassOf* ?type
}
```

For example, this query will find the products that are subclasses of "shirt":

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ex: <http://example.com>

SELECT ?product
WHERE
```

```
  {
    ?product rdf:type/rdfs:subClassOf* ex:Shirt ;
}
```

For more about inference, see “Inference” on page 147.

6.1.14 SPARQL Aggregates

You can do simple analytic queries over triples using SPARQL aggregate functions. An aggregate function performs an operation over values or value co-occurrences in triples.

For example, you can use an aggregate function to compute the sum of values. This SPARQL query uses `SUM` to find the total sales:

```
PREFIX demov: <http://demo/verb/>
PREFIX vcard: <http://www.w3.org/2006/vcard/ns/>

SELECT (SUM (?sales) as ?sum_sales)
FROM <http://marklogic.com/semantics/COMPANIES100/>
WHERE {
?company a vcard:Organization .
?company demov:sales ?sales
}
```

These SPARQL aggregate functions are supported:

Aggregate Function	Example
`COUNT`	`SELECT (COUNT (?company) as ?count_companies)` `Count of “companies”`
`SUM`	`SELECT (SUM (?sales) as ?sum_sales)`
`MIN`	`SELECT (MIN (?sales) as ?min_sales)`
`MAX`	`SELECT ?country ( MAX (?sales) AS ?max_sales )`
`AVG`	`SELECT ?industry ( ROUND( AVG (?employees) ) AS ?avg_employees )`
Grouping Operations:	All aggregate functions are supported with `GROUP BY`

`GROUP BY`	`GROUP BY ?industry`
	`or`
	`GROUP BY ?country ?industry`

`GROUP BY <some_aggregate_variable>`	`GROUP BY AVG`
`GROUP BY. . HAVING <some_aggregate_variable>`	`GROUP BY ?industry` `HAVING (?sum_sales > 3000000000 )`

Aggregate Function	Example

GROUP CONCAT<more_than_one_item>	SELECT
	  ?region
	  (GROUP_CONCAT(DISTINCT ?industry ; separator=" + ") AS ?industries)

SAMPLE	SELECT ?country (SAMPLE(?industry) AS ?sample_industry) (SUM (?sales) AS ?sum_sales) SAMPLE is required for proper evaluation of unaggregated variables

Here is a SPARQL query using the aggregate function COUNT over a large number of triples:

```
PREFIX demor: <http://demo/resource/>
PREFIX demov: <http://demo/verb/>
PREFIX vcard: <http://www.w3.org/2006/vcard/ns/>

# count the companies
# (more precisely, count things of type organization)

(SELECT ( COUNT (?company) AS ?count_companies )

FROM <http://marklogic.com/semantics/test/COMPANIES100/>
WHERE {
  ?company a vcard:Organization .

}=>
100
```

Here is another example using COUNT and ORDER BY DESC:

```
PREFIX demor: <http://demo/resource/>
PREFIX demov: <http://demo/verb/>
PREFIX vcard: <http://www.w3.org/2006/vcard/ns/>

SELECT DISTINCT ?object (COUNT(?subject) AS ?count)
WHERE {
  ?subject <http://www.w3.org/1999/02/22-rdf-syntax-ns#type/> ?object
}
GROUP BY ?object
ORDER BY DESC (?count)
LIMIT 10
```

This query uses aggregates (MAX) to find the baseball player with the highest uniform number, and then get all the triples that pertain to him (or her). It uses an arbitrary triple (bb:number) that it knows every player in the dataset has, stores the subject in ?key, then queries for all triples and filters out where the subject in the outer query matches the ?key value:

```
PREFIX bb:  <http://marklogic.com/baseball/players/>
PEFIX bbr:  <http://marklogic.com/baseball/rules/>
```

```
PREFIX xs: <http://www.w3.org/2001/XMLSchema#>

SELECT *
FROM <Athletics>
{
  ?s ?p ?o .
  {
     SELECT(MAX(?s1) as ?key)
     WHERE
     {
       ?s1 bb:number ?o1 .
     }
  }
  FILTER (?s = ?key)
}
ORDER BY ?p
```

This complex nested query uses `COUNT AVG` to find the ten cheapest vendors for a specifc product type, selected by the highest percentage of their product below the average cost, and then filters for vendors containing either "`name1`" or "`name2`":

```
PREFIX bsbm: <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX bsbm-inst: <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX cts: <http://marklogic.com/cts#>

SELECT ?vendor (xsd:float(?belowAvg)/?offerCount As
?cheapExpensiveRatio)
 {
  { SELECT ?vendor (count(?offer) As ?belowAvg)

     {
    { ?product a <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductType459> .
    ?offer bsbm:product ?product .
    ?offer bsbm:vendor ?vendor .
    ?offer bsbm:price ?price .
    { SELECT ?product (avg(xsd:float(xsd:string(?price))) As ?avgPrice)
     {
      ?product a <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
        instances/ProductType459> .
      ?offer bsbm:product ?product .
      ?offer bsbm:vendor ?vendor .
      ?offer bsbm:price ?price .
     }
     GROUP BB ?product
    }
   } .
  FILTER (xsd:float(xsd:string(?price)) < ?avgPrice)
  }
   GROUP BY ?vendor
```

```
  }
  { SELECT ?vendor (count(?offer) As ?offerCount)
   {
    ?product a <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
      instances/ProductType459> .
    ?offer bsbm:product ?product .
    ?offer bsbm:vendor ?vendor .
   }
   GROUP BY ?vendor
  }
  FILTER cts:contains(?vendor, cts:or-query(("name1", "name2")))
}
ORDER BY desc(xsd:float(?belowAvg)/?offerCount) ?vendor
LIMIT 10
```

6.1.15 Using the Results of sem:sparql

Here is an example of using the results of `sem:sparql` in a query:

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

cts:search(
  fn:doc(),
  cts:triple-range-query(
    (), "takenIn",
    (: Use sem:sparql to run a query, then use the ! operator to
     : convert the solution to a sequence of strings
     :)
    sem:sparql(
      'select ?countryIRI
       {
         ?continentIRI <http://www.w3.org/2004/02/skos/core#prefLabel>
?continentLabel .
         ?countryIRI <http://dbpedia.org/property/continent>
?continentIRI .
       }',
      map:entry("continentLabel", rdf:langString("Countries in South
America", "en"))
    ) ! map:get(., "countryIRI")
  ))
```

6.1.16 SPARQL Resources

The SPARQL recommendation is closely related to these specifications:

- The SPARQL Protocol for RDF [SPROT] specification defines the remote protocol for issuing SPARQL queries and receiving the results.
  http://www.w3.org/TR/rdf-sparql-protocol/
- MarkLogic supports simple entailment, as described in the W3C recommendation:
  http://www.w3.org/TR/rdf-mt/#entail

- The SPARQL Query Results XML Format specification defines an XML document format for representing the results of SPARQL `SELECT` and `ASK` queries. http://www.w3.org/TR/rdf-sparql-XMLres/
- SPARQL 1.1 Graph Store HTTP Protocol: http://www.w3.org/TR/2012/CR-sparql11-http-rdf-update-20121108/

There are a variety of tutorials available for learning more about the SPARQL query language. For example:

- http://www.cambridgesemantics.com/semantic-university
- https://jena.apache.org/tutorials/sparql.html

Recommended reading:

- *Learning SPARQL* by Bob DuCharme (Publisher: O'Reilly)
- *Semantic Web for the Working Ontologist* by Dean Allemang and Jim Hendler (Publisher: Morgan Kaufmann)

Additional useful resources include:

- SPARQL Implementations: http://www.w3.org/wiki/SparqlImplementations
- SPARQL Working Group: http://www.w3.org/2009/sparql/wiki/Main_Page
- SPARQL query results - JSON format: http://www.w3.org/TR/2012/PR-sparql11-results-json-20121108/
- SPARQL Frequently Asked Questions: http://thefigtrees.net/lee/sw/sparql-faq

6.2 Querying Triples with XQuery or JavaScript

This section contains examples of using XQuery or JavaScript with semantic data. When you use JavaScript or XQuery to query triples in MarkLogic, you can use the Semantics API library, built-in functions, the Search API built-in functions, or a combination of these.

This section includes the following topics:

- Preparing to Run the Examples
- Using Semantic Functions to Query
- Using Bindings for Variables
- Viewing Results as XML and RDF
- Working with CURIEs
- Using Semantics with cts Searches

6.2.1 Preparing to Run the Examples

These examples for querying triples with XQuery or Javascript assume that you have the GovTrack dataset stored on Archive.org. If you prefer to use your own dataset or cannot access https://bit.ly/2EGlo9j, you can skip this section.

This data is free, publicly available legislative information about bills in the US Congress, representatives, and voting records. The information originates from a variety of official government Web sites. The `Govtrack.us` data applies the principles of open data to legislative transparency.

Before installing the GovTrack dataset, make sure you have the following:

- MarkLogic Server 8.0-4 or later.
- MarkLogic Content Pump (mlcp). See Installation and Configuration in the *mlcp User Guide*
- The GovTrack dataset from Archive.org and access to https://bit.ly/2EGlo9j.

Follow this procedure to download the GovTrack dataset and load it into MarkLogic Server.

1. Download the following files into a directory on your local file system:

   - https://bit.ly/2tIRHhq – `bills.108.cosponsors.rdf.gz`
   - https://bit.ly/2VsQJBw – `bills.108.rdf.gz`
   - https://bit.ly/2IS0wA2 – `people.rdf.gz`
   - http://bit.ly/2NB5OOT – `people.roles.rdf.gz`

2. Create a `govtrack` database and forest. For these examples you can use the application server on port 8000 with the GovTrack data. This default server can function as an XDBC server and REST instance as well.

   To create your own XDBC server and REST instance see Setting Up Additional Servers in this guide and Administering REST Client API Instances in the in the *REST Application Developer's Guide* for more information.

3. Verify that the triples index and the collection lexicon are enabled for the `govtrack` database. See "Enabling the Triple Index" on page 66.

4. Import the data into your `govtrack` database with mlcp, specifying the collections of `info:govtrack/people` and `info:govtrack/bills`. See "Loading Triples with mlcp" on page 44. Your import command on Windows will look similar to the following:

```
mlcp.bat import -host localhost -port 8000 -username admin ^
  -password password -database govtrack –input_file_type rdf ^
  -input_file_path c:\space\GovTrack -input_compressed true^
```

```
-input_compression_codec gzip ^
-output_collections "info:govtrack/people,info:govtrack/bills"
```

Modify the `host`, `port`, `username`, `password`, and `-input_file_path` options to match your environment. In this example, long lines have been broken for readability and Windows continuation characters ("^") have been added.

Note: Be sure to add the `-database` parameter to the command. If you leave this parameter out, the data will go into the default Documents database.

The equivalent command for UNIX is:

```
mlcp.sh import -host localhost -port 8000 -username admin \
-password password -database govtrack -input_file_type RDF \
-input_file_path /space/GovTrack -input_compressed true \
-input_compression_codec gzip \
-output_collections 'info:govtrack/people,info:govtrack/bills'
```

In this example, the long lines have been broken and the UNIX continuation characters ("\") have been added.

Note: It is important to specify the `-input_file_type` as RDF to invoke the correct parser.

6.2.2 Using Semantic Functions to Query

You can execute SPARQL `SELECT`, `ASK`, and `CONSTRUCT` queries with the `sem:sparql` and `sem:sparql-values` functions in XQuery, and with the `sem.sparql` and `sem.sparqlValues` functions in Javascript. For details about the function signatures and descriptions, see the Semantics documentation under XQuery Library Modules in the *MarkLogic XQuery and XSLT Function Reference*.

The following examples execute SPARQL queries against the triples index of the `govtrack` database. See "Preparing to Run the Examples" on page 129.

Note: Although some of the semantics functions are built-in, others are not, so we recommend that you import the Semantics API library into every XQuery module or JavaScript module that uses the Semantics API.

Using XQuery, the import statement is:

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

For Javascript, the import statement is:

```
var sem = require("/MarkLogic/semantics.xqy");
```

6.2.2.1 sem:sparql

You can use the `sem:sparql` function to query RDF data in the database in the same way you would in the SPARQL language. To use `sem:sparql`, you pass the SPARQL query to the function as a string.

Using XQuery the query would look like:

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

sem:sparql('
PREFIX bill: <http://www.rdfabout.com/rdf/usgov/congress/108/bills/>
SELECT ?predicate ?object
WHERE { bill:h963 ?predicate ?object }
')
```

Using Javascript, the query would be:

```
var sem = require("/MarkLogic/semantics.xqy");

sem.sparql( +
'PREFIX bill: <http://www.rdfabout.com/rdf/usgov/congress/108/bills/>' +
'SELECT ?predicate ?object' +
'WHERE { bill:h963 ?predicate ?object }' )
```

Note: In JavaScript, you must either use a left-quote (“`”) at the beginning of a literal string that spans multiple lines. Otherwise, you must use a “+” or “\” to concatenate the substrings.

The XQuery code returns an array as a sequence, whereas the JavaScript code returns a Sequence. See Sequence in the *JavaScript Reference Guide* for more information.

The result of the example query for all triples where the subject is bill number “h963” would look like this:

predicate	object
<http://www.rdfabout.com/rdf/schema/usbill/hasTitle>	<-276ac564:14008b22ed1:-3313>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.rdfabout.com/rdf/schema/usbill/HouseBill>
<http://www.rdfabout.com/rdf/schema/usbill/inCommittee>	<http://www.rdfabout.com/rdf/usgov/congress/committees/HouseGovernmentReform>
<http://www.rdfabout.com/rdf/schema/usbill/cosponsor>	<http://www.rdfabout.com/rdf/usgov/congress/people/D000598>
<http://www.rdfabout.com/rdf/schema/usbill/sponsor>	<http://www.rdfabout.com/rdf/usgov/congress/people/F000116>
<http://www.rdfabout.com/rdf/schema/usbill/congress>	"108"
<http://purl.org/dc/terms/created>	"2003-02-27"
<http://www.rdfabout.com/rdf/schema/usbill/number>	"963"
<http://purl.org/dc/elements/1.1/title>	"H.R. 108/963: To redesignate the facility of the United States Postal Service located at 2777 Loga \"Cesar E. Chavez Post Office\"."
<http://purl.org/ontology/bibo/shortTitle>	"H.R. 963: To redesignate the facility of the United States Postal Service located at 2777 Logan A \"Cesar E. Chavez Post Office\"."
<http://www.rdfabout.com/rdf/schema/usbill/title>	"H.R. 963: To redesignate the facility of the United States Postal Service located at 2777 Logan A \"Cesar E. Chavez Post Office\"."
<http://www.w3.org/2000/01/rdf-schema#label>	"H.R. 963: To redesignate the facility of the United States Postal Service located at 2777 Logan A \"Cesar E. Chavez Post Office\". (108th Congress)"
<http://www.rdfabout.com/rdf/schema/usbill/type>	"h"
<http://www.rdfabout.com/rdf/schema/usbill/status>	"introduced"
<http://www.rdfabout.com/rdf/schema/usbill/introduced>	"2003-02-27"^^xs:date

For more information about constructing SPARQL queries, see “Constructing a SPARQL Query” on page 87.

You can also construct your SPARQL query as an input string in a FLWOR statement. In the following example, the `let` statement contains the SPARQL query. This is a SPARQL `ASK` query, to find out if there are any male politicians who are members of the Latter Day Saints:

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

let $sparql := '
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX politico: <http://www.rdfabout.com/rdf/schema/politico/>
PREFIX govtrack: <http://www.rdfabout.com/rdf/schema/usgovt/>
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0/>

ASK  { ?x rdf:type  politico:Politician ;
       foaf:religion "Latter Day Saints" ; foaf:gender "male". }
'
return sem:sparql($sparql)

=>
true
```

6.2.2.2 sem:sparql-values

Use the `sem:sparql-values` function to allow sequences of bindings to restrict what a SPARQL query returns. In this example, a sequence of values are bound to the subject IRIs that represent two members of congress.

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

let $bindings := ( map:entry("s",
sem:iri("http://www.rdfabout.com/rdf/usgov/congress/people/A000069")),
map:entry("s",
sem:iri("http://www.rdfabout.com/rdf/usgov/congress/people/G000359"))
)
return
sem:sparql-values("select * { ?s ?p ?o }",$bindings)
```

The results are returned as sequences of values for the two members of congress:

s	p	o
<http://www.rdfabout.com/rdf/usgov/congress/people/A000069>	<http://www.w3.org/2001/vcard-rdf/3.0#N>	<-276ac564:14008b2
<http://www.rdfabout.com/rdf/usgov/congress/people/A000069>	<http://www.rdfabout.com/rdf/schema/politico/hasRole>	<-7059da6e:1403c27
<http://www.rdfabout.com/rdf/usgov/congress/people/A000069>	<http://www.rdfabout.com/rdf/schema/politico/hasRole>	<-7059da6e:1403c27
<http://www.rdfabout.com/rdf/usgov/congress/people/A000069>	<http://www.rdfabout.com/rdf/schema/politico/hasRole>	<-7059da6e:1403c27
<http://www.rdfabout.com/rdf/usgov/congress/people/A000069>	<http://xmlns.com/foaf/0.1/img>	<http://www.govtrack
<http://www.rdfabout.com/rdf/usgov/congress/people/A000069>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.rdfabout
<http://www.rdfabout.com/rdf/usgov/congress/people/A000069>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://xmlns.com/foa
<http://www.rdfabout.com/rdf/usgov/congress/people/A000069>	<http://www.w3.org/2001/vcard-rdf/3.0#BDAY>	"1924-09-11"
<http://www.rdfabout.com/rdf/usgov/congress/people/A000069>	<http://www.rdfabout.com/rdf/schema/usgovt/congressBioGuideID>	"A000069"
<http://www.rdfabout.com/rdf/usgov/congress/people/A000069>	<http://xmlns.com/foaf/0.1/religion>	"Congregationalist"
<http://www.rdfabout.com/rdf/usgov/congress/people/A000069>	<http://www.rdfabout.com/rdf/schema/usgovt/name>	"Daniel Akaka"
<http://www.rdfabout.com/rdf/usgov/congress/people/A000069>	<http://xmlns.com/foaf/0.1/name>	"Daniel Akaka"
<http://www.rdfabout.com/rdf/usgov/congress/people/A000069>	<http://xmlns.com/foaf/0.1/title>	"Sen."

The `sem:sparql-values` function can be considered as equivalent to the SPARQL 1.1 facility of an outermost `VALUES` block. See “Values Sections” on page 111 for more information.

Everywhere you use a variable in a SPARQL values query, you can set the variable to a fixed value by passing in external bindings as arguments to `sem:sparql-values`. See “Using Bindings for Variables” on page 135.

6.2.2.3 sem:store

The `sem:store` function contains a set of criteria used to select the set of triples to be passed in to `sem:sparql`, `sem:sparql-values`, or `sem:sparql-update` and evaluated as part of the query. The triples included in `sem:store` come from the current database's triple index, restricted by the options and the `cts:query` argument in `sem:store` (for instance, "all triples in documents matching this query"). If multiple `sem:store` constructors are supplied, the triples from all the sources are merged and queried together.

If a `sem:store` constructor is not supplied as an option for `sem:sparql`, `sem:sparql-values`, or `sem:sparql-update`, then the default `sem:store` constructor for the query will be used (the default database's triple index).

6.2.2.4 Querying Triples in Memory

You can use `sem:in-memory-store` to query triples in memory.

For example:

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

let $triples := sem:rdf-parse($string, ("turtle", "myGraph"))
let $query := '
PREFIX ad: <http://marklogic.com/addressbook/>
PREFIX d:  <http://marklogic.com/id/>

CONSTRUCT{ ?person ?p ?o .}
FROM <myOtherGraph>
WHERE
{
  ?person ad:firstName "Elvis" ;
  ad:lastName  "Presley" ;
  ?p ?o .
}
'
for $result in sem:sparql($query, (), (), sem:in-memory-
store($triples))
order by sem:triple-object($result)
return <result>{$result}</result>
```

This query constructs a graph of triples in memory named "myGraph" containing persons named Elvis with a last name of Presley. The source of these triples is "myOtherGraph" and the results are returned in order.

6.2.3 Using Bindings for Variables

Extensions to standard SPARQL enable you to use bindings for variables in the body of a query statement. Everywhere you use a variable in a SPARQL query, you can set the variable to a fixed value by passing in external bindings as arguments to `sem:sparql`.

Bindings for variables can also be used as values in `OFFSET` and `LIMIT` clauses (in the syntax where they previously were not allowed). This example query uses bindings for variables with both `LIMIT` and `OFFSET`.

```
xquery version "1.0-ml";
import module namespace sem = "http://marklogic.com/semantics"
  at "/MarkLogic/semantics.xqy";
import module namespace json = "http://marklogic.com/xdmp/json"
  at "MarkLogic/json/json.xqy";
declare namespace jbasic = "http://marklogic.com/xdmp/json/basic";

let $query := '
PREFIX bb: <http://marklogic.com/baseball/players/>

SELECT ?firstname ?lastname ?team
FROM <SportsTeams>
{
 {
  ?s bb:firstname ?firstname .
  ?s bb:lastname ?lastname .
  ?s bb:team ?team .
  ?s bb:position ?position .
FILTER (?position = ?pos)
 }
}
ORDER BY ?lastname
LIMIT ?lmt
'
let $mymap := map:map()
let $put := map:put($mymap, "pos", "pitcher")
let $put := map:put($mymap, "lmt", "3")
let $triples := sem:sparql($query, $mymap)
let $triples-xml := sem:query-results-serialize($triples, "xml")
return <results>{$triples-xml}</results>

=>
<results>
 <sparql xmlns="http://www.w3.org/2005/sparql-results/">
  <head>
    <variable name="firstname"></variable>
    <variable name="lastname"></variable>
    <variable name="team"></variable>
  </head>
 <results>
  <result>
   <binding name="firstname">
    <literal datatype="http://www.w3.org/2001/XMLSchema#string">
```

```
    Fernando</literal>
   </binding>
   <binding name="lastname">
    <literal datatype="http://www.w3.org/2001/XMLSchema#string">
    Abad</literal>
   </binding>
   <binding name="team">
    <literal datatype="http://www.w3.org/2001/XMLSchema#string">
    Athletics</literal>
   </binding>
  </result>
  <result>
   <binding name="firstname">
    <literal datatype="http://www.w3.org/2001/XMLSchema#string">
    Jesse</literal>
   </binding>
   <binding name="lastname">
    <literal datatype="http://www.w3.org/2001/XMLSchema#string">
   Chavez</literal>
   </binding>
   <binding name="team">
    <literal datatype="http://www.w3.org/2001/XMLSchema#string">
    Athletics</literal>
   </binding>
  </result>
  <result>
   <binding name="firstname">
    <literal datatype="http://www.w3.org/2001/XMLSchema#string">
    Ryan</literal>
   </binding>
   <binding name="lastname">
    <literal datatype="http://www.w3.org/2001/XMLSchema#string">
    Cook</literal>
   </binding>
   <binding name="team">
    <literal datatype="http://www.w3.org/2001/XMLSchema#string">
    Athletics</literal>
   </binding>
  </result>
 </results>
 </sparql>
</results>
```

Bindings can be used with SPARQL (`sem:sparql`), SPARQL values (`sem:sparql-values`), and SPARQL Update (`sem:sparql-update`). See “Bindings for Variables” on page 184 for an example of bindings for variables used with SPARQL Update.

6.2.4 Viewing Results as XML and RDF

You can use `sem:query-results-serialize` and `sem:rdf-serialize` functions to view results in XML, JSON, or RDF serialization.

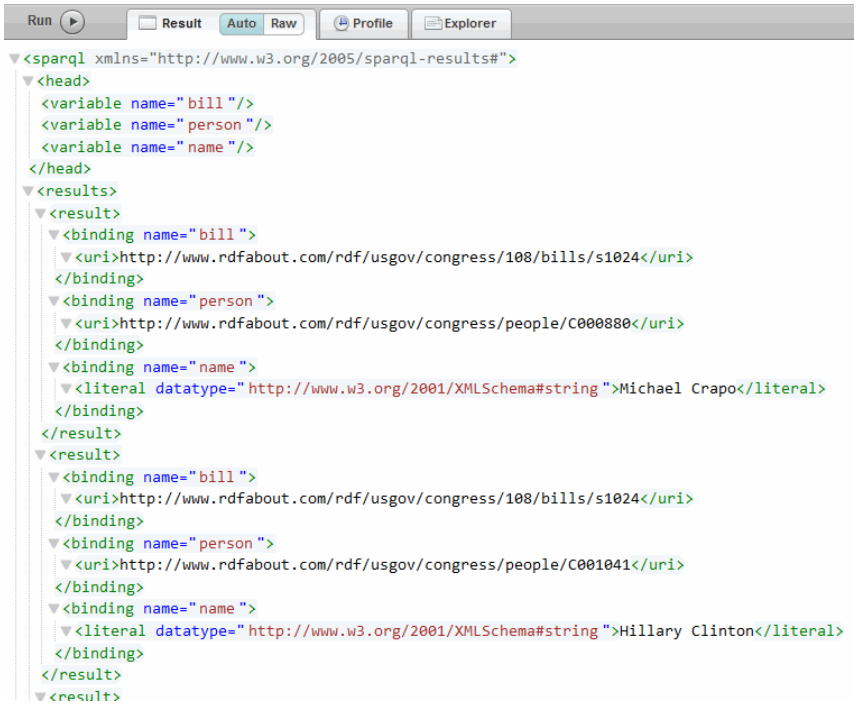
In this example, the `sem:sparql` query finds the cosponsors of bill number "1024" and passes the value sequence into `sem:query-results-serialize` to return the results as variable bindings in default XML format:

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

sem:query-results-serialize(sem:sparql('
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX bill: <http://www.rdfabout.com/rdf/schema/usbill/>

SELECT ?bill ?person ?name
WHERE {?bill rdf:type bill:SenateBill ;
             bill:congress "108" ;
             bill:number "1024" ;
             bill:cosponsor ?person .
       ?person foaf:name ?name .}
'))
```

The results are returned in W3C SPARQL Query Results format:



To view the same results in JSON serialization, add the format option after the query.

For example:

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

sem:query-results-serialize(sem:sparql('
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX bill: <http://www.rdfabout.com/rdf/schema/usbill/>

SELECT ?bill ?person ?name
WHERE {?bill rdf:type bill:SenateBill ;
             bill:congress "108" ;
             bill:number "1024" ;
             bill:cosponsor ?person .
      ?person foaf:name ?name .}
'), "json")
```

When you use the `sem:rdf-serialize` function, you pass the triple to return as a string, or optionally you can specify a parsing serialization option.

For example:

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

sem:rdf-serialize(
   sem:triple(
      sem:iri(
   "http://www.rdfabout.com/rdf/usgov/congress/people/D000060"),
      sem:iri("http://www.rdfabout.com/rdf/schema/usgovt/name"),
           "Archibald Darragh"), "rdfxml")
```

This table describes the serialization options available for the output:

Serialization	Output As
ntriple	`xs:string`
nquad	`xs:string`
turtle	`xs:string`
rdfxml	an element
rdfjson	a `json:object`
triplexml	a sequence of `sem:triple` elements

You can also select different ways to display results. See “Selecting Results Rendering” on page 86.

6.2.5 Working with CURIEs

A CURIE (Compact URI Expression) is a shortened version of a URI signifying a specific resource. With MarkLogic, lengthy IRIs can be shortened using a mechanism similar to that built into the SPARQL language. As a convenience, the definitions of several common prefixes are built in, as shown in the examples in this section.

CURIEs are composed of two components: a prefix, and a reference. The prefix is separated from the reference by a colon (:), for example, `dc:description` is a prefix for Dublin Core and the reference - `http://purl.org/dc/elements/1.1/` - is the description.

These are the most common prefixes and their mapping:

```
map:entry("atom", "http://www.w3.org/2005/Atom/"),
map:entry("cc", "http://creativecommons.org/ns/"),
map:entry("dc", "http://purl.org/dc/elements/1.1/"),
map:entry("dcterms", "http://purl.org/dc/terms/"),
map:entry("doap", "http://usefulinc.com/ns/doap/"),
map:entry("foaf", "http://xmlns.com/foaf/0.1/"),
map:entry("media", "http://search.yahoo.com/searchmonkey/media/"),
map:entry("og", "http://ogp.me/ns/"),
map:entry("owl", "http://www.w3.org/2002/07/owl/"),
map:entry("prov", "http://www.w3.org/ns/prov/"),
map:entry("rdf", "http://www.w3.org/1999/02/22-rdf-syntax-ns"),
map:entry("rdfs", "http://www.w3.org/2000/01/rdf-schema/"),
map:entry("result-set",
"http://www.w3.org/2001/sw/DataAccess/tests/result-set/"),
map:entry("rss", "http://purl.org/rss/1.0/"),
map:entry("skos", "http://www.w3.org/2004/02/skos/core/"),
map:entry("vcard", "http://www.w3.org/2006/vcard/ns/"),
map:entry("void", "http://rdfs.org/ns/void/"),
map:entry("xhtml", "http://www.w3.org/1999/xhtml/"),
map:entry("xs","http://www.w3.org/2001/XMLSchema#")
```

You can use the `sem:curie-expand` and `sem:curie-shorten` functions to work with CURIEs in MarkLogic. When you use `sem:curie-expand`, you eliminate the need to declare common prefixes.

For example:

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

sem:curie-expand("foaf:name")

=>

<http://xmlns.com/foaf/0.1/name>
```

In this example, the `cts:triple-range-query` finds a person named “Lamar Alexander”. Note that the results are returned from a `cts:search` to find the `sem:triple` elements where the `foaf:name` equals “Lamar Alexander”. The predicate CURIE is displayed as the fully expanded IRI for `foaf:name`.

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

let $query := cts:triple-range-query((), sem:curie-
expand("foaf:name"), "Lamar Alexander", "sameTerm")

return cts:search(fn:collection()//sem:triple, $query)

=>
<sem:triples xmlns="http://marklogic.com/semantics">
   <sem:subject>
      http://www.rdfabout.com/rdf/usgov/congress/people/A000360/
   </sem:subject>
   <sem:predicate>
      http://xmlns.com/foaf/0.1/name
   </sem:predicate>
   <sem:object datatype="http://www.w3.org/2001/XMLSchema#string">
      Lamar Alexander
   </sem:object>
</sem:triples>
```

In the following example, the query includes a series of `cts:triples` function calls and `sem:curie-expand` to find the name of the congressperson who was born on November 20, 1917. The person’s name is returned as an RDF literal string from the object position (`sem:triple-object`) of the returned triple statement:

```
xquery version "1.0-ml";

import module namespace sem = "http://marklogic.com/semantics"
  at "/MarkLogic/semantics.xqy";

let $person-triples := cts:triples((), sem:curie-expand("vcard3:BDAY",
map:entry("vcard3", "http://www.w3.org/2001/vcard-rdf/3.0/")),
"1917-11-20")
let $subject := sem:triple-subject($person-triples)
let $name-triples := cts:triples($subject,
sem:curie-expand("foaf:name"), ())
let $name := sem:triple-object($name-triples)
return ($name)

=>

Robert Byrd
```

Use the `sem:curie-shorten` to compact an IRI to a CURIE. Evaluating the function involves replacing the CURIE with a concatenation of the value represented by the prefix and the part after the colon (the reference).

For example:

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

sem:curie-shorten(sem:iri("http://www.w3.org/1999/02/
    22-rdf-syntax-ns#resource/"))

=>
rdf:resource
```

Note: Although CURIEs map to IRIs, do not use them as values for attributes or other content that are specified to contain only IRIs.

For example, the following query will return an empty sequence since the `cts:triple-range-query` expects an IRI (`sem:iri`) in that position not a `sem:curie-shorten`, which is a string:

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

let $query := cts:triple-range-query((),
   sem:curie-shorten(sem:iri("http://xmlns.com/foaf/0.1/name")),
   "Lamar Alexander", "sameTerm")

return cts:search(fn:collection()//sem:triple, $query)
```

Instead, either of the following can be used:

```
let $query := cts:triple-range-query((),
   sem:curie-expand("foaf:name"), "Lamar Alexander", "sameTerm")
```

Or alternatively expand the prefix to the full IRI:

```
let $query := cts:triple-range-query((),
   sem:iri("http://xmlns.com/foaf/0.1/name/"), "Lamar Alexander",
   "sameTerm")
```

Note: The `sameTerm` function that is defined in SPARQL, performs the value equality operation. It differs from the equality operator (`=`) in the way that types are handled. In MarkLogic, types and timezones are the only things that make `sameTerm` different from `=`. For example, `sameTerm(A,B)` implies A=B. In SPARQL terms, using `sameTerm` semantics to match graphs to the graph patterns in a SPARQL query is called *simple entailment*. For more information, see “Triple Values and Type Information” on page 64.

6.2.6 Using Semantics with cts Searches

This section discusses using cts searches to return RDF data from a MarkLogic triple store. It includes the following topics:

- cts:triples
- cts:triple-range-query
- cts:search
- cts:contains

6.2.6.1 cts:triples

The `cts:triples` function retrieves the parameter values from the triple index. Triples can be returned in any of the sort orders present in the triple index.

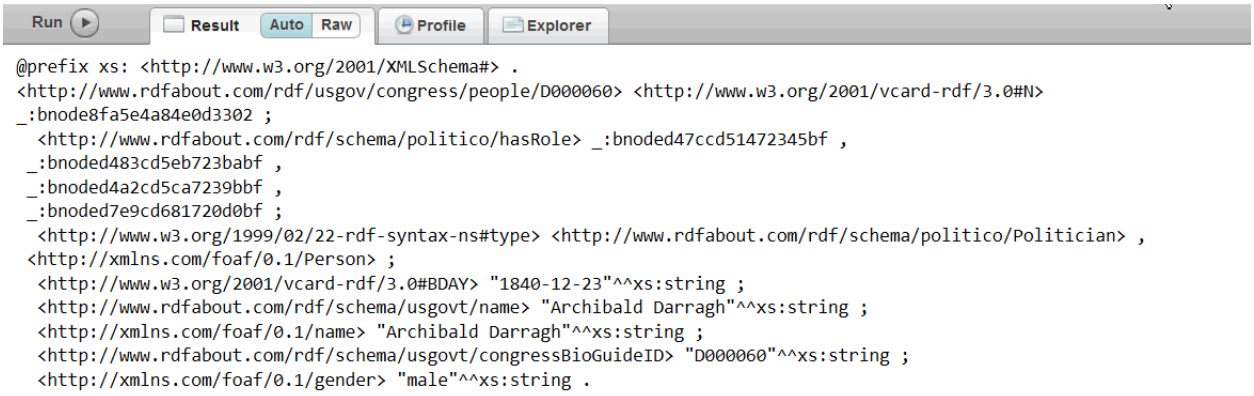
In this example, the subject IRI for a member of congress is passed as the first parameter for the subject IRI:

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

let $r := cts:triples
  (sem:iri(
    "http://www.rdfabout.com/rdf/usgov/congress/people/D000060"),
   )

return ($r)
```

The matching results return triples for that member of congress (Archibald Darragh):



6.2.6.2 cts:triple-range-query

Access to the triple index is provided through the `cts:triple-range-query` function. The first parameter in this example is an empty sequence for the subject. The predicate and object parameters are provided, along with the `sameTerm` operator to find someone named “Lamar Alexander”:

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

let $query := cts:triple-range-query((),
  sem:iri("http://xmlns.com/foaf/0.1/name"), "Lamar Alexander",
"sameTerm")

return cts:search(fn:collection()//sem:triple, $query)
```

6.2.6.3 cts:search

The built-in cts search functions are XQuery functions used to perform text searches. In this example, the `cts:search` queries against the `info:govtrack/bills` collection of XML docs to determine how many bills have the word “Guam” in the document (the `cts:word-query` of the specified string).

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

let $search := cts:search(//sem:triple,
  cts:and-query((cts:collection-query("info:govtrack/bills"),
cts:word-query("Guam"))
  )
 )[1]

return cts:remainder($search)

=>
16
```

You can use a combination of `cts:query` and comparison operators. The `cts:triple-range-query` function in this example is used within a `cts:search` to find the `sem:triple` elements, where the `foaf:name` equals “Lamar Alexander” or where Alexander’s subject IRI contains a `foaf:img` property conveying an image IRI.

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

declare namespace dc = "http://purl.org/dc/elements/1.1/";

cts:search(collection()//sem:triple, cts:or-query((
  cts:triple-range-query((), sem:curie-expand("foaf:name"),
   "Lamar Alexander", "sameTerm"),
  cts:triple-range-query(
```

```
    sem:iri
     ("http://www.rdfabout.com/rdf/usgov/congress/people/A000360"),
    sem:curie-expand("foaf:img"), (), "="
  )
)))
```

You can construct sequences in SPARQL expressions and the SPARQL 1.1 `IN` and `NOT IN` operators to make effective use of built-in cts functions such as `cts:and-query`, which expect a sequence of `cts:query` values as the first argument.

You can also use `cts:order` constructors as an option to `cts:search` to to specify an ordering. This lets you order cts search results using a specified index for better, predictable performance. See Creating a cts:order Specification in the *Query Performance and Tuning Guide*.

6.2.6.4 cts:contains

You can use the `cts:contains` function in SPARQL expressions, which occur in `FILTER` and `BIND` clauses. For an example, see “The FILTER Keyword” on page 102.

Since `cts:contains` allows any value as the first argument, you can pass a variable that is bound by a triple pattern in the query as the first argument. The triple pattern uses the full-text index to reduce the results it returns during the lookup in the triple index. For example:

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

fn:count(sem:sparql('
PREFIX cts: <http://marklogic.com/cts#>

SELECT DISTINCT *
WHERE
{ ?s ?p ?o .
     FILTER cts:contains(?o, cts:word-query("Environment")) }
   ')
  )
=>
53
```

The following example is a query to verify if there is a bill number “hr543”.

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

cts:contains(collection("info:govtrack/bills")//sem:subject,
  cts:word-query('hr543'))

=>
true
```

6.3 Querying Triples with the Optic API

The Optic API can also be used for server-side queries of triples. The following Optic example query returns a list of the people who were born in Brooklyn in the form of a table with two columns, `person` and `name`.

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

let $resource := op:prefixer("http://dbpedia.org/resource/")
let $foaf     := op:prefixer("http://xmlns.com/foaf/0.1/")
let $onto     := op:prefixer("http://dbpedia.org/ontology/")
let $person   := op:col("person")

return op:from-triples((
   op:pattern($person, $onto("birthPlace"), $resource("Brooklyn")),
   op:pattern($person, $foaf("name"), op:col("name"))))
   => op:result()
```

This query uses the same data set as the one used for queries earlier in this chapter (see “Querying Triples with SPARQL” on page 82). The results would look like this:


sql as Table

person	name
http://dbpedia.org/resource/A.E._Coleby	A.E. Coleby
http://dbpedia.org/resource/A.S._Douglas	Alexander Shafto Douglas
http://dbpedia.org/resource/A._A._Pearson	A. A. Pearson
http://dbpedia.org/resource/A._B._Campbell	A. B. Campbell
http://dbpedia.org/resource/A._E._W._Mason	A E W Mason
http://dbpedia.org/resource/A._Follett_Osler	A. Follett Osler
http://dbpedia.org/resource/A._J._Ayer	Alfred Ayer
http://dbpedia.org/resource/A._L._Lloyd	Albert Lancaster Lloyd
http://dbpedia.org/resource/A._M._W._Stirling	A.M.W. Stirling
http://dbpedia.org/resource/A._R._Rawlinson	A.R Rawlinson
http://dbpedia.org/resource/Aaron_Hart_(businessman)	Aaron Hart
http://dbpedia.org/resource/Aaron_Patton	Aaron Patton

For more about the Optic API, see Optic API for Multi-Model Data Access and Data Access Functions in the *Application Developer’s Guide* and `op:from-triples` or `op.fromTriples` in the Optic API for more about server-side queries using Optic.

6.4 Serialization

You can set the output serialization for results in a variety of ways. These options can be set at the query level as part of the JSON or XQuery function to override any default options, or you could set the method in an XQuery declaration, or the method can be configured in the app server. These output options affect how data returned from the App Server or sent over REST is serialized.

6.4.1 Setting the Output Method

You can set the output method for the results of your query in the following ways. Each method overrides the next method in the list:

- set an option to `xdmp:quote()`
- set `xdmp:set-response-output-method()`
- set the XSLT output method
- Use a static declaration in XQuery (or JavaScript)
- Configure the output in app server

In other words, any configuration you have set in the app server will be overwritten by a static declaration in XQuery or Javascript.

To set the output method in an XQuery declaration use:

```
declare option xdmp:output "method = sparql-results-json"
```

To set the output method as part of an XQuery function use:

```
set-response-output-method("sparql-results-json")
```

As part of a server-side JavaScript function use to set the output method:

```
setResponseOutputMethod("sparql-results-json")
```

6.5 Security

The triple index, `cts:triples`, and `sem:sparql` queries will only return triples from documents which the database user has permission to read.

Named graphs inherit the write protection settings available to collections.

Task	Privilege
Executing `sem:sparql`	http://marklogic.com/xdmp/privileges/sem-sparql

For more information about MarkLogic security, see Document Permissions in the *Security Guide*.

7.0 Inference

In the context of MarkLogic Semantics, and semantic technology in general, the process of "inference" involves the automated discovery of new facts based on a combination of data and rules for understanding that data. Inference is the process of "inferring" or discovering new facts about your data based on a set of rules. Inference with semantic triples means that automatic procedures can generate new relationships (new facts) from existing triples.

An inference query is any SPARQL query that is affected by automatic inference. The W3C specification describing inference, with links to related standards, can be found here: http://www.w3.org/standards/semanticweb/inference

New facts may be added to the database (forward-chaining inference), or they may be inferred at query time (backward chaining inference), depending on the implementation. MarkLogic supports automatic backward-chaining inference.

This chapter includes the following sections:

- Automatic Inference
- Other Ways to Achieve Inference
- Performance Considerations
- Using Inference with the REST API
- Summary of APIs Used for Inference

7.1 Automatic Inference

Automatic inference is done using rulesets and ontologies. As the name implies, automatic inference is performed automatically and can also be centrally managed. MarkLogic semantics uses backward-chaining inference, meaning that the inference is performed at query time. This is very flexible; it means you can specify which ruleset(s) and ontology (or ontologies) to use per-query, with default rulesets per-database.

This section includes these topics:

- Ontologies
- Rulesets
- Memory Available for Inference
- A More Complex Use Case

7.1.1 Ontologies

An ontology is used to define the semantics of your data; it describes relationships in your data that can be used to infer new facts about your data. In Semantics, an ontology is a set of triples that provides a semantic model of a portion of the world, a model that enables knowledge to be represented for a particular domain (relationships between people, types of publications, or a taxonomy of medications). This knowledge model is a collection of triples used to describe the relationships in your data. Different vocabularies can supply sets of terms to define concepts and relationships to represent facts.

An ontology describes what types of things exist in the domain and how they are related. A vocabulary is composed of terms with clear definitions controlled by some internal or external authority. For example, the ontology triple `ex:dog skos:broader ex:mammal` states that `dog` is part of the broader concept `mammal`.

This SPARQL example inserts that ontology triple into a graph.

```
PREFIX skos: <http://www.w3.org/2004/02/skos/core#Concept/>
PREFIX ex: <http://example.org/>

INSERT DATA
{
GRAPH <http://marklogic.com/semantics/animals/vertebrates>
{
ex:dog skos:broader ex:mammal .
}}
```

You may want to use an ontology you have created to model your business or your area of research, and use that along with one or more rulesets to discover additional information about your data.

The rulesets are applied across all of the triples in scope for the query, including onotology triples. Ontology triples have to be in scope for the query in order to be used. There are multiple ways to do this:

- Use `FROM` or `FROM NAMED/GRAPH` in the query to specify what data is being accessed. Ontologies are organized by collection/named graph.
- Use `default-graph=` and `named-graph=` options to `sem:sparql` or `sem:sparql-update`.
- Use a `cts:query` to exclude data to be queried. Ontologies can be organized by directory, or anything else that a `cts:query` can find.
- Add the ontology to an in-memory store, and query across both the database and the in-memory store. In this case, the ontology is not stored in the database, and can be manipulated and changed for each query.
- Add the ontology to a ruleset as axiomatic triples. Axiomatic triples are triples that the ruleset says are always true - indicated by having an empty `WHERE` clause in the rule. You can then choose to include the ontologies in certain ruleset files or not at query time.

7.1.2 Rulesets

A ruleset is a set of inference rules, or rules that can be used to infer additional facts from data. Rulesets are used by the inference engine in MarkLogic to infer new triples from existing triples *at query time*. A ruleset may be built up by importing other rulesets. Inference rules enable you to search over both asserted triples and inferred triples. The semantic inference engine uses rulesets to create new triples from existing triples at query time.



For example, if you know that John lives in London and London is in England, you (as a human) know that John lives in England. You *inferred* that fact. Similarly, if there are triples in the database that say that John lives in London *and* that London is in England, and there are also triples that express the meaning of "lives in" and "is in" as part of an ontology, MarkLogic can infer that John lives in England. When you query your data for all the people that live in England, John will be included in the results.

Here is a simple custom rule (ruleset) to express the concept of "lives in":

```
# geographic rules for inference
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema/>
PREFIX ex: <http://example.com/>
PREFIX gn: <http://www.geonames.org/ontology/>

RULE "livesIn" CONSTRUCT {
  ?person ex:livesIn ?place2
} {
  ?person ex:livesIn ?place1 .
  ?place1 gn:parentFeature ?place2
}
```

This rule states (reading from the bottom up): if `place1` is in (has a `parentFeature`) `place2`, and a `person` lives in `place1`, then a `person` also lives in `place2`.

Inference that is done at query time using rulesets is referred to as "backward chaining" inference. Each SPARQL query looks at the specified ruleset(s) and creates new triples based on the results. This type of inferencing is faster during ingestion and indexing, but potentially a bit slower at query time. In general, inference becomes more expensive (slower) as you add more (and more complex) rules.

MarkLogic allows you to apply just the rulesets you need for each query. For convenience, you can specify the default ruleset or rulesets for a database, but you can also ignore those defaults for certain queries. It is possible to override the default ruleset association to allow querying without using inferencing and/or querying with alternative rulesets.

This section includes these topics:

- Pre-Defined Rulesets
- Specifying Rulesets for Queries
- Using the Admin UI to Specify a Default Ruleset for a Database
- Overriding the Default Ruleset
- Creating a New Ruleset
- Ruleset Grammar
- Example Rulesets

7.1.2.1 Pre-Defined Rulesets

Some pre-defined, standards-based rulesets (RDFS, RDFS-Plus, and OWL Horst) for inference are included with MarkLogic semantics. The rulesets are written in a language specific to MarkLogic that has the same syntax as the SPARQL `CONSTRUCT` query. Each ruleset has two versions; the full ruleset (`xxx-full.rules`) and the optimized version (`xxx.rules`).

The components of each of these rulesets are available separately so that you can do fine-grained inference with queries. You can also create your own rulesets by importing some of those rulesets and/or writing your own rules. See “Creating a New Ruleset” on page 156 for more information.

To see these pre-defined rulesets (in Linux), go to the Config directory under your MarkLogic install directory (`/`*MarkLogic_install_dir*`/Config/*.rules`). For example:

```
/opt/MarkLogic/Config/*.rules
```

You will see a set of files with the `.rules` extension. Each of these `.rules` files is a ruleset. For a Windows installation, these files are usually located in `C:\Program Files\MarkLogic\Config`).

Here is an example of the rule `domain.rules`:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema/>

tbox {
  ?p rdfs:domain ?c .
}

RULE "domain axioms" construct {
  rdfs:domain rdfs:domain rdf:Property .
  rdfs:domain rdfs:range rdfs:Class .
} {}

RULE "domain rdfs2" CONSTRUCT {
  ?x a ?c
} {
```

```
  ?x ?p ?y .
  ?p rdfs:domain ?c
}
```

In this example, `a` means "type of" (`rdf:type` or `rdfs:type`). The "`domain rdfs2`" rule states that if all the things in the second set of braces are true (`p` has domain `c`; that is, for every triple that has the predicate `p`, the object must be in the domain `c`), then construct the triple in the first set of braces (if you see `x p y`, then `x` is a `c`).

If you open a rule in a text editor you will see that some of the rulesets are componentized – that is, they are defined in small component rulesets, then built up into larger rulesets. For example, `rdfs.rules` imports four other rules to create the optimized set of rdfs rules:

```
# RDFS 1.1 optimized rules
prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#>

import "domain.rules"
import "range.rules"
import "subPropertyOf.rules"
import "subClassOf.rules"

RULE "rdf classes" construct {
...
...
```

By using a building block approach to using (and creating) rulesets, you can enable only the rules you really need, so that your query can be as efficient as possible. The syntax for rulesets is similar to the syntax for SPARQL `CONSTRUCT`.

7.1.2.2 Specifying Rulesets for Queries

You can choose which rulesets to use for your SPARQL query by using `sem:ruleset-store`. The ruleset is specified as part of the function. The `sem:ruleset-store` function returns a set of triples that result from the application of the ruleset to the triples defined by the `sem:store` function provided in `$store` (for example, "all of the triples that can be inferred from this rule").

This statement specifies the `rdfs.rules` ruleset as part of `sem:ruleset-store`:

```
let $rdfs-store := sem:ruleset-store("rdfs.rules",sem:store() )
```

So this says, let `$rdfs-store` contain triples derived by inference using the `rdfs.rules` against the `sem:store`. If no value is provided for `sem:store`, the query uses the triples in the current database's triple index. The built-in functions `sem:store` and `sem:ruleset-store` are used to define the triples over which to query and the rulesets (if any) to use with the query. The `$store` definition includes a ruleset, as well as other ways of restricting a query's domain, such as a `cts:query`.

This example uses the pre-defined `rdfs.rules` ruleset:

```
import module namespace sem = "http://marklogic.com/semantics"
  at "/MarkLogic/semantics.xqy";
let $sup :=
'PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

INSERT DATA
{ <someMedicalCondition> rdf:type <osteoarthritis> .
  <osteoarthritis> rdfs:subClassOf <bonedisease> . }'
return sem:sparql-update($sup)
; (: transaction separator :)

let $sq :=
'PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX d: <http://diagnoses#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?diagnosis
WHERE { ?diagnosis rdf:type <bonedisease>. } '

(: rdfs.rules is a predefined ruleset :)
let $rs := sem:ruleset-store("rdfs.rules", sem:store())
return sem:sparql($sq, (), (), $rs)
(: the rules specify that query for <bonedisease> will return the
subclass <osteoarthritis> :)
```

Note: If graph URIs are included as part of a SPARQL query that includes `sem:store` or `sem:ruleset-store`, the query will include "triples that are in the store and also in these graphs".

This example is a SPARQL query against the data in `$triples`, using the rulesets `rdfs:subClassOf` and `rdfs:subPropertyOf`:

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

let $triples := sem:store((), cts:word-query("henley"))
return
sem:sparql("
PREFIX skos: <http://www.w3.org/2004/02/skos/core#Concept/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT * { ?c a skos:Concept; rdfs:label ?l }",(),(),
sem:ruleset-store(("subClassOf.rules","subPropertyOf.rules"),
  ($triples))
)
```

You can manage the default rulesets for a database using the Admin UI, the REST Mangement API, or XQuery Admin API. See "Using the Admin UI to Specify a Default Ruleset for a Database" on page 153 for information about specifying rulesets with the Admin UI. See the `default-ruleset` property in `PUT:/manage/v2/databases/{id|name}/properties` for REST Management API details. See `admin:database-add-default-ruleset` for Admin API details.

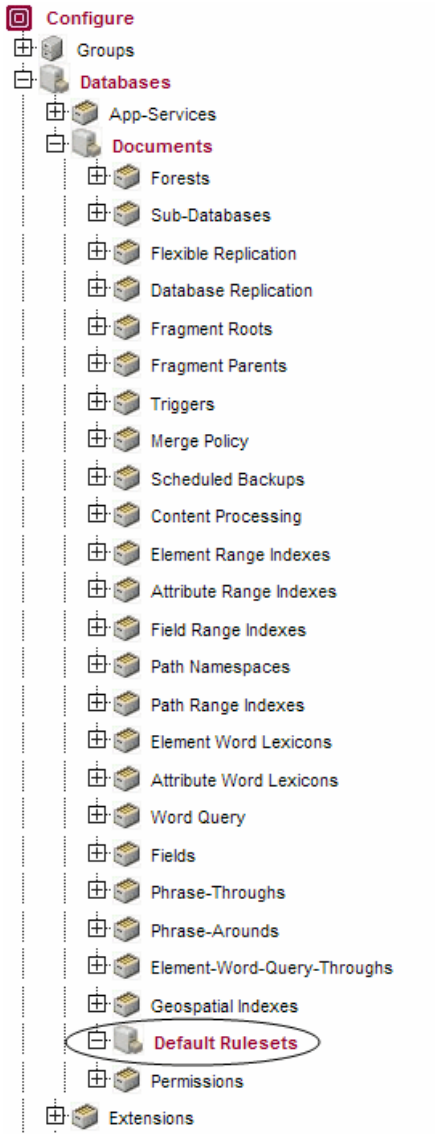
7.1.2.3 Using the Admin UI to Specify a Default Ruleset for a Database

You can use the Admin UI to set the default ruleset to be used for queries against a specific database (for example, “when using this database, use this ruleset for queries”).

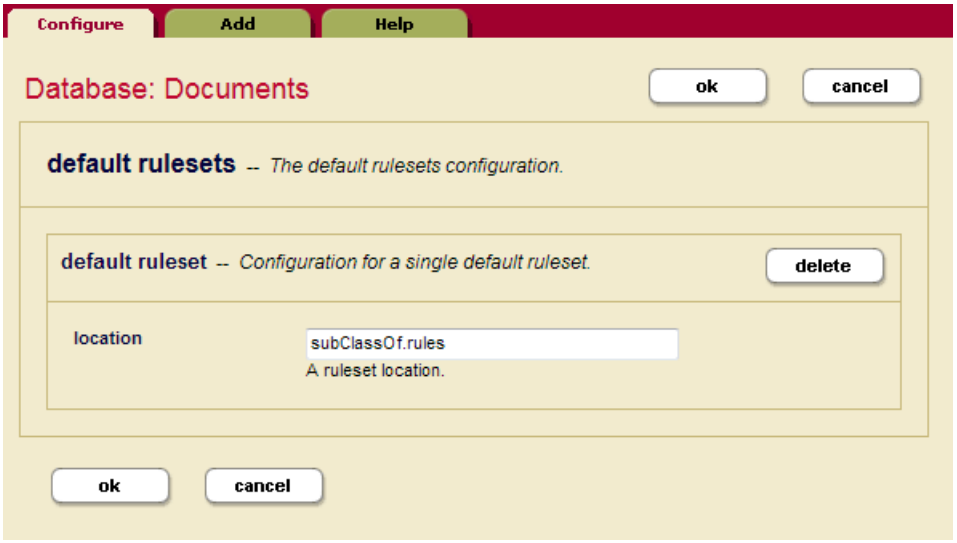
To specify the ruleset or rulesets for a database:

1. Click the Databases in left tree menu of the Admin UI.

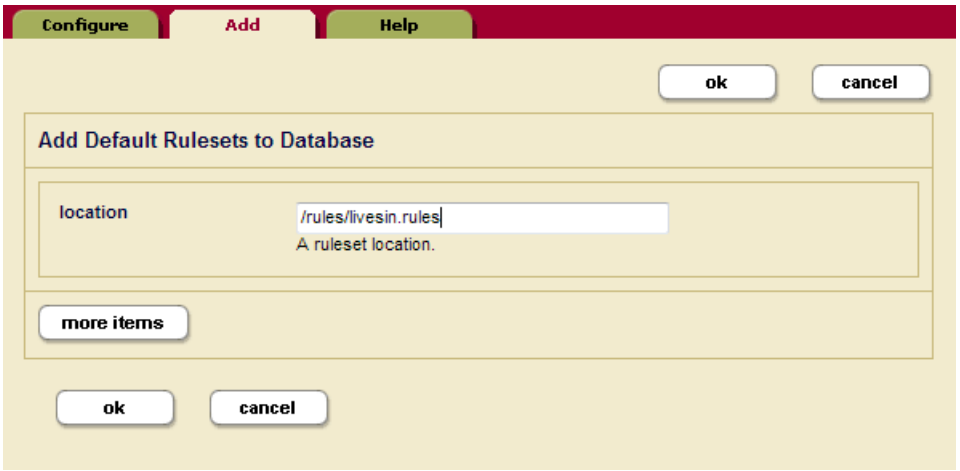
2. Click the database name to expand the list and scroll to Default Rulesets.



3. Click Default Rulesets to see the rulesets currently associated with the Documents database.



4. To add your own ruleset, click Add to enter the name and location of the ruleset.



5. Your custom rulesets will be located in the schemas database.

   The rulesets supplied by MarkLogic are located in the Config directory under your MarkLogic installation directory (`/`*MarkLogic_install_dir*`/Config/*.rules`).

6. Click more items to associate additional rulesets with this database.

   Note: Security for rulesets is managed the same way that security is handled for MarkLogic schemas.

You can use Query Console to find out what default rulesets are currently associated with a database using the `admin:database-get-default-rulesets` function.

This example will return the name and location of the default rulesets for the Documents database:

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

let $config := admin:get-configuration()
let $dbid := admin:database-get-id($config, "Documents")
return admin:database-get-default-rulesets($config, $dbid)

=>

<default-ruleset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://marklogic.com/xdmp/database">
    <location>/rules/livesin.rules</location>
</default-ruleset>
```

Note: If you have a default ruleset associated with a database and you specify a ruleset as part of your query, both rulesets will be used. Rulesets are additive. Use the `no-default-ruleset` option in `sem:store` to ignore the default ruleset.

7.1.2.4 Overriding the Default Ruleset

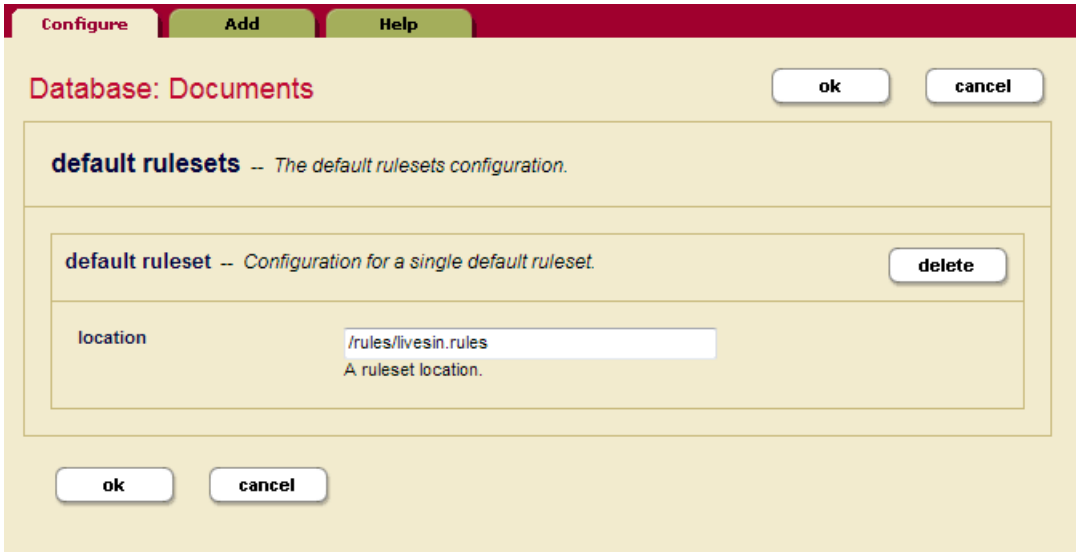
You can turn off or ignore a ruleset set as the default on a database. In this example, a SPARQL query is executed against the database, ignoring the default rulesets and using the `rdfs:subClassOf` inference ruleset for the query:

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

sem:sparql("
PREFIX skos: <http://www.w3.org/2004/02/skos/core#Concept/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT * {
  ?c a skos:Concept;
  rdfs:label ?l }",(),(),
sem:ruleset-store("subClassOf.rules",sem:store("no-default-rulesets"))
)
```

You can also turn off or ignore a ruleset as part of a query, through the Admin UI, or by using XQuery or JavaScript to specify the ruleset.

You can also change the default ruleset for a database in the Admin UI by “deleting” the default ruleset from that database. In the Admin UI, select the database name from the left navigation panel, click the database name. Click Default Rulesets.



On the Database: Documents panel, select the default ruleset you want to remove, and click delete. Click ok when you are done. The ruleset is no longer the default ruleset for this database.

Note: This action does not delete the ruleset, only removes it as the default ruleset.

You can also use `admin:database-delete-default-ruleset` with XQuery to change a database's default ruleset. This example removes `subClassOf.rules` as the default ruleset for the Documents database.

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

let $config := admin:get-configuration()
let $dbid := admin:database-get-id($config, "Documents")
let $rules := admin:database-ruleset("subClassOf.rules")
let $c := admin:database-delete-default-ruleset($config, $dbid, $rules)

return admin:save-configuration($c)
```

7.1.2.5 Creating a New Ruleset

You can create your own rulesets to use for inference in MarkLogic. MarkLogic rulesets are written in a language specific to MarkLogic, based on the SPARQL `CONSTRUCT` query.

One way to think of inference rules is as a way to construct some inferred triples, then search over the new data set (one that includes the portion of the database defined by the `sem:store` plus the inferred triples).

The MarkLogic-supplied rulesets are located in the install directory:

```
/MarkLogic_install_dir/Config/*.rules
```

When you create a custom ruleset, insert it into the schemas database and refer to it as a URI in the schemas database. A ruleset location is either a URI for the database you are using in the schemas database, or a filename in *MarkLogic_Install_Directory*`/Config`.

> **Note:** MarkLogic will search first for the MarkLogic-provided rulesets in `/Config` and then in the schemas database for any other rulesets.

7.1.2.6 Ruleset Grammar

MarkLogic rulesets are written in a language specific to MarkLogic. The language is based on the SPARQL 1.1 grammar. The syntax of an inference rule is similar to the grammar for SPARQL `CONSTRUCT`, with the `WHERE` clause restricted to a combination of only triple patterns, joins, and filters. The ruleset must have a unique name.

The following grammar specifies the MarkLogic Ruleset Language.

```
Rules ::= RulePrologue Rule*
Rule ::= 'RULE' RuleName 'CONSTRUCT' ConstructTemplate 'WHERE'?
    RuleGroupGraphPattern
RuleName ::= String
RuleGroupGraphPattern ::= '{' TriplesBlock? ( ( Filter
    RuleGroupGraphPattern ) '.'? TriplesBlock? )* '}'
RulePrologue ::= ( BaseDecl | PrefixDecl | RuleImportDecl )*
RuleImportDecl ::= 'IMPORT' RuleImportLocation
RuleImportLocation ::= String
```

The `String` for `RuleImportLocation` must be a URI for the location of the rule to be imported. Non-terminals that are not defined here (like `BaseDecl`) are references to productions in the SPARQL 1.1 grammar.

- The grammar restricts the contents of a rule's `WHERE` clause, and it is further restricted during static analysis to a combination of only triple patterns, joins, and filters.
- Comments are allowed using standard SPARQL comment syntax (comments in the form of "`#`", outside an IRI or string, and continuing to the end of line).
- A MarkLogic ruleset uses the extension "`.rules`" and has a mimetype of "`application/vnd.marklogic-ruleset`".
- Some kinds of property path operators ("`/`", "`^`", for instance) can be used as part of ruleset. However, you cannot use these operators as part of a property path in a ruleset: "|", "?", "*", or "+".

The import statement in the prolog includes all rules from the ruleset found at the location given, and all other rulesets imported transitively. If a ruleset at a given location is imported more than once, the effect of the import will be as if it had only been imported once. If a ruleset is imported more than once from different locations, MarkLogic will assume that they are different rulesets and raise an error because of the duplicate rule names they contain (`XDMP-DUPRULE`).

7.1.2.7 Example Rulesets

This ruleset (`subClassOf.rules`) from the */MarkLogic_Install*`/Config` directory includes prefixes, and has rule names and a `CONSTRUCT` clause. The `subClassOf rdfs9` rule is the one doing the work:

```
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>

tbox {
  ?c1 rdfs:subClassOf ?c2 .
}

RULE "subClassOf axioms" CONSTRUCT {
  rdfs:subClassOf rdfs:domain rdfs:Class .
  rdfs:subClassOf rdfs:range rdfs:Class .
} {}

RULE "subClassOf rdfs9" CONSTRUCT {
  ?x a ?c2
} {
  ?x a ?c1 .
  ?c1 rdfs:subClassOf ?c2 .
  FILTER(?c1!=?c2)
}
```

Note that the `subClassOf rdfs9` rule also includes a `FILTER` clause.

This ruleset from same directory (`rdfs.rules`) imports smaller rulesets to make a ruleset approximating the full RDFS ruleset:

```
# RDFS 1.1 optimized rules
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>

import "domain.rules"
import "range.rules"
import "subPropertyOf.rules"
import "subClassOf.rules"

RULE "rdf classes" CONSTRUCT {
  rdf:type a rdf:Property .
  rdf:subject a rdf:Property .
  rdf:predicate a rdf:Property .
  rdf:object a rdf:Property .
  rdf:first a rdf:Property .
  rdf:rest a rdf:Property .
  rdf:value a rdf:Property .
  rdf:nil a rdf:List .
} {}

RULE "rdfs properties" CONSTRUCT {
  rdf:type rdfs:range rdfs:Class .
```

```
  rdf:subject rdfs:domain rdf:Statement .
  rdf:predicate rdfs:domain rdf:Statement .
  rdf:object rdfs:domain rdf:Statement .

  rdf:first rdfs:domain rdf:List .
  rdf:rest rdfs:domain rdf:List .
  rdf:rest rdfs:range rdf:List .

  rdfs:isDefinedBy rdfs:subPropertyOf rdfs:seeAlso .
} {}

RULE "rdfs classes" CONSTRUCT {
  rdf:Alt rdfs:subClassOf rdfs:Container .
  rdf:Bag rdfs:subClassOf rdfs:Container .
  rdf:Seq rdfs:subClassOf rdfs:Container .
  rdfs:ContainerMembershipProperty rdfs:subClassOf rdf:Property .
} {}

RULE "datatypes" CONSTRUCT {
  rdf:XMLLiteral a rdfs:Datatype .
  rdf:HTML a rdfs:Datatype .
  rdf:langString a rdfs:Datatype .
} {}

RULE "rdfs12" CONSTRUCT {
  ?p rdfs:subPropertyOf rdfs:member
} {
  ?p a rdfs:ContainerMembershipProperty
}
```

This is the custom rule shown earlier that you could create and use to infer information about geographic locations:

```
# geographic rules for inference
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema/>
PREFIX ex: <http://example.com/>
PREFIX gn: <http://www.geonames.org/ontology/>

RULE "lives in" CONSTRUCT {
  ?person ex:livesIn ?place2
} {
  ?person ex:livesIn ?place1 .
  ?place1 gn:parentFeature ?place2
}
```

Add the `livesIn` rule to the schemas database using `xdmp:document-insert` and Query Console. Make sure the schemas database is selected as the Content Source before you run the code:

```
xquery version "1.0-ml";

xdmp:document-insert(
'/rules/livesin.rules',
text{'
# geographic rules for inference
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema/>
PREFIX ex: <http://example.com/>
PREFIX gn: <http://www.geonames.org/ontology/>

RULE "lives in" CONSTRUCT {
  ?person ex:livesIn ?place2
} {
  ?person ex:livesIn ?place1 .
  ?place1 gn:parentFeature ?place2
  }'
})
```

The example stores the `livesin.rule` in the schemas database, in the rules directory (`/rules/livesin.rules`). You can include your ruleset as part of inference in the same way you can include the supplied rulesets. MarkLogic will check the location for rules in the schemas database and then the location for the supplied rulesets.

7.1.3 Memory Available for Inference

The default, maximum, and minimum inference size values are all per-query, not per-system. The maximum inference size is the memory limit for inference. The `appserver-max-inference-size` function allows the administrator to set a memory limit for inference. You cannot exceed this amount.

The default inference size is the amount of memory available to use for inference. By default the amount of memory available for inference is 100mb (`size=100`). If you run out of memory and get an inference full error (`INFFULL`), you need to increase the default memory size using `admin:appserver-set-default-inference-size` or by changing the default inference size on the HTTP Server Configuration page in the Admin UI.

You can also set the inference memory size in your query as part of `sem:ruleset-store`. This query sets the memory size for inference to 300mb (`size=300`):

```
Let $store := sem:ruleset-store(("baseball.rules", "rdfs-plus-
full.rules"),
sem:store(), ("size=300"))
```

If your query returns an `INFFULL` exception, you can to change the size in `ruleset-store`.

7.1.4 A More Complex Use Case

You can use inference in more complex queries. This is a JavaScript example of a SPARQL query where an ontology is added to an in-memory store. The in-memory store uses inference to discover recipes that use soft goat's cheese. The query then returns the list of possible recipe titles.

```
var sem = require("/MarkLogic/semantics.xqy");

var inmem = sem.inMemoryStore(
  sem.rdfParse(`
    prefix ch: <http://marklogic.com/semantics/cheeses/>
    prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
    prefix owl: <http://www.w3.org/2002/07/owl#>
    prefix dcterms: <http://purl.org/dc/terms/>

    ch:FreshGoatsCheese owl:intersectionOf (
      ch:SoftFreshCheese
      [ owl:hasValue ch:goatsMilk ;
      owl:onProperty ch:milkSource ]
    ) .`,"turtle"));
var rules = sem.rulesetStore(
  ["intersectionOf.rules","hasValue.rules"],
  [inmem,sem.store()])

sem.sparql(`
    prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
    prefix dcterms: <http://purl.org/dc/terms/>
    prefix f: <http://linkedrecipes.org/schema/>
    prefix ch: <http://marklogic.com/semantics/cheeses/>

select ?title ?ingredient WHERE {
  ?recipe dcterms:title ?title ;
    f:ingredient [
      a ch:FreshGoatsCheese ;
      rdfs:label ?ingredient]
    }`,[],[],rules)
```

To get results back from this query, you would need to have a triplestore of recipes, that also includes triples describing cheese made from goat's milk.

7.2 Other Ways to Achieve Inference

Before going down the path of automatic inference, consider other ways to achieve inference that may be more appropriate for your use case.

This section includes these topics:

- Using Paths
- Materialization

7.2.1 Using Paths

In many cases, you can do inference by rewriting your query. For example, you can do some simple inference using unenumerated property paths. Property paths (as explained in “Property Path Expressions” on page 118) enable a simple kind of inference.

You can find all the possible types of a resource, including supertypes of a resources, using RDFS vocabulary and the “/” property path in a SPARQL query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2001/01/rdf-schema#>
SELECT ?type
{
   <http://example/thing> rdf:type/rdfs:subClassOf* ?type
}
```

The result will be all resources and their inferred types. The unenumerated property path expression with the asterisk (*) will look for triples where the subject and object are connected by `rdf:type` and followed by zero or more occurrences of `rdfs:subClassOf`.

For example, you could use this query to find the products that are subClasses of “shirt”, at any depth in the hierarchy:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ex: <http://example.com>

SELECT ?product
WHERE
  {
    ?product rdf:type/rdfs:subClassOf* ex:Shirt ;
}
```

Or you could use a property path to find people who live in England:

```
PREFIX gn: <http://www.geonames.org/ontology/>
PREFIX ex: <http://www.example.org>

SELECT ?p
{
  ?p ex:livesIn/gn:parentFeature "England"
}
```

For more about property paths and how to use them with semantics, see “Property Path Expressions” on page 118.

7.2.2 Materialization

A possible alternative to automatic inference at query time (backward-chaining inference) is materialization or forward-chaining inference, where you perform inference on parts of your data, not as part of a query, and then store those inferred triples to be queried later. Materialization worsk best for triple data that is fairly static; performing inference with rules and ontologies that do not change often.

This process of materialization at ingestion or update time may be time-consuming and will require a significant amount of disk space for storage. You will need to write code or scripts to handle transactions and security and to handle changes in data and ontologies.

> **Note:** These tasks are all handled for you if you choose automatic inference.

Materialization can be very useful if you need very fast queries and you are prepared to do the pre-processing work up front and use the extra disk space for the inferred triples. You may want to use this type of inference in situations where the data, rulesets, your ontologies, and some parts of your data do not change very much.

You can mix and match; materialize some inferred triples that do not change very much (such as ontology triples: for example, a customer is a person is a legal entity), while using automatic inference for triples that change or are added to more often. You can also use automatic inference where inference has a broader scope (`new-order-111` contains `line-item-222`, which contains `product-333`, which is related to `accessory-444`).

7.3 Performance Considerations

The key to making your SPARQL queries run fast is “partitioning” the data (by writing a sufficiently rich query that the number of results returned is small). Backward-chaining inference will run faster in the available memory if it is querying over fewer triples. To achieve this, make your inference queries very selective by using a `FILTER` or constraining the scope of the query through `cts:query` (for example a collection-query).

7.3.1 Partial Materialization

You can do partial materialization of data, rulesets, and ontologies that you use frequently and that do not change often. You can perform inference on parts of your data to materialize the inferred triples and use these materialized triples in your inference queries.

To materialize these triples, construct SPARQL queries for the rules that you want to use for inference and run them on your data as part of your ingestion or update pipeline.

7.4 Using Inference with the REST API

When you execute a SPARQL query or update using the REST Client API methods `POST:/v1/graphs/sparql` or `GET:/v1/graphs/sparql`, you can specify rulesets through the request parameters `default-rulesets` and `rulesets`. If you omit both of these parameters, the default rulesets for the database are applied.

After you set `rdfs.rules` and `equivalentProperties.rules` as the default rulesets for the database, you can perform this SPARQL query using REST from the Query Console:

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

let $uri := "http://localhost:8000/v1/graphs/sparql"
return
let $sparql :='
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX prod:   <http://example.com/products/>
PREFIX ex:     <http://example.com/>

SELECT ?product
FROM <http://marklogic.com/semantics/products/inf-1>
WHERE
 {
  ?product  rdf:type  ex:Shirt ;
  ex:color  "blue"
}
'
let $response :=
xdmp:http-post($uri,
<options xmlns="xdmp:http">
  <authentication method="digest">
    <username>admin</username>
    <password>admin</password>
  </authentication>
  <headers>
    <content-type>application/sparql-query</content-type>
    <accept>application/sparql-results+xml</accept>
  </headers>
</options>
text {$sparql})
  return
  ($response[1]/http:code, $response[2] /node())

=>

    product
<http://example.com/products/1001>
<http://example.com/products/1002>
<http://example.com/products/1003>
```

Using the REST endpoint and `curl` (with the same default rulesets for the database), the same query would look like this:

```
curl --anyauth --user Admin:janem-3 -i -X POST \
-H "Content-type:application/x-www-form-urlencoded" \
-H "Accept:application/sparql-results+xml" \
--data-urlencode query='PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
```

```
syntax-ns#> \
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX prod: <http://example.com/products/> \
PREFIX ex: <http://example.com/> \
SELECT ?product FROM <http://marklogic.com/semantics/products/inf-1> \
WHERE {?product rdf:type ex:Shirt ; ex:color  "blue"}' \
http://localhost:8000/v1/graphs/sparql
```

To run this `curl` example, remove the "\" characters and make the command one line. See "Using Semantics with the REST Client API" on page 189 and Querying Triples in the *REST Application Developer's Guide* for more information.

7.5 Summary of APIs Used for Inference

MarkLogic has APIs that can be used for semantic inference. Semantic APIs are available for use as part of the actual inference query (specifying which triples to query and which rules to apply). Database APIs can be used to choose rulesets to be used for inference by a particular database. Management APIs can control the memory used by inference by either an appserver or a taskserver.

- Semantic APIs
- Database Ruleset APIs
- Management APIs

7.5.1 Semantic APIs

MarkLogic Semantic APIs can be used for managing triples for inference and for specifying rulesets to be used with individual queries (or by default with databases). Stores are used to identify the subset of triples to be evaluted by the query.

Semantic API	Description
`sem:store`	The query argument of `sem:sparql` accepts `sem:store` to indicate the source of the triples to be evaluated as part of the query. If multiple `sem:store` constructors are supplied, the triples from all the sources are merged and queried together. The `sem:store` can contain one or more options along with a `cts:query` to restrict the scope of triples to be evaluated as part of the `sem:sparql` query. The `sem:store` parameter can also be used with `sem:sparql-update` and `sem:sparql-values`.
`sem:in-memory-store`	Returns a `sem:store` that represents the set of triples from the `sem:triple` values passed in as an argument. The default rulesets configured on the current database have no effect on a `sem:store` created with `sem:in-memory-store`.

Semantic API	Description
`sem:ruleset-store`	Returns a new `sem:store` that represents the set of triples derived by applying the ruleset to the triples in `sem:store` in addition to the original triples.

Note: Use the `sem:in-memory-store` function with `sem:sparql` in preference to the deprecated `sem:sparql-triples` function (available in MarkLogic 7). The `cts:query` argument to `sem:sparql` has also been deprecated.

If you call `sem:sparql-update` with a store that is based on in-memory triples (that is, a store that was created by `sem:in-memory-store`) you will get an error because you cannot update triples that are in memory and not on disk. Similarly, if you pass in multiple stores to `sem:sparql-update` and any of them is based on in-memory triples you will get an error.

7.5.2 Database Ruleset APIs

These Database Ruleset APIs are used to manage the rulesets associated with databases.

Ruleset API	Description
`admin:database-ruleset`	The ruleset element to be used for inference on a database. One or more rulesets can be used for inference. By default, no ruleset is configured.
`admin:database-get-default-rulesets`	Returns the default ruleset(s) for a database.
`admin:database-add-default-ruleset`	Adds a ruleset to be used for inference on a database. One or more rulesets can be used for inference. By default, no ruleset is configured.
`admin:database-delete-default-ruleset`	Deletes the default ruleset used by a database for inference.

7.5.3 Management APIs

These Management APIs are used to manage memory sizing (default, minimum, and maximum) alloted for inference.

Management API (admin:)	Description
`admin:appserver-set-default-inference-size`	Specifies the default value for any request's inference size on this application server.

Management API (admin:)	Description
`admin:appserver-get-default-inference-size`	Returns the default amount of memory (in megabytes) that can be used by `sem:store` for inference by an application server.
`admin:taskserver-set-default-inference-size`	Specifies the default value for any request's inference size on this task server.
`admin:taskserver-get-default-inference-size`	Returns the default amount of memory (in megabytes) that can be used by `sem:store` for inference by a task server.
`admin:appserver-set-max-inference-size`	Specifies the upper bound for any request's inference size. The inference size is the maximum amount of memory in megabytes allowed for `sem:store` performing inference on this application server.
`admin:appserver-get-max-inference-size`	Returns the maximum amount of memory (in megabytes) that can be used by `sem:store` for inference by an application server.
`admin:taskserver-set-max-inference-size`	Specifies the upper bound for any request's inference size. The inference size is the maximum amount of memory in megabytes allowed for `sem:store` performing inference on this task server.
`admin:taskserver-get-max-inference-size`	Returns the maximum amount of memory (in megabytes) that can be used by `sem:store` for inference by a task server.

8.0 SPARQL Update

The SPARQL1.1 Update language is used to delete, insert, and update (delete/insert) triples and graphs. An update is actually an `INSERT`/`DELETE` operation in the database.

SPARQL Update is a part of the SPARQL 1.1 suite of specifications at http://www.w3.org/TR/2013/REC-sparql11-update-20130321. It is a separate language from the SPARQL Query. SPARQL Update enables you to manipulate triples or sets of triples, while the SPARQL query language enables you to search and query your triple data.

You can manage security level using SPARQL Update. All SPARQL queries over managed triples are governed by the graph permissions. Triple documents will inherit those permissions at ingest.

Only triples managed by MarkLogic–triples whose document root is `sem:triples`–are affected by SPARQL Update. Managed triples are triples that have been loaded into the database using:

- mlcp with `-input_file_type RDF`
- `sem:rdf-load`
- `sem:rdf-insert`
- `sem:sparql-update`

Embedded triples are part of an XML or JSON document . If you want to create, delete, or update embedded triples, use the appropriate document update functions. See “Unmanaged Triples” on page 73 for more information about triples embedded in documents. Unmanaged triples can also be modified and updated with document management functions. See “Inserting, Deleting, and Modifying Triples with XQuery and Server-Side JavaScript” on page 239 for details.

This section includes the following parts:

- Using SPARQL Update
- Graph Operations with SPARQL Update
- Graph-Level Security
- Data Operations with SPARQL Update
- Bindings for Variables
- Using SPARQL Update with Query Console
- Using SPARQL Update with XQuery or Server-Side JavaScript
- Using SPARQL Update with REST

8.1 Using SPARQL Update

You can use SPARQL Update to insert and delete managed triples in a Graph Store. There are two kinds of SPARQL Update operations: graph data operations, and graph management operations.

There are several ways to use SPARQL Update:

- From Query Console - Select SPARQL Update as the Query Type from the drop-down list. See "Using SPARQL Update with Query Console" on page 185.
- Using XQuery or JavaScript - Call SPARQL Update from XQuery (`sem:sparql-update`) or JavaScript (`sem.sparqlUpdate`). See "Using SPARQL Update with XQuery or Server-Side JavaScript" on page 186.
- Through the REST API (`GET:/v1/graphs/sparql` or `POST:/v1/graphs/sparql`). See "Using SPARQL Update with REST" on page 187.

SPARQL Update is used with managed triples. To modify "embedded" or unmanaged triples, use the appropriate document update functions with XQuery or JavaScript. See "Inserting, Deleting, and Modifying Triples with XQuery and Server-Side JavaScript" on page 239.

A new role has been added for SPARQL Update - `sparql-update-user` . Users must have `sparql-update-user` capabilities to `INSERT`, `DELETE`, or `DELETE/INSERT` triples into graphs. See Role-Based Security Model in the *Security Guide* for details.

8.2 Graph Operations with SPARQL Update

You can manipulate RDF graphs using SPARQL Update. Graph management operations include `CREATE`, `DROP`, `COPY`, `MOVE`, and `ADD`.

The SPARQL Update spec includes these commands and options for working with RDF graphs:

Command	Options	Description
CREATE	SILENT, GRAPH IRIref	Creates a new graph. Use GRAPH to name the graph. SILENT creates the graph silently. This means if the graph already exists, do not return an error.
DROP	SILENT, GRAPH IRIref, DEFAULT, NAMED, ALL	Drops a graph and its contents. Use GRAPH to name a graph to remove, DEFAULT to remove the default graph
COPY	SILENT, GRAPH, IRIref_from, DEFAULT, TO, IRIref_to	Copies the source graph into the destination graph. Any content in the destination graph will be overwritten (deleted).

Command	Options	Description
MOVE	SILENT, GRAPH, IRIref_from, DEFAULT, TO, IRIref_to	Moves the contents of the source graph into the destination graph, and removes that content from the source graph. Any content in the destination graph will be overwritten (deleted).
ADD	SILENT, GRAPH, IRIref_from, DEFAULT, TO, IRIref_to	Add the contents of the source graph to the destination graph. The ADD operation keeps the content of both the source graph and destination graph intact.

Multiple statements separated by semicolons (;) in one SPARQL Update operation, run in the same transaction, and can be included in the same request. It is important to note that each statement can see the result of the previous statements.

For example, in this query, the `COPY` operation can see the graph `<TEST>` created in the first statement:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>

INSERT DATA
{
  <http://example/book0> dc:title "A default book"
  GRAPH <TEST> {<http://example/book1> dc:title "A new book" }
  GRAPH <TEST> {<http://example/book2> dc:title "A second book" }
  GRAPH <TEST> {<http://example/book3> dc:title "A third book" }
};
COPY <TEST> TO <BOOKS1>
```

Note: SPARQL Update operations return the empty sequence.

8.2.1 CREATE

This operation creates a graph. If the graph already exists, an error will be returned unless the `SILENT` option is used. The contents of already existing graphs remain unchanged. If you do not have permission to create the graph, an error is returned (unless the `SILENT` option is used).

The syntax for a `CREATE` operation is:

```
CREATE ( SILENT )? GRAPH IRIref
```

If the `SILENT` option is used, the operation will not return an error. The `GRAPH IRIref` option specifies the IRI for the new graph.

For example:

```
CREATE GRAPH <http://marklogic.com/semantics/tutorial/update> ;
```

If the specified destination graph does not exist, the graph will be created. The CREATE operation will create a graph with the permissions specified through sem:sparql-update, or with the user's default permission if no permissions are specified.

8.2.2 DROP

The DROP operation removes the specified graph or graphs from the Graph Store. The syntax for a DROP operation is:

```
DROP ( SILENT )? GRAPH IRIref | DEFAULT |NAMED | ALL )
```

The GRAPH keyword is used to remove a graph specified by IRIref . The DEFAULT keyword option is used to remove the default graph from the Graph Store. The NAMED keyword is used to remove all named graphs from the Graph Store. All graphs are removed from the Graph Store with the ALL keyword; this is the same as resetting the Graph store.

For example:

```
DROP SILENT GRAPH <http://marklogic.com/semantics/tutorial/intro> ;
```

After successful completion of this operation, the specified graphs are no longer available for further graph update operations. This operation returns an error if the specified named graph does not exist. If SILENT is present, the result of the operation will always be success.

Note: If the default graph of the Graph Store is dropped, MarkLogic creates a new, empty default graph with the user's default permissions.

8.2.3 COPY

The COPY operation is used for inserting all of the triples from a source graph into a destination graph. Triples from the source graph are not affected, but triples in the destination graph, if any exist, are removed before the new triples are inserted.

The syntax for a COPY operation is:

```
COPY ( SILENT )? ( ( GRAPH )? IRIref_from | DEFAULT) TO ( ( GRAPH )?
    IRIref_to | DEFAULT )
```

The COPY operation copies permissions from a source graph to the destination graph. Since source graph has the permission info, $perm parameter in sem:sparql-update does not apply in a COPY operation.

For example:

```
COPY <http://marklogic.com/semantics/tutorial/intro> TO
  <http://marklogic.com/semantics/tutorial/start> ;
```

If the destination graph does not exist, it will be created. The operation returns an error if the source graph does not exist. If the SILENT option is used, the result of the operation will always be success.

The COPY operation is similar to dropping a graph and then inserting a new graph:

```
DROP SILENT (GRAPH IRIref_to | DEFAULT); INSERT { ( GRAPH IRIref_to )?
    { ?s ?p ?o } } WHERE { ( GRAPH IRIref_from )? { ?s ?p ?o } }
```

If COPY is used to copy a graph onto itself, no operation is performed and the data is left as it was.

If you want the update to fail when the destination graph does not already exist, the `existing-graph` option in `sem:sparql-update` needs to be specified. If you copy into a new graph, that new graph takes the permissions of the graph that you copied from. If you copy into an existing graph, the permssions of that graph do not change.

8.2.4 MOVE

The MOVE operation is used for moving all triples from a source graph into a destination graph. The syntax for a MOVE operation is:

```
MOVE (SILENT)? ( ( GRAPH )? IRIref_from | DEFAULT) TO ( ( GRAPH )?
    IRIref_to | DEFAULT)
```

The source graph is removed after insertion. Triples in the destination graph, if any exist, are removed before destination triples are inserted.

For example:

```
MOVE <http://marklogic.com/semantics/tutorial/queries> TO
  <http://marklogic.com/semantics/tutorialSearches> ;
```

The graph `<http://marklogic.com/semantics/queries>` is removed after its triples have been inserted into `<http://marklogic.com/semantics/searches>`. Any triples in the graph `<http://marklogic.com/semantics/searches>` are deleted before the other triples are inserted.

> **Note:** If MOVE is used to move a graph onto itself, no operation will be performed and the data will be left as it was.

If the destination graph does not exist, it will be created. The MOVE operation returns an error if the source graph does not exist. If the SILENT option is used, the result of the operation will always be success.

The MOVE operation is similar to :

```
DROP SILENT (GRAPH IRIref_to | DEFAULT); INSERT { ( GRAPH IRIref_to )?
    { ?s ?p ?o } } WHERE { ( GRAPH IRIref_from )? { ?s ?p ?o } };
    DROP ( GRAPH IRIref_from | DEFAULT)
```

The MOVE operation moves permissions from source graph to destination graph. Since source graph has the permission info, the $perm parameter in sem:sparql-update does not apply to the operation.

If you want the update to fail when the destination graph does not already exist, the existing-graph option in sem:sparql-update needs to be specified. If you copy into a new graph, that new graph takes the permissions of the graph that you copied from. If you copy into an existing graph, the permssions of that graph do not change.

8.2.5 ADD

The ADD operation is used for inserting all triples from a source graph into a destination graph. The triples from the source graph are not affected, and existing triples from the destination graph, if any exist, are kept intact.

The syntax for an ADD operation is:

```
ADD ( SILENT )? ( ( GRAPH )? IRIref_from | DEFAULT) TO ( ( GRAPH )?
IRIref_to | DEFAULT)
```

For example:

```
ADD <http://marklogic.com/semantics/tutorial/queries> TO
  <http://marklogic.com/semantics/searches> ;
```

If ADD is used to add a graph onto itself, no operation will be performed and the data will be left as it was. If the destination graph does not exist, it will be created. The ADD operation returns an error if the source graph does not exist. If the SILENT option is used, the result of the operation will always be success.

If you are adding triples into a new graph, you can set the permissions of the new graph through sem:sparql-update. If no permissions are specified, your default graph permissions will be applied to the graph.

The ADD operation is equivalent to:

```
INSERT { ( GRAPH IRIref_to )? { ?s ?p ?o } } WHERE
    { ( GRAPH IRIref_from )? { ?s ?p ?o } }
```

If you want the update to fail when the destination graph does not already exist, the existing-graph option in sem:sparql-update needs to be specified. If you copy into a new graph, that new graph takes the permissions of the graph that you copied from. If you copy into an existing graph, the permssions of that graph do not change.

8.3 Graph-Level Security

You can manage security at the graph level using SPARQL Update. Graph-level security means that only users with a role corresponding to the permissions set on a graph can view a graph, change graph content, or create a new graph. As a user, you can only see the triples and graphs for which you have read permissions (via a role). See Role-Based Security Model in the *Security Guide*.

By default, graphs are created with the default permissions of the user creating the graph. If graph permissions are specified by passing in permissions as an argument to the `sem:sparql-update` call, graph operations and graph management operations that result in the creation of a new graph will use the specified permissions. This is true whether the graph is created explicitly using `CREATE GRAPH`, or implicitly by inserting or copying triples (`INSERT` or a graph operation to copy, move, or add) into a graph that does not already exist.

If you pass in permissions with `sem:sparql-update` on an operation that inserts triples into an existing graph, the permissions you passed in are ignored. If you're copying a graph from one graph to another, the permissions are ignored in favor of “transferring” the data and the permissions from one graph to another.

> **Note:** Graph-level security is enforced for all semantic operations using SPARQL or SPARQL Update, via XQuery or JavaScript, and includes semantic REST functions.

Your default user permissions are set by the MarkLogic admininstrator. These are the same default permissions for document creation that are discussed in Default Permissions in the *Security Guide*.

To see what permissions are currently on a graph, use `sem:graph-get-permissions`:

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

sem:graph-get-permissions(
  sem:iri("MyGraph"))

=>

<sec:permission xmlns:sec="http://marklogic.com/xdmp/security">
  <sec:capability>read</sec:capability>
  <sec:role-id>5995163769635647336</sec:role-id>
</sec:permission>

<sec:permission xmlns:sec="http://marklogic.com/xdmp/security">
<sec:capability>update</sec:capability>
<sec:role-id>5995163769635647336</sec:role-id>
</sec:permission>
```

This returns the two capabilities set on in this graph: read and update. If you have the role with the ID `5995163769635647336`, you will be able to read information in this graph, you will be able to see the graph and the triples in the graph. If you have the role with the ID `5995163769635647336`, you will be able to update the graph. You must have read capability for the graph to use `sem:graph-get-permissions`.

Note: In a new database, the graph document for the default graph does not exist yet. Once you insert triples into this database, the default graph is created.

To set permissions on a graph, use `sem:graph-set-permissions`:

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

declare namespace sec = "http://marklogic.com/xdmp/security";

sem:graph-set-permissions(sem:iri("MyGraph"),
  (xdmp:permission("sparql-update-role", "update"))
```

This will set the permissions on the graph and the triples in the graph. If the specified IRI does not exist, the graph will be created. You must have update permissions for the graph to set the permissions.

To add permissions to a graph, use `sem:graph-add-permissions`

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

declare namespace sec = "http://marklogic.com/xdmp/security";

sem:graph-add-permissions(sem:iri("MyGraph"),
  (xdmp:permission("sparql-read-role", "read"),))
```

This will add read permissions for the `sparql-read-role` to the graph and to the triples in the graph. If the graph named by the IRI does not exist, the graph will be automatically created. You must have update permissions for the graph to add permissions to an existing graph.

Note: A graph that is created by a non-admin user (that is, by any user who does not have the admin role) must have at least one update permission for the graph, otherwise the creation of the graph will return an `XDMP-MUSTHAVEUPDATE` exception.

To remove permissions, use `sem:graph-remove-permissions`:

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

declare namespace sec = "http://marklogic.com/xdmp/security";

sem:graph-remove-permissions(sem:iri("MyGraph"),
  (xdmp:permission("sparql-read-role", "read"),))
```

This removes the read permission on "MyGraph" for `sparql-read-role`. You must have update permissions for the graph to remove permissions. If the graph does not exist, this will result in an error.

8.4 Data Operations with SPARQL Update

These data operations are part of SPARQL Update: `INSERT DATA`, `DELETE DATA`, `DELETE/INSERT`, `LOAD`, and `CLEAR`. Data operations involve triple data contained in graphs.

SPARQL Update includes these commands and options for working with data in RDF graphs:

Command	Options	Description
INSERT DATA	QuadData, WITH, GRAPH	Inserts triples into a graph. If no graph is specified, the default graph will be used.
DELETE DATA	QuadData	Deletes triples from a graph, as specified by QuadData. If no graph is specified, deletes from all in-scope graphs.
DELETE..INSERT WHERE	WITH, IRIref, USING, NAMED, WHERE, DELETE, INSERT	Remove or add triples from/to the Graph Store based on bindings for a query pattern specified in a `WHERE` clause.
DELETE WHERE	WITH, IRIref, USING, NAMED, WHERE, DELETE, INSERT	Remove triples from the Graph Store based on bindings for a query pattern specified in a `WHERE` clause. `DELETE WHERE` is `DELETE..INSERT WHERE` with a missing `INSERT` (which is optional).

Command	Options	Description
INSERT WHERE	WITH, IRIref, USING, NAMED, WHERE, DELETE, INSERT	Add triples to the Graph Store based on bindings for a query pattern specified in a `WHERE` clause. `INSERT WHERE` is `DELETE..INSERT WHERE` with a missing `DELETE` (which is optional).
CLEAR	SILENT, (GRAPH IRIref | DEFAULT | NAMED | ALL)	Removes all the triples in the specified graph.

You can run multiple statements separated by semicolons (;) in one SPARQL Update operation in the same transaction, and you can include them in the same request. It is important to note that each statement can see the result of the previous statements.

Note: SPARQL Update operations return the empty sequence.

Other ways to load triples into MarkLogic: mlcp, `sem:rdf-load`, or the HTTP REST endpoints. See “Loading Triples with mlcp” on page 44, “sem:rdf-load” on page 53, “Addressing the Graph Store” on page 57, and “Loading Semantic Triples” on page 37. For bulk loading, mlcp is the preferred method.

8.4.1 INSERT DATA

`INSERT DATA` adds triples specified in the request into a graph. The syntax for `INSERT DATA` is:

```
INSERT DATA QuadData
(GRAPH VarOrIri) ? {TriplesTemplates}
```

The *`QuadData`* parameter is made up of sets of triple patterns (*`TriplesTemplates`*), which can optionally be wrapped in a `GRAPH` block.

Note: All of the managed triples from a triples document will go into the default graph unless you specify the destination graph when inserting them using SPARQL Update.

If you are inserting triples into a new graph, create the graph with the permissions you want, specified through `sem:sparql-update`. If you do not specify permissions on the graph, the graph will be created with your default permissions. To manage the permissions of the graph, use `sem:graph-add-permissions`. See “Graph-Level Security” on page 175 for more about permissions and security.

For example, this update uses sem:graph-add-permissions to add update permissions to the `sparql-update-role` to `update` for `<My Graph>`, and inserts three triples into that graph:

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

sem:graph-add-permissions(sem:iri("MyGraph"),
xdmp:permission("sparql-update-role", "update"));
sem:sparql-update('

PREFIX exp: <http://example.org/marklogic/people>
PREFIX pre: <http://example.org/marklogic/predicate>

INSERT DATA {
  GRAPH <MyGraph>{
  exp:John_Smith pre:livesIn "London" .
  exp:Jane_Smith pre:livesIn "London" .
  exp:Jack_Smith pre:livesIn "Glasgow" .
  }}
')
```

If no graph is described in *`QuadData`*, the default graph is assumed (`http://marklogic.com/semantics#default-graph`). If data is inserted into a graph that does not exist in the Graph Store, a new graph is created for the data with the user's permissions.

This example uses `INSERT DATA` to insert a triple into a default graph and then insert three triples into a graph named "BOOKS".

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

sem:sparql-update('
PREFIX dc: <http://marklogic.com/dc/elements/1.1/>
INSERT DATA
{
  <http://example/book0> dc:title "A default book"
  GRAPH <BOOKS> {<http://example/book1> dc:title "A new book" }
  GRAPH <BOOKS> {<http://example/book2> dc:title "A second book" }
  GRAPH <BOOKS> {<http://example/book3> dc:title "A third book" }
 }
');
```

This example will delete any book titled “A new book”in the graph “BOOKS” and insert the title “Inside MarkLogic Server” in its place:

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

sem:sparql-update('
PREFIX dc: <http://marklogic.com/dc/elements/1.1/>
WITH <BOOKS>
DELETE {?b dc:title "A new book"}
INSERT {?b dc:title "Inside MarkLogic Server" }
WHERE {
?b dc:title "A new book".
}')
```

The `WITH` keyword means to use this graph (`<BOOKS>`) in every clause.

8.4.2 DELETE DATA

The `DELETE DATA` operation removes some triples specified in the request, if the respective graphs in the Graph Store contain them. The syntax for `DELETE DATA` is:

```
DELETE DATA QuadData
```

The *`QuadData`* parameter specifies the triples to be removed. If no graph is described in *`QuadData`*, then the default graph is assumed.

Any MarkLogic-managed triple that matches subject, predicate, and object on any of the triples specified in *`QuadData`* will be deleted. If a graph is specified in *`QuadData`*, the scope of deletion is limited to the triples in that graph; otherwise, the scope of deletion is limited to the triples in default graph.

This example deletes triples that match “true” and “Retail/Wholesale” from the `<http://marklogic.com/semantics/COMPANIES100A/>` graph.

```
PREFIX demov: <http:demo/verb#>
PREFIX demor: <http:demo/resource#>

DELETE DATA
{
  GRAPH <http://marklogic.com/semantics/COMPANIES100A/>
    {
    demor:COMPANY100 demor:listed "true" .
    demor:COMPANY100 demov:industry "Retail/Wholesale" .
  }
}
```

An error will be thrown if a DELETE DATA operation includes variables or blank nodes. The deletion of non-existing triples has no effect. Any triples specified in *QuadData* that do not exist in the Graph Store are ignored.

8.4.3 DELETE..INSERT WHERE

The DELETE..INSERT WHERE operation is used to remove and/or add triples from/to the Graph Store based on bindings for a query pattern specified in a WHERE clause. You can use DELETE..INSERT WHERE to specify a pattern to match against and then delete and/or insert triples.

See http://www.w3.org/TR/sparql11-update/#updateLanguage for details.

To delete triples according to a specific pattern, use the DELETE..INSERT WHERE construct without the optional INSERT clause. If you do not specify a graph, you will insert into or delete triples from the default graph (http://marklogic.com/semantics#default-graph), also called the unnamed graph. The syntax for DELETE..INSERT WHERE is:

```
( WITH IRIref )?
( ( DeleteClause InsertClause? ) | InsertClause )
( USING ( NAMED )? IRIref )*
WHERE GroupGraphPattern
DeleteClause ::= DELETE  QuadPattern
InsertClause ::= INSERT  QuadPattern
```

This operation identifies data using the WHERE clause, which is used to determine matching sequences of bindings for a set of variables. The bindings for each solution are then substituted into the DELETE template to remove triples, and then in the INSERT template to create new triples.

If an operation tries to insert into a graph that does not exist, that graph will be created. DELETE and INSERT run in the same transaction, and obey MarkLogic security rules.

In this example each triple containing “Healthcare/Life Sciences” will be deleted and two triples and two triples will be inserted, one for “Healthcare” and a second one for “Life Sciences”.

```
PREFIX demov: <http:demo/verb#>

WITH <http://marklogic.com/semantics/COMPANIES100A/>
DELETE
{
  ?company demov:industry "Healthcare/Life Sciences" .
}
INSERT
{
  ?company demov:industry "Healthcare" .
  ?company demov:industry "Life Sciences" .
}
WHERE {
  ?company demov:industry "Healthcare/Life Sciences" .}
```

Note that the DELETE and INSERT are independent of each other.

8.4.4 DELETE WHERE

The DELETE WHERE operation is used to remove triples from the Graph Store based on bindings for a query pattern specified in a WHERE clause. DELETE WHERE is DELETE..INSERT WHERE with a missing INSERT, which is optional. You can use DELETE WHERE to specify a pattern to match against and then delete the matching triples.

To delete triples according to a specific pattern, use the DELETE WHERE construct without the optional INSERT clause. If you do not specify a graph, you will delete triples from the default graph (http://marklogic.com/semantics#default-graph), also called the unnamed graph.

The syntax for DELETE WHERE is:

```
( WITH IRIref )?
( ( DeleteClause InsertClause? ) | InsertClause )
( USING ( NAMED )? IRIref )*
WHERE GroupGraphPattern
DeleteClause ::= DELETE  QuadPattern
InsertClause ::= INSERT  QuadPattern
```

This example of DELETE WHERE deletes the sales data for any companies with less than 100 employees from the graph <http://marklogic.com/semantics/COMPANIES100A/>:

```
PREFIX demov: <http:demo/verb#>
PREFIX vcard: <http:www.w3c.org/2006/vcard/ns#>

WITH <http://marklogic.com/semantics/COMPANIES100A/>
DELETE
{
  ?company demov:sales ?sales .
}
WHERE {
  ?company a vcard:Organization .
  ?company demov:employees ?employees .
}
  FILTER ( ?employees < 100 )
```

8.4.5 INSERT WHERE

The INSERT WHERE operation is used to add triples to the Graph Store based on bindings for a query pattern specified in a WHERE clause. INSERT WHERE is DELETE..INSERT WHERE with a missing DELETE, which is optional.You can use INSERT WHERE to specify a pattern to match against and then insert triples based on that match.

To insert triples according to a specific pattern, use the INSERT WHERE construct without the optional DELETE clause. If you do not specify a graph, you will insert triples into the default graph (http://marklogic.com/semantics#default-graph), also called the unnamed graph.

The syntax for `INSERT WHERE` is:

```
( WITH IRIref )?
( ( DeleteClause InsertClause? ) | InsertClause )
( USING ( NAMED )? IRIref )*
WHERE GroupGraphPattern
DeleteClause ::= DELETE  QuadPattern
InsertClause ::= INSERT  QuadPattern
```

This example of `INSERT WHERE` finds each company in New York, USA and adds `State="NY"` and `deliveryRegion="East Coast"`.

```
PREFIX demov: <http:demo/verb#>
PREFIX vcard: <http:www.w3c.org/2006/vcard/ns#>

WITH <http://marklogic.com/semantics/sb/COMPANIES100A/>
INSERT
{
  ?company demov:State "NY" .
  ?company demov:deliveryRegion "East Coast"
}
WHERE {
  ?company a vcard:Organization .
  ?company vcard:hasAddress [
    vcard:region "New York" ;
    vcard:country-name "USA"]
}
```

8.4.6 CLEAR

The `CLEAR` operation removes all of the triples in the specified graph(s). The syntax for `CLEAR` is:

```
CLEAR (SILENT)? (GRAPH IRIref | DEFAULT | NAMED | ALL )
```

The `GRAPH IRIref` option is used to remove all triples from the graph specified by `GRAPH IRIref`. The `DEFAULT` option is used to remove all the triples from the default graph of the Graph Store. The `NAMED` option is used to remove all of the triples in all of the named graphs of the Graph Store, and the `ALL` option is used to remove all triples from all graphs of the Graph Store.

For example:

```
CLEAR GRAPH <http://marklogic.com/semantics/COMPANIES100A/> ;
```

This operation removes all of the triples from the graph. The graph and its metadata (such as permissions) will be kept after the graph is cleared. The `CLEAR` will fail if the specified graph does not exist. If the `SILENT` option is used, the operation will not return an error.

8.5 Bindings for Variables

Binding variables can be used as values in `INSERT DATA`, and `DELETE DATA`, and are passed in as arguments to `sem:sparql-update`.

To create bindings for variables to be used with SPARQL Update, you create an XQuery or JavaScript function to map the bindings and then pass in the bindings as part of a call to `sem:sparql-update`.

In this example we create a function, assign the variables, build a set of bindings, and then use the bindings to insert repetitive data into the triple store using `INSERT DATA`:

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

declare function
local:add-player-triples($id as xs:integer,
    $lastname as xs:string,
    $firstname as xs:string,
    $position as xs:string,
    $team as xs:string,
    $number as xs:integer
  )
{
let $query := '
PREFIX bb: <http://marklogic.com/baseball/players/>
PREFIX xs: <http://www.w3.org/2001/XMLSchema#>

INSERT DATA
{
GRAPH <PlayerGraph>
{
?playertoken bb:playerid ?id .
?playertoken bb:lastname ?lastname .
?playertoken bb:firstname ?firstname .
?playertoken bb:position ?position .
?playertoken bb:number ?number .
?playertoken bb:team ?team .
}
}

'
let $playertoken := fn:concat("bb:", $id)
let $player-map := map:entry("id", $id)
let $put := map:put($player-map, "playertoken", $playertoken)
let $put := map:put($player-map, "lastname", $lastname)
let $put := map:put($player-map, "firstname", $firstname)
let $put := map:put($player-map, "position", $position)
let $put := map:put($player-map, "number", $number)
let $put := map:put($player-map, "team", $team)

return sem:sparql-update($update, $player-map)
```

```
};

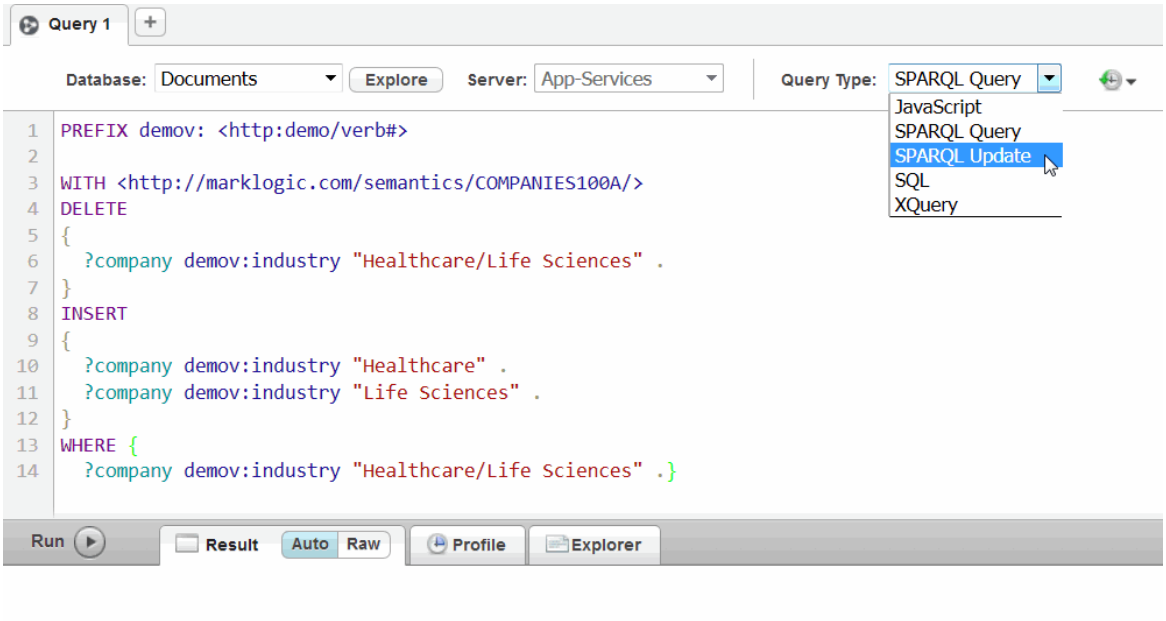
local:add-player-triples(417, "Doolittle", "Sean", "pitcher", 62,
"Athletics"),
local:add-player-triples(215, "Abad", "Fernando", "pitcher", 56,
"Athletics"),
local:add-player-triples(109, "Kazmir", "Scott", "pitcher", 26,
"Athletics"),
```

The order of the variables does not matter because of the mapping.

This same pattern can be used with `LIMIT` and `OFFSET` clauses. Bindings for variables can be used with SPARQL Update (`sem:sparql-update`), SPARQL (`sem:sparql`), and SPARQL values (`sem:sparql-values`). See “Using Bindings for Variables” on page 135 for an example using bindings for variables with SPARQL using `LIMIT` and `OFFSET` clauses.

8.6 Using SPARQL Update with Query Console

You can use SPARQL Update with Query Console. It is listed as a Query type in the Query Console drop-down menu.



Query Console highlights SPARQL Update keywords as you create your query. You can run your SPARQL Update queries in the Query Console the same way you run SPARQL Queries.

8.7 Using SPARQL Update with XQuery or Server-Side JavaScript

You can execute SPARQL Updates with `sem:sparql-update` in XQuery and `sem.sparqlUpdate` in Javascript. For details about the function signatures and descriptions, see the Semantics documentation under XQuery Library Modules in the *MarkLogic XQuery and XSLT Function Reference* and in the *MarkLogic Server-Side JavaScript Function Reference*.

Note: Although some of the semantics functions are built-in, others are not. We therefore recommend that you import the Semantics API library into every XQuery module or JavaScript module that uses the Semantics API.

Using XQuery, the import statement is:

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

For Javascript, the import statement is:

```
var sem = require("/MarkLogic/semantics.xqy");
```

Here is a complex query using SPARQL Update with XQuery. The query selects a random player from the Athletics baseball team, formerly with the Twins, and deletes that player the Athletics:

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

let $query := '
PREFIX bb:  <http://marklogic.com/baseball/players#>
PREFIX bbr:  <http://marklogic.com/baseball/rules#>
PREFIX xs: <http://www.w3.org/2001/XMLSchema#>

WITH <Athletics>
DELETE
{
  ?s ?p ?o .
}
INSERT
{
  ?s ?p bbr:Twins .
}
WHERE
  {
    ?s ?p ?o .
    {
      SELECT (max(?s1) as ?key) (count(?s1) as ?inner_numbers)
      WHERE
     {
       ?s1 bb:number ?o1 .
     }
    }
FILTER (?s = ?key)
```

```
FILTER (?p = bb:team)
}
return sem:sparql-update($query)
```

8.8 Using SPARQL Update with REST

SPARQL Update can be used with REST in client applications to manage graphs and triple data via the SPARQL endpoint. See “SPARQL Update with the REST Client API” on page 211 for information about using SPARQL Update with REST.

For more information about using SPARQL with Node.js client applications, see Working With Semantic Data in the *Node.js Application Developer’s Guide*. For semantic client applications using Java, you can find the Java Client API on GitHub at http://github.com/marklogic/java-client-api or get it from the central Maven repository.

9.0 Using Semantics with the REST Client API

This section describes how to use MarkLogic Semantics with the REST Client API to view, query, and modify triples and graphs using REST (Representational State Transfer) over HTTP. The REST Client API enables a client application to perform SPARQL queries and updates. A MarkLogic SPARQL endpoint (`/v1/graphs/sparql`) is available to use with SPARQL query and SPARQL Update to access triples in MarkLogic. The `/v1/graphs/sparql` service is a compliant SPARQL endpoint, as defined by the SPARQL 1.1 Protocol: http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/#terminology. If your client requires configuration of a SPARQL Update or Query endpoint, use this service.

A SPARQL endpoint is a web service that implements the SPARQL protocol and can respond to SPARQL queries. RDF data published to the web using a SPARQL endpoint can be queried to answer a specific question, instead of downloading all of the data. If you have an application that does standard queries and updates over a SPARQL endpoint, you can point the application to this endpoint.

The SPARQL endpoint URL is addressed as:

```
http://host:port/v1/graphs/sparql
```

The `graph` endpoint is used for CRUD procedures on graphs; creating, reading, updating, and deleting graphs. The URL is addressed as:

```
http://host:port/v1/graphs
```

The `things` endpoint is used for viewing content in the database. The URL is addressed as:

```
http://host:port/v1/graphs/things
```

The following table shows the supported operations available for Semantics (viewing, querying, inserting, or deleting content):

Operation	Method	Description
`/v1/graphs/sparql`		
Retrieve	GET	Perform a SPARQL query on the database.
Create/Retrieve	POST	Perform a SPARQL query or SPARQL Update on one or more graphs. (These two operations are mutually exclusive.)
`/v1/graphs`		
Retrieve	GET	Retrieve the contents or permissions metadata of a graph, or a list of available graph URIs.

Operation	Method	Description
Merge	POST	Merge N-quads into the triple store, merge other types of triples, or new permissions into a named graph or the default graph.
Create/Replace	PUT	Create or replace quads in the triple store; or create or replace other kinds of triples in a named graph or the default graph; or replace the permissions on a named graph or the default graph.
Return	HEAD	Returns the same headers as an equivalent GET on the `/graphs` service.
Delete	DELETE	Remove triples in a named graph or the default graph, or remove all graphs from the triple store.
`/v1/graphs/things`		
Retrieve	GET	Retrieve a list of all graph nodes in the database, or a specified set of nodes.

For more information about usage and parameters for a service, see the REST Client APIs for Semantics.

This chapter includes the following sections:

- Assumptions
- Specifying Parameters
- Supported Operations for the REST Client API
- Serialization
- Examples Using curl and REST
- Response Output Formats
- SPARQL Query with the REST Client API
- SPARQL Update with the REST Client API
- Listing Graph Names with the REST Client API
- Exploring Triples with the REST Client API
- Managing Graph Permissions

To use the SPARQL endpoint or graphs endpoints with SPARQL query, you must have the rest-reader privilege, along with any security requirements for your environment. To use SPARQL Update with the SPARQL endpoint or graphs endpoints, you must have the rest-writer privilege. See Controlling Access to Documents Created with the REST API in the *REST Application Developer's Guide* for more information about permissions.

9.1 Assumptions

To follow along with the examples later in this section the following assumptions are made:

- You have access to the GovTrack dataset. For details, see "Preparing to Run the Examples" on page 129. If you do not have access to the GovTrack data or prefer to use your own data, you can modify queries to fit your data.
- You have `curl` or an equivalent command-line tool for issuing HTTP requests is installed.

  Note: Though the examples rely on `curl`, you can use any tool capable of sending HTTP requests. If you are not familiar with `curl` or do not have `curl` on your system, see the Introduction to the curl Tool in the *REST Application Developer's Guide*.

9.2 Specifying Parameters

A variety of parameters can be used with REST services. The complete list can be found in the REST Client APIs, for instance `POST:/v1/graphs/sparql`. This section describes a selection of those parameters that can be used with SPARQL query and/or SPARQL Update.

For the parameters, "*" and "?" both imply that a parameter is optional. "*" means that you can use a parameter 0 or more times and "?" means that you can use a parameter 0 or 1 times.

9.2.1 SPARQL Query Parameters

Some of the parameters supported for SPARQL query on the SPARQL endpoint, using `POST:/v1/graphs/sparql` or `GET:/v1/graphs/sparql`, include:

- `query` - SPARQL query to execute
- `default-graph-uri*` - The URI of the graph or graphs to use as the default graphs in the query operation. This is addressed as `http://host:port/v1/graphs/sparql?default-graph-uri=<default-graph-uri*>`
- `named-graph-uri*` - The URI of the graph or graphs to include in the query operation. This is addressed as `http://host:port/v1/graphs/sparql?named-graph-uri=<named-graph-uri*>`

  The "`*`" indicates that one or more `default-graph-uri` or `named-graph-uri` parameters can be specified. The *`named-graph-uri`* parameter is used with `FROM NAMED` and `GRAPH` in queries to specify the IRI(s) to be substituted for a name within particular kinds of queries. You can have one or more *`named-graph-uri`* specified as part of a query.

- `database?` - The database on which to perform the query.
- `base?` - The initial base IRI for the query.
- `bind:{name}*` - A binding name and value. This format assumes that the type of the bind variable is an IRI.

- `bind:{name}:{type}*` - A binding name, type, and value. This parameter accepts an XSD type, for example “string”, “date” or “unsignedLong”.
- `bind:{name}@{lang}*` - A binding name, language tag, and value. Use this pattern to bind to language-tagged strings.
- `txid?` - The transaction identifier of the multi-statement transaction in which to service this request. Use the `/transactions` service to create and manage multi-statement transactions.
- `start?` - The index of the first result to return. Results are numbered beginning with 1. The default is 1.
- `pageLength?` - The maximum number of results to return in this request.

These optional search query parameters can be used to constrain which documents will be searched with the SPARQL query:

- `q?` - A string query.
- `structuredQuery?` - A structured search query string, a serialized representation of a `search:query` element.
- `options?` - The name of query options.

If you do not specify a graph name with a query, the `UNION` of all graphs will be queried. If you specify `default-graph-uri`, one or more graph names that you specify will be queried (this is *not* the “default” graph that contains the unnamed triples). You can also query `http://marklogic.com/semantics#default-graph`, where unnamed triples are stored.

Any valid IRI can be used for these graph names (for example, `/my_graph/` or `http://www.example.com/rdf-graph-store/`). The `default-graph-uri` is used to specify one or more default graphs to query as part of the operation, and the `named-graph-uri` can specify one or more additional graphs to use in the operation. If no dataset is defined, the dataset will include all triples (the `UNION` of all graphs).

If you specify a dataset in both the request parameters and the query, the dataset defined with `named-graph-uri` or `default-graph-uri` takes precedence. When you specify more than one `default-graph-uri` or `named-graph-uri` in a query via the REST Client API, the format will be `http://host:port/v1/graphs/sparql?named-graph-uri=<named-graph-uri*>` for each graph named in the query.

For example, this is a simple REST request to send the SPARQL query in the `bills.sparql` file and return the results as JSON:

```
curl --anyauth --user admin:admin -i -X POST \
--data-binary @./bills.sparql \
-H "Content-type: application/sparql-query" \
-H "Accept: application/sparql-results+json" \
http://localhost:8000/v1/graphs/sparql
```

```
=>
HTTP/1.1 200 OK
Content-type: application/sparql-results+json; charset=UTF-8
Server: MarkLogic
Content-Length: 1268
Connection: Keep-Alive
Keep-Alive: timeout=5

{"head":{"vars":["bill","title"]},
"results":{"bindings":[
   {"bill":{"type":"uri",
   "value":"http://www.rdfabout.com/rdf/usgov/congress/108/
            bills/h1171"},
   "title":{"type":"literal", "value":"H.R. 108/1171: Iris Scan
            Security Act of 2003",
   "datatype":"http://www.w3.org/2001/XMLSchema#string"}},
   {"bill":{"type":"uri",
   "value":"http://www.rdfabout.com/rdf/usgov/congress/108/
            bills/h1314"},
   "title":{"type":"literal", "value":"H.R. 108/1314: Screening
            Mammography Act of 2003",
   "datatype":"http://www.w3.org/2001/XMLSchema#string"}},
   {"bill":{"type":"uri",
   "value":"http://www.rdfabout.com/rdf/usgov/congress/108/
            bills/h1384"},
   "title":{"type":"literal", "value":"H.R. 108/1384: To amend the
            Railroad Retirement Act of 1974 to eliminate a limitation
            on benefits.",
   "datatype":"http://www.w3.org/2001/XMLSchema#string"}},
   {"bill":{"type":"uri",
   "value":"http://www.rdfabout.com/rdf/usgov/congress/108/
            bills/h1418"},
   "title":{"type":"literal", "value":"H.R. 108/1418: Veterans'
            Housing Equity Act",
   "datatype":"http://www.w3.org/2001/XMLSchema#string"}},
...
}]}}
```

Note: In the command line example above, long lines have been broken into multiple lines using the UNIX line continuation character '\' and extra line breaks have been added for readability. Extra line breaks for readability have been added in the results.

9.2.2 SPARQL Update Parameters

In addition to the query parameters, these parameters can be used with SPARQL Update on the `POST:/v1/graphs/sparql` endpoint:

- `update` - The URL-encoded SPARQL Update operation. Only use this parameter when you put the request parameters in the request body and use `application/x-www-form-urlencoded` as the request content type.

- `using-graph-uri*` - The URI of the graph or graphs to address as part of a SPARQL Update operation. This is addressed as `http://host:port/v1/graphs/sparql?using-graph-uri=<using-graph-uri*>`
- `using-named-graph-uri*` - The URI of a named graph or graphs to address as part of a SPARQL update operation. This is addressed as `http://host:port/v1/graphs/sparql?using-named-graph-uri=<using-named-graph-uri*>`
- `perm:{role}*` - Assign permissons to the inserted graph(s), and the permission has a role and a capability. When you insert a new graph, you can set its permissions to allow a certain capability for a certain role. Valid values for the permissions: `read`, `update`, `execute`. These permissions only apply to newly created graphs. See "Managing Graph Permissions" on page 217 for more about permissions.
- `txid?` - The transaction identifier of the multi-statement transaction in which to service this request. Use the `/transactions` service to create and manage multi-statement transactions.
- `database?` - The database on which to perform the query.
- `base?` - The initial base IRI for the query.
- `bind:{name}*` - A binding name and value. This format assumes that the type of the bind variable is an IRI.
- `bind:{name}:{type}*` - A binding name, type, and value. This parameter accepts an XSD type, for example "string", "date" or "unsignedLong".
- `bind:{name}@{lang}*` - A binding name, language tag, and value. Use this pattern to bind to language-tagged strings.

See "Target RDF Graph" on page 91 for more information about graphs. See Querying Triples in the *REST Application Developer's Guide* for more infomation about using the REST Client API with RDF triples.

9.3 Supported Operations for the REST Client API

For the `/v1/graphs/sparql` endpoint, these operations are supported:

Operation	Description	Method	Privilege
Retrieve	Evaluates a SPARQL query to retrieve a named graph.	GET	rest-reader

Operation	Description	Method	Privilege
Create/ Retrieve	Evaluates a SPARQL query as a parameter or URL-encoded as part of the `POST` body. Using SPARQL Update, `POST` merges triples into a named graph when used as parameter, or as a URL-encoded SPARQL Update in the `POST` body. SPARQL query and SPARQL Update operations are mutually exclusive.	POST	rest-writer (SPARQL query) rest-writer (SPARQL Update)

Note: For SPARQL Update, only `POST:/v1/graphs/sparql` is suported.

There is also a `/v1/graphs` endpoint and a `/v1/graphs/things` endpoint to access and view RDF data. For the `/v1/graphs` endpoint, these verbs are supported:

Operation	Description	Method	Privilege
Retrieve	Retrieve a graph or a list of available graph URIs.	GET	rest-reader
Merge	Without parameters, merges quads into the triple store. With graph or default, merges triples into a named graph or the default graph.	POST	rest-writer
Create/ Replace	Without parameters, creates or replaces quads. With `default` or `graph`, creates or replaces triples in a named graph or the default graph. Using `PUT` with an empty graph, will delete the graph.	PUT	rest-writer
Delete	Without parameters, removes all graphs from the triple store. With `graph` or `default`, removes triples in a named graph or the default graph.	DELETE	rest-writer
Return	Returns the same headers as an equivalent GET on the `/graphs` service.	HEAD	rest-reader

And for the `/v1/graphs/things` endpoint, the verb `GET` is supported for REST requests:

Operation	Description	Method	Privilege
Retrieve	Retrieves a list of all graph nodes in the database, or a specified set of nodes.	GET	rest-reader

9.4 Serialization

Serialization of RDF data occurs whenever it is loaded, queried, or updated. The data can be serialized in a variety of ways. The supported serializations are shown in the table shown in "Supported RDF Triple Formats" on page 38.

Several types of optimized serialization are available for SPARQL results (solutions - sets of bindings) and RDF (triples) over REST. Using these serializations in your interactions will make them faster. These serializations specify a MIME type for the input and output format. Formats are specified as part of the accept headers of the REST request.

We recommend using one of the following choices for optimized serialization of SPARQL results when using the `/v1/graphs/sparql` endpoint:

Format	SPARQL Query Type `/v1/graphs/sparql`	MIME Type/Accept Header
`json`	`SELECT` or `ASK`	`application/sparql-results+json`
`csv`	`SELECT` or `ASK`	`application/sparql-results+csv`
`n-triples`	`CONSTRUCT` or `DESCRIBE`	`application/n-triples`

For `CONSTRUCT` or `DESCRIBE` all of the supported triples formats are supported. See the table in "Supported RDF Triple Formats" on page 38.

> **Note:** N-Quads and TriG formats are quad formats, not triple formats, and REST does not serialize quads.

For optimized RDF results (triple or quads), choose one of these serialization options when using the `/v1/graphs` endpoint:

Format	RDF Query Type `/v1/graphs`	MIME Type/Accept Header
`n-triples`	`CONSTRUCT` or `DESCRIBE`	`application/n-triples`
`n-quads`	`SELECT` or `ASK`	`application/quads`

This information is shown in a different way in "SPARQL Query Types and Output Formats" on page 200. For more about serialization, see Setting Output Options for an HTTP Server in the *Administrator's Guide*.

9.4.1 Unsupported Serialization

A `GET` or `POST` request for a response in an unsupported serialization yields a "406 Not Acceptable" error. If the SPARQL payload fails to parse, the response yields a "400 Bad Request" error.

For example:

```
<rapi:error xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:status-code>400</rapi:status-code>
  <rapi:status>Bad Request</rapi:status>
  <rapi:message-code>RESTAPI-INVALIDCONTENT</rapi:message-code>
  <rapi:message>RESTAPI-INVALIDCONTENT: (err:FOER0000) Invalid
content:
  Unexpected Payload: c:\space\example.ttl</rapi:message>
</rapi:error>
```

For more about the REST Client API error handling, see Error Reporting in the *REST Application Developer's Guide*.

9.5 Examples Using curl and REST

These two examples use `curl` with cygwin (Linux) and Windows to do Semantic queries over REST. This SPARQL query is encoded and used in the examples:

```
SELECT *
WHERE {
 ?s ?p ?o }
```

For readability, the character (" \ ") is used in the cygwin (Linux) example to indicate a line break. The actual request must be entered on one continuous line. The query looks like this:

```
curl --anyauth --user user:password
"http://localhost:8000/v1/graphs/sparql" \
-H "Content-type:application/x-www-form-urlencoded" \
-H "Accept:application/sparql-results+xml" \
-X POST --data-binary 'query=SELECT+*+WHERE+{+%3fs+%3fp+%3fo+}'

=>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
<head>
  <variable name="s"/>
  <variable name="p"/>
  <variable name="o"/>
</head>
<results>
  <result>
    <binding name="s"><uri>http://example/book1/</uri></binding>
 <binding
name="p"><uri>http://purl.org/dc/elements/1.1/title</uri></binding>
 <binding name="o"><literal>A new book</literal></binding>
  </result>
```

```
  <result>
    <binding name="s">
      <uri>http://example/book1/</uri>
    </binding>
    <binding name="p">
      <uri>http://purl.org/dc/elements/1.1/title</uri>
    </binding>
    <binding name="o">
      <literal>Inside MarkLogic Server</literal>
    </binding>
  </result>
  <result>
    <binding name="s">
      <uri>http://www.w3.org/2000/01/rdf-schema#subClassOf</uri>
    </binding>
    <binding name="p">
      <uri>http://www.w3.org/2000/01/rdf-schema#domain</uri>
    </binding>
    <binding name="o">
      <uri>http://www.w3.org/2000/01/rdf-schema#Class</uri>
    </binding>
  </result>
  <result>
    <binding name="s">
      <uri>http://www.w3.org/2000/01/rdf-schema#subClassOf</uri>
    </binding>
    <binding name="p">
      <uri>http://www.w3.org/2000/01/rdf-schema#range</uri>
    </binding>
    <binding name="o">
      <uri>http://www.w3.org/2000/01/rdf-schema#Class</uri>
    </binding>
  </result>
</results></sparql>
```

Note: The results have been formatted for clarity.

In the Windows example, the character (“ ^ ”) is used to indicate a line break for readability. The actual request must be entered on one continuous line. The Windows query looks like this:

```
curl --anyauth --user user:password
"http://localhost:8000/v1/graphs/sparql" ^
  -H "Content-type:application/x-www-form-urlencoded" ^
  -H "Accept:application/sparql-results+xml" ^
  -X POST --data-binary 'query=SELECT+*+WHERE+{+%3fs+%3fp+%3fo+}'
=>

<sparql xmlns="http://www.w3.org/2005/sparql-results#">
<head>
  <variable name="s"/>
  <variable name="p"/>
  <variable name="o"/>
</head>
<results>
```

```
<result>
  <binding name="s">
    <uri>http://example.org/marklogic/people/Jack_Smith</uri>
  </binding>
  <binding name="p">
    <uri>http://example.org/marklogic/predicate/livesIn</uri>
  </binding>
  <binding name="o"><literal>Glasgow</literal>
  </binding>
</result>
<result>
  <binding name="s">
    <uri>http://example.org/marklogic/people/Jane_Smith</uri>
  </binding>
  <binding name="p">
    <uri>http://example.org/marklogic/predicate/livesIn</uri>
  </binding>
  <binding name="o">
    <literal>London</literal>
  </binding>
</result>
<result>
  <binding name="s">
    <uri>http://example.org/marklogic/people/John_Smith</uri>
  </binding>
  <binding name="p">
    <uri>http://example.org/marklogic/predicate/livesIn</uri>
  </binding>
  <binding name="o">
    <literal>London</literal>
  </binding>
</result>
</results></sparql>
```

Note: The results have been formatted for clarity.

9.6 Response Output Formats

This section describes the header types and response output formats available when using SPARQL endpoints with the REST Client API. Examples of results in different formats are included. These topics are covered:

- SPARQL Query Types and Output Formats
- Example: Returning Results as XML
- Example: Returning Results as JSON
- Example: Returning Results as HTML
- Example: Returning Results as CSV
- Example: Returning Results as N-triples
- Example: Returning a Boolean as XML or JSON

9.6.1 SPARQL Query Types and Output Formats

When you query the SPARQL endpoint with REST Client APIs (`GET:/v1/graphs/sparql` or `POST:/v1/graphs/sparql`), you can specify the result output format. The response type format depends on the type of query and the MIME type in the HTTP Accept header.

A SPARQL `SELECT` query can return results as XML, JSON, HTML, or CSV, while a SPARQL `CONSTRUCT` query can return the results as triples in N-Triples or N-Quads format, or XMLor JSON triples in any of the supported triples formats. A SPARQL `DESCRIBE` query returns triples in XML, N-Triples, or N-Quads format describing the triples found by the query. Using SPARQL `ASK` query will return a boolean (`true` or `false`) in either XML or JSON. See “Types of SPARQL Queries” on page 82 for more information about query types.

This table describes the MIME types and Accept Header/Output formats (MIME type) for different types of SPARQL queries.

Query Type	Format	Accept Header MIME Type
`SELECT` or `ASK` Returns SPARQL results - solutions	`xml`	`application/sparql-results+xml` See “Example: Returning Results as XML” on page 201 and “Example: Returning a Boolean as XML or JSON” on page 206.
	`json`	`application/sparql-results+json` See “Example: Returning Results as JSON” on page 202.
	`html`	`text/html` See “Example: Returning Results as HTML” on page 203.
	`csv`	`text/csv` See “Example: Returning Results as CSV” on page 204. **Note**: Only preserves the order of the results, not the type.
	Note: `ASK` queries return a boolean (`true` or `false`).	
`CONSTRUCT` or `DESCRIBE` Returns RDF triples	`n-triples`	`application/n-triples` For faster serialization - see “Example: Returning Results as N-triples” on page 205. **Note**: If you want triples returned as JSON, the proper MIME type is `application/rdf+json`.
	other	`CONSTRUCT` or `DESCRIBE` queries return RDF triples in any of the available formats. See “Supported RDF Triple Formats” on page 38

Note: You can request any of the triple MIME types (`application/rdf+xml`, `text/turtle`, and so on), but use `application/n-triples` for best performance. See “Serialization” on page 196 for details.

The following examples use this SPARQL `SELECT` query to find US Congress bills that were sponsored by Robert Andrews ("A000210"):

```
#filename bills.sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX bill: <http://www.rdfabout.com/rdf/schema/usbill/>
PREFIX people: <http://www.rdfabout.com/rdf/usgov/congress/people/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT  ?bill ?title
WHERE { ?bill rdf:type bill:HouseBill ;
           dc:title ?title ;
           bill:sponsor people:A000210 .
    }
    LIMIT 5
```

The SPARQL query is saved as `bills.sparql`. The query limits responses to 5 results. Using `curl` and the REST Client API, you can query the SPARQL endpoint and get the results back in a variety of formats.

Note: If you use `curl` to make a `PUT` or `POST` request and read in the request body from a file, use `--data-binary` rather than `-d` to specify the input file. When you use `--data-binary`, `curl` inserts the data from the file into the request body as-is. When you use `-d`, `curl` strips newlines from the input, which can make your triple data or SPARQL syntactically invalid.

9.6.2 Example: Returning Results as XML

The SPARQL `SELECT` query in the `bills.sparql` file returns the response in XML format in this example.

```
curl --anyauth --user admin:password -i -X POST \
--data-binary @./bills.sparql \
-H "Content-type: application/sparql-query" \
-H "Accept: application/sparql-results+xml" \
http://localhost:8050/v1/graphs/sparql

=>

HTTP/1.1 200 OK
Content-type: application/sparql-results+xml
Server: MarkLogic
Content-Length: 1528
Connection: Keep-Alive
Keep-Alive: timeout=5

<sparql xmlns="http://www.w3.org/2005/sparql-results/">
 <head><variable name="bill"/>
       <variable name="title"/>
 </head>
 <results>
```

```
    <result>
      <binding name="bill">
       <uri>http://www.rdfabout.com/rdf/usgov/congress/108/bills/h1171
       </uri>
      </binding><binding name="title">
        <literal datatype="http://www.w3.org/2001/XMLSchema#string">
        H.R. 108/1171: Iris Scan Security Act of 2003
        </literal>
      </binding>
    </result>
    <result>
      <binding name="bill">
       <uri>http://www.rdfabout.com/rdf/usgov/congress/108/bills/h1314/
       </uri>
      </binding>
      <binding name="title">
       <literal datatype="http://www.w3.org/2001/XMLSchema#string">
       H.R. 108/1314: Screening Mammography Act of 2003</literal>
      </binding>
    </result>
    <result>
      <binding name="bill"><uri>http://www.rdfabout.com/rdf/usgov
      /congress/108/bills/h1384/</uri>
      </binding>
...
    </result>
  </results>
</sparql>
```

Note: In the example above, long lines have been broken into multiple lines using the UNIX line continuation character '\' and extra line breaks have been added for readability. Extra line breaks for readability have also been added in the results.

9.6.3 Example: Returning Results as JSON

The SPARQL `SELECT` query in the `bills.sparql` file returns the response in JSON format in this example:

```
curl --anyauth --user admin:password -i -X POST \
--data-binary @./bills.sparql \
-H "Content-type: application/sparql-query" \
-H "Accept: application/sparql-results+json" \
http://localhost:8050/v1/graphs/sparql

=>

HTTP/1.1 200 OK
Content-type: application/sparql-results+json
Server: MarkLogic
Content-Length: 1354
Connection: Keep-Alive
Keep-Alive: timeout=5
```

```
{"head":{"vars":["bill","title"]},
"results":{"bindings":[
   {"bill":{"type":"uri",
   "value":"http://www.rdfabout.com/rdf/usgov/congress/108/
            bills/h1171"},
   "title":{"type":"literal", "value":"H.R. 108/1171: Iris Scan
            Security Act of 2003",
   "datatype":"http://www.w3.org/2001/XMLSchema#string"}},
   {"bill":{"type":"uri",
   "value":"http://www.rdfabout.com/rdf/usgov/congress/108/
            bills/h1314"},
   "title":{"type":"literal", "value":"H.R. 108/1314: Screening
            Mammography Act of 2003",
   "datatype":"http://www.w3.org/2001/XMLSchema#string"}},
   {"bill":{"type":"uri",
   "value":"http://www.rdfabout.com/rdf/usgov/congress/108/
            bills/h1384"},
   "title":{"type":"literal", "value":"H.R. 108/1384: To amend the
            Railroad Retirement Act of 1974 to eliminate a limitation
            on benefits.",
   "datatype":"http://www.w3.org/2001/XMLSchema#string"}},
   {"bill":{"type":"uri",
   "value":"http://www.rdfabout.com/rdf/usgov/congress/108/
            bills/h1418"},
   "title":{"type":"literal", "value":"H.R. 108/1418: Veterans'
            Housing Equity Act",
   "datatype":"http://www.w3.org/2001/XMLSchema#string"}},
...
}]}}
```

Note: In the command line example above, long lines have been broken into multiple lines using the UNIX line continuation character '\' and extra line breaks have been added for readability. Extra line breaks for readability have also been added in the results.

9.6.4 Example: Returning Results as HTML

The same SPARQL `SELECT` query in the `bills.sparql` file returns the response in HTML format in this example:

```
curl --anyauth --user admin:password -i -X POST \
--data-binary @./bills.sparql \
-H "Content-type: application/sparql-query" \
-H "Accept: text/html" http://localhost:8050/v1/graphs/sparql"

=>

HTTP/1.1 200 OK
Content-type: text/html; charset=UTF-8
Server: MarkLogic
Content-Length: 1448
Connection: Keep-Alive
```

```
Keep-Alive: timeout=5

<html xmlns="http://www.w3.org/1999/xhtml">
 <head>
  <title>SPARQL results</title>
 </head>
 <body><table border="1">
 <tr>
  <th>bill</th>
  <th>title</th></tr>
 <tr>
  <td><a href="/v1/graphs/things?iri=http%3a//www.rdfabout.com/
    rdf/usgov/congress/108/bills/h1171">http://www.rdfabout.com/
    rdf/usgov/congress/108/bills/h1171</a>
  </td>
  <td>H.R. 108/1171: Iris Scan Security Act of 2003</td>
 </tr><tr>
  <td><a href="/v1/graphs/things?iri=http%3a//www.rdfabout.com/
    rdf/usgov/congress/108/bills/h1314">http://www.rdfabout.com/
    rdf/usgov/congress/108/bills/h1314</a>
  </td>
  <td>H.R. 108/1314: Screening Mammography Act of 2003</td>
 </tr><tr>
  <td><a href="/v1/graphs/things?iri=http%3a//www.rdfabout.com/
    rdf/usgov/congress/108/bills/h1384">http://www.rdfabout.com/
    rdf/usgov/congress/108/bills/h1384</a>
  </td>
  <td>H.R. 108/1384: To amend the Railroad Retirement Act of
    1974 to eliminate a limitation on benefits.
  </td>
 </tr>
...
 </table>
 </body>
</html>
```

Note: In the preceding example, long lines have been broken into multiple lines using the UNIX line continuation character '\' and extra line breaks have been added for readability. Extra line breaks for readability have also been added in the results.

9.6.5 Example: Returning Results as CSV

Here is the same SPARQL `SELECT` query (`bills.sparql`) with the results returned in CSV format:

```
curl --anyauth --user Admin:janem-3 -i -X POST --data-binary \
@./bills.sparql -H "Content-type: application/sparql-query" \
-H "Accept: text/csv" http://janem-3:8000/v1/graphs/sparql
=>
bill,title

http://www.rdfabout.com/rdf/usgov/congress/108/bills/h1171,H.R.
108/1171: Iris Scan Security Act of 2003
```

```
http://www.rdfabout.com/rdf/usgov/congress/108/bills/h1314,H.R.
108/1314: Screening Mammography Act of 2003

http://www.rdfabout.com/rdf/usgov/congress/108/bills/h1384,H.R.
108/1384: To amend the Railroad Retirement Act of 1974 to eliminate a
limitation on benefits.

http://www.rdfabout.com/rdf/usgov/congress/108/bills/h1418,H.R.
108/1418: Veterans' Housing Equity Act

http://www.rdfabout.com/rdf/usgov/congress/108/bills/h1419,H.R.
108/1419: Seniors' Family Business Enhancement Act[jmckean@janem-3 ~]$
```

Note: In the preceding example, long lines have been broken into multiple lines using the UNIX line continuation character '\' and extra line breaks have been added for readability. Extra line breaks for readability have also been added in the results.

9.6.6 Example: Returning Results as N-triples

For this example, we will use a DESCRIBE query that was introduced and used earlier:

```
DESCRIBE <http://dbpedia.org/resource/Pascal_Bedrossian>
```

The following command uses this query to match triples that describe Pascal and return the results as N-Triples. Long lines in the command below have been broken with the UNIX line continuation character "\". The query is URL encoded and passed as the value of the "query" request parameter.

```
curl -X GET --digest --user "admin:password" \
-H "Accept: application/n-triples" \
-H "Content-type: application/x-www-form-urlencoded"\
"http://localhost:8321/v1/graphs/sparql?query=DESCRIBE%20%3Chttp\
%3A%2F%2Fdbpedia.org%2Fresource%2FPascal_Bedrossian%3E"
=>
<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/France> .

<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/resource/Marseille> .

<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://xmlns.com/foaf/0.1/Person> .

<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://xmlns.com/foaf/0.1/surname> "Bedrossian"@en .

<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://xmlns.com/foaf/0.1/givenName> "Pascal"@en .
```

```
<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://xmlns.com/foaf/0.1/name> "Pascal Bedrossian"@en .

<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://purl.org/dc/elements/1.1/description> "footballer"@en .

<http://dbpedia.org/resource/Pascal_Bedrossian>
<http://dbpedia.org/ontology/birthDate> "1974-11-
28"^^<http://www.w3.org/2001/XMLSchema#date> .
```

9.6.7 Example: Returning a Boolean as XML or JSON

In this example, a SPARQL `ASK` query (from an earlier example) is used to determine whether Carolyn Kennedy was born after Eunice Kennedy.

Here are the contents of the `ask-sparql.sparql` file used in the following query:

```
#file: ask-sparql.sparql
PREFIX db: <http://dbpedia.org/resource/>
PREFIX onto: <http://dbpedia.org/ontology/>
ASK
{
    db:Carolyn_Bessette-Kennedy onto:birthDate ?by .
    db:Eunice_Kennedy_Shriver onto:birthDate ?bd .
    FILTER (?by>?bd).
}
```

Note: If you use `curl` to make a `PUT` or `POST` request and read in the request body from a file, use `--data-binary` rather than `-d` to specify the input file. When you use `--data-binary`, `curl` inserts the data from the file into the request body as-is. When you use `-d`, `curl` strips newlines from the input, which can make your triple data or SPARQL syntactically invalid.

This request, containing SPARQL `ASK` query, returns the boolean result as XML:

```
curl --anyauth --user user:password -i -X POST \
--data-binary @./ask-sparql.sparql \
-H "Content-type: application/sparql-query" \
-H "Accept: application/sparql-results+xml" \
http://localhost:8050/v1/graphs/sparql

=>

HTTP/1.1 200 OK
Content-type: application/sparql-results+xml
Server: MarkLogic
Content-Length: 89
Connection: Keep-Alive
Keep-Alive: timeout=5

<sparql
```

```
<xmlns="http://www.w3.org/2005/sparql-results/">
<boolean>true</boolean>
</sparql>
```

Here is the same request (containing the SPARQL `ASK` query) where the boolean result is returned as JSON:

```
curl --anyauth --user user:password -i -X POST \
--data-binary @./ask-sparql.sparql \
-H "Content-type: application/sparql-query" \
-H "Accept: application/sparql-results+json" \
http://localhost:8050/v1/graphs/sparql

=>

HTTP/1.1 200 OK
Content-type: application/sparql-results+json
Server: MarkLogic
Content-Length: 17
Connection: Keep-Alive
Keep-Alive: timeout=5

{"boolean":true}
```

9.7 SPARQL Query with the REST Client API

SPARQL queries (`SELECT`, `DESCRIBE`, `CONSTRUCT`, and `ASK`) can be used with either `POST` or `GET` and the REST Client API. For more about query types and output, see the table in "SPARQL Query Types and Output Formats" on page 200.

This section includes the following:

- SPARQL Queries in a POST Request
- SPARQL Queries in a GET Request

9.7.1 SPARQL Queries in a POST Request

This section describes how SPARQL query can be used to manage graphs and triple data through `/v1/graphs/sparql` endpoint.

```
http://hostname:port/v1/graphs/sparql
```

where the *hostname* and *port* are the host and port on which you are running MarkLogic.

You can specify your input SPARQL query to `POST:/v1/graphs/sparql` in the following ways:

- Include a SPARQL query as a file in the `POST` body
- Include the SPARQL query as URL-encoded data

This is a SPARQL DESCRIBE query used to find out about US Congress bill 44. The query is saved as a file, named bill44.sparql.

```
#file name bill44.sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX bill: <http://www.rdfabout.com/rdf/schema/usbill/>
DESCRIBE ?x WHERE { ?x rdf:type bill:HouseBill ;
                       bill:number "44" . }
```

Note: If you use curl to make a PUT or POST request and read in the request body from a file, use --data-binary rather than -d to specify the input file. When you use --data-binary, curl inserts the data from the file into the request body as-is. When you use -d, curl strips newlines from the input, which can make your triple data or SPARQL syntactically invalid.

The endpoint requires a SPARQL query to be either a parameter or in the POST body. In the following example, the bill44.sparql file with the DESCRIBE query is passed to the body of the POST request:

```
# Windows users, see Modifying the Example Commands for Windows
curl --anyauth --user admin:password \
-i -X POST --data-binary @./bill44.sparql \
-H "Content-type: application/sparql-query" \
-H "Accept: application/rdf+xml" \
http://localhost:8000/v1/graphs/sparql
```

The request body MIME type is specified as application/sparql-query and the requested response MIME type (the Accept:) is specified as application/rdf+xml. The output is returned as triples in XML format. See “Response Output Formats” on page 199 for more details.

The query returns the following triples describing bill 44:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns">
 <OfficialTitle rdf:ID="bnodebnode3097714188l9f878"
    xmlns="http://www.rdfabout.com/rdf/schema/usbill/">
  <rdf:typerdf:resource="http://www.rdfabout.com/rdf/schema/usbill
    /OfficialTitle"/>
  <rdf:value rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    To amend the Internal Revenue Code of 1986 to provide reduced
    capital gain rates for qualified economic stimulus gain and to
    index the basis of assets of individuals for purposes of
    determining gains and losses.</rdf:value></OfficialTitle>
 <ShortTitle rdf:ID="bnodebnode30b47143b819db78"
   xmlns="http://www.rdfabout.com/rdf/schema/usbill/">
  <rdf:type rdf:resource="http://www.rdfabout.com/rdf/schema/usbill
   /ShortTitle"/>
  <rdf:value rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
   Investment Tax Incentive Act of 2003</rdf:value></ShortTitle>
 <ShortTitle rdf:ID="bnodebnodee1860b72fb58b315"
   xmlns="http://www.rdfabout.com/rdf/schema/usbill/">
```

```
 <rdf:type rdf:resource="http://www.rdfabout.com/rdf/schema/usbill
  /ShortTitle"/>
 <rdf:value rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
  Investment Tax Incentive Act of 2003</rdf:value></ShortTitle>
<HouseBill rdf:about="http://www.rdfabout.com/rdf/usgov/congress/108
  /bills/h44" xmlns="http://www.rdfabout.com/rdf/schema/usbill/">
<inCommittee rdf:resource="http://www.rdfabout.com/rdf/usgov
  /congress/committees/HouseWaysandMeans"/>
<cosponsor rdf:resource="http://www.rdfabout.com/rdf/usgov/congress
  /people/A000358"/>
<cosponsor rdf:resource="http://www.rdfabout.com/rdf/usgov/congress
  /people/B000208"/>
<cosponsor rdf:resource="http://www.rdfabout.com/rdf/usgov/congress
  /people/B000575"/>
<cosponsor
.....
```

Another way to use the POST request is to specify the URL-encoded query as the value of the "query" parameter and use `application/x-www-form-urlencoded` as the request body MIME type, as described in the Semantics documentation of the *REST Client API*.

The following SPARQL SELECT query finds the House of Congress bills that were cosponsored by the person with a Congress BioGuideID of "A000358":

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX bill: <http://www.rdfabout.com/rdf/schema/usbill/>
PREFIX people: <http://www.rdfabout.com/rdf/usgov/congress/people/>
SELECT ?x WHERE { ?x rdf:type bill:HouseBill ; bill:cosponsor
people:A000358. }
```

In this example, the SELECT query is URL-encoded and then sent as form-encoded data:

```
curl -X POST --anyauth --user admin:password \
-H "Accept:application/sparql-results+xml" --data-binary \
"query=PREFIX%20rdf%3A%20%3Chttp%3A%2F%2Fwww.w3.org%2F1999%2F02%2F22-r
df-syntax-ns\
%23%3E%20%0APREFIX%20bill%3A%20%3Chttp%3A%2F%2Fwww.rdfabout.com%2Frdf\
%2Fschema%2Fusbill%2F%3E%0APREFIX%20people%3A%20%3Chttp%3A%2F%2Fwww.rd
fabout.com\
%2Frdf%2Fusgov%2Fcongress%2Fpeople%2F%3E%0ASELECT%20%3Fx%20WHERE%20%7B
%20%3Fx\
%20rdf%3Atype%20bill%3AHouseBill%20%3B%20bill%3Acosponsor%20%20people%
3AA000358.%20%7D%0A"\
-H "Content-type:application/x-www-form-urlencoded" \
http://localhost:8000/v1/graphs/sparql
=>

<sparql xmlns="http://www.w3.org/2005/sparql-results/">
   <head><variable name="x"/></head>
  <results>
    <result><binding name="x">
     <uri>http://www.rdfabout.com/rdf/usgov/congress/108/bills/h1036
```

```
      </uri></binding></result>
     <result><binding name="x">
      <uri>http://www.rdfabout.com/rdf/usgov/congress/108/bills/h1057
      </uri></binding></result>
     <result><binding name="x">
      <uri>http://www.rdfabout.com/rdf/usgov/congress/108/bills/h1078
      </uri></binding></result>
     <result><binding name="x">
      <uri>http://www.rdfabout.com/rdf/usgov/congress/108/bills/h110
      </uri></binding></result>
     <result><binding name="x">
      <uri>http://www.rdfabout.com/rdf/usgov/congress/108/bills/h1117
      </uri></binding></result>
     <result><binding name="x">
      <uri>http://www.rdfabout.com/rdf/usgov/congress/108/bills/h1153
      </uri></binding></result>
.......
     <result><binding name="x">
      <uri>http://www.rdfabout.com/rdf/usgov/congress/108/bills/h975
      </uri></binding></result>
  </results>
</sparql>
```

Note: For readability, the long command line is broken into multiple lines using the UNIX line continuation character '\'. Extra line breaks have been inserted for readability of the URL-encoded query.

9.7.2 SPARQL Queries in a GET Request

For a `GET` request, the SPARQL query in the query request parameter must be URL-encoded. Here is the SPARQL `DESCRIBE` query, searching for a US Congress bill (44), before it is encoded:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bill: <http://www.rdfabout.com/rdf/schema/usbill/>
DESCRIBE ?x
WHERE {
  ?x rdf:type bill:HouseBill ;
    bill:number "44" . }
```

In this example `curl` sends an HTTP `GET` request to execute the SPARQL `DESCRIBE` query :

```
curl -X GET --digest --user "user:password" \
-H "accept: application/sparql-results+xml" \
"http://localhost:8000/v1/graphs/sparql?query=PREFIX%20rdf%3A%20%3C\
http%3A%2F%2Fwww.w3.org%2F1999%2F02%2F22-rdf-syntax-ns%23%3E%20%0A\
PREFIX%20bill%3A%20%3Chttp%3A%2F%2Fwww.rdfabout.com%2Frdf%2Fschema\
%2Fusbill%2F%3E%0ADESCRIBE%20%3Fx%20WHERE%20%7B%20%3Fx%20rdf%3Atype\
%20bill%3AHouseBill%20%3B%20%20bill%3Anumber%20%2244%22%20.%20%7D"
```

Your results will be similar to these triples:

```
<http://www.rdfabout.com/rdf/usgov/congress/108/bills/h44>
<http://purl.org/dc/elements/1.1/title> "H.R. 108/44: Investment Tax
Incentive Act of 2003" .

<http://www.rdfabout.com/rdf/usgov/congress/108/bills/h44>
<http://purl.org/dc/terms/created> "2003-01-07" .

<http://www.rdfabout.com/rdf/usgov/congress/108/bills/h44>
<http://purl.org/ontology/bibo/shortTitle> "H.R. 44: Investment Tax
Incentive Act of 2003" .

<http://www.rdfabout.com/rdf/usgov/congress/108/bills/h44>
<http://www.rdfabout.com/rdf/schema/usbill/congress> "108" .

<http://www.rdfabout.com/rdf/usgov/congress/108/bills/h44>
<http://www.rdfabout.com/rdf/schema/usbill/cosponsor>
<http://www.rdfabout.com/rdf/usgov/congress/people/A000358> .

<http://www.rdfabout.com/rdf/usgov/congress/108/bills/h44>
<http://www.rdfabout.com/rdf/schema/usbill/cosponsor>
<http://www.rdfabout.com/rdf/usgov/congress/people/B000208> .

<http://www.rdfabout.com/rdf/usgov/congress/108/bills/h44>
<http://www.rdfabout.com/rdf/schema/usbill/cosponsor>
<http://www.rdfabout.com/rdf/usgov/congress/people/B000575> .
```

The triples describe information about bill 44 in the U.S. Congress; it's title, when it was created, who cosponsored the bill, and so on.

9.8 SPARQL Update with the REST Client API

This section describes how SPARQL Update can be used to manage graphs and triple data through `/v1/graphs/sparql` endpoint.

```
http://hostname:port/v1/graphs/sparql
```

where the *hostname* and *port* are the host and port on which you are running MarkLogic.

You can specify your SPARQL Update (which is a `DELETE`/`INSERT`) to `POST:/v1/graphs/sparql` in the following ways:

- Include a SPARQL Update as a file in the `POST` body in the form of:

  ```
  http://host:port/v1/graphs/sparql
  content-type:application/sparql-update
  ```

  See "SPARQL Update in a POST Request" on page 212.

- Include the SPARQL Update as URL-encoded data in the form of:

```
http://host:port/v1/graphs/sparql
content-type:application/x-www-form-urlencoded
```

See “SPARQL Update via POST with URL-encoded Parameters” on page 214.

The examples in this section use a `POST` request, with no URL encoding, and with `content-type:application/sparql-update`.

You can specify the RDF dataset against which to execute the update using the `using-graph-uri` and `using-named-graph-uri` request parameters, or within the update. If the dataset is specified in both the request parameters and the update, the dataset defined by the request parameters is used. If neither is specified, all graphs (the `UNION` of all graphs) are included in the operation.

> **Note:** Including the `using-graph-uri` or `using-named-graph-uri` parameters, with a SPARQL 1.1 Update request that contains an operation that uses the `USING`, `USING NAMED`, or `WITH` clause, will cause an error.

See “Specifying Parameters” on page 191 for details on specifying parameters for use with the REST Client API. See “Supported Operations for the REST Client API” on page 194 and the list of verbs supported by the Graph store endpoint for SPARQL Update for more about `POST`. See `POST:/v1/graphs/sparql` for more about the SPARQL endpoint.

This section includes the following:

- SPARQL Update in a POST Request
- SPARQL Update via POST with URL-encoded Parameters

9.8.1 SPARQL Update in a POST Request

You can send requests using the `POST` method by including SPARQL Update in the request body. Set the Content-type HTTP header to `application/sparql-update`.

Here is the SPARQL Update before it is added to the request body:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
WITH <BOOKS>
DELETE {?b dc:title "A new book"}
INSERT
{?b dc:title "Inside MarkLogic Server" }
WHERE {?b dc:title "A new book".}
```

In the graph named `<BOOKS>`, SPARQL Update matches a triple with `dc:title` in the predicate position and `A new book` in the object position and deletes it. Then a new triple is inserted (`?b dc:title "MarkLogic Server"`).

In this example, the SPARQL Update is sent in the request body using `application/sparql-update` and the `-d` option for the query:

```
# Windows users, see Modifying the Example Commands for Windows

curl --anyauth --user admin:admin -i -X POST \
-H "Content-type:application/sparql-update" \
-H "Accept:application/sparql-results+xml" \
-d 'PREFIX dc: <http://purl.org/dc/elements/1.1/> \
WITH <BOOKS> \
DELETE {?b dc:title "A new book"} \
INSERT {?b dc:title "Inside MarkLogic Server" } \
WHERE {?b dc:title "A new book".}' \
http://localhost:8000/v1/graphs/sparql
```

Note: For clarity, long command lines are broken into multiple lines using the line continuation characters "\" . Remove the line continuation characters when you use the `curl` command. (For Windows the line continution character is "^".)

Alternatively, you can use `curl` to execute a SPARQL Update from a file as part of a `POST` request. The SPARQL Update is saved in a file named `booktitle.sparql`. Here are the file contents:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA
{
<http://example/book1> dc:title "book title" ;
dc:creator "author name" .
}
```

The `POST` request with the SPARQL Update in a file would look like this:

```
curl --anyauth --user admin:admin -i -X POST \
--data-binary @./booktitle.sparql \
-H "Content-type:application/sparql-update" \
-H "Accept:application/sparql-results+xml" \
http://localhost:8000/v1/graphs/sparql
```

Notice that the request uses the `--data-binary` option instead of `-d` to call the file containing the SPARQL Update. You can include `using-graph-uri`, `using-named-graph-uri` and `role-capability` as HTTP request parameters. The `perm` parameter is expected in this syntax, with role and capability.

```
perm:admin=update&perm:admin=execute
```

See "Default Permissions and Required Privileges" on page 218 for more about permissions.

9.8.2 SPARQL Update via POST with URL-encoded Parameters

You can also send update protocol requests via the HTTP `POST` method by URL-encoding the parameters. When you do this, URL percent-encode all parameters and include them as parameters within the request body via the `application/x-www-form-urlencoded` media type. The content type header of the HTTP request is set to `application/x-www-form-urlencoded`.

This next example uses SPARQL Update and `POST` with URL-encoded parameters to insert data (along with a set of permissions) into graph `C1`.

```
curl --anyauth --user admin:admin -i -X POST \
--data-urlencode update='PREFIX dc: <http://purl.org/dc/elements/1.1/>
\
INSERT DATA \
{<http://example/book1> dc:title "book title" ; \
dc:creator "author name" .}' \
-H "Content-type:application/x-www-form-urlencoded" \
-H "Accept:application/sparql-results+xml" \
'http://localhost:8000/v1/graphs/sparql?using-named-graph-uri=C1 \
&perm:admin=update&perm:admin+execute'
```

If you supply the `using-graph-uri` or `using-named-graph-uri` parameters when using this protocol to convey a SPARQL 1.1 Update request that uses the `USING NAMED`, or `WITH` clause, the operation will result in an error.

This `curl` example uses `POST` with URL-encoding for the SPARQL Update and permissions:

```
curl --anyauth --user admin:admin -i -X POST \
-H "Content-type:application/x-www-form-urlencoded" \
-H "Accept:application/sparql-results+xml" \
--data-urlencode update='PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA{ GRAPH <C1> {http://example/book1/> dc:title "C book"} }' \
--data-urlencode perm:rest-writer=read \
--data-urlencode perm:rest-writer=update \
http://localhost:8321/v1/graphs/sparql
```

If a new RDF graph is created, the server responds with a `201 Created` message. The response to an update request indicates success or failure of the request via HTTP response status code (`200` or `400`). If the request body is empty, the server responds with `204 No Content`.

9.9 Listing Graph Names with the REST Client API

You can list the graphs in your database with the REST Client API using the `things` endpoint.

```
http://hostname:port/v1/things
```

where the *hostname* and *port* are the host and port on which you are running MarkLogic.

For example when this endpoint is called with no parameters, a list of graphs in the database is returned:

```
http://localhost:8000/v1/things
```

The request might return graphs like these:

```
graphs/MyDemoGraph
http://marklogic.com/semantics#default-graph
http://marklogic.com/semantics#graphs
```

9.10 Exploring Triples with the REST Client API

The following endpoint provides RESTful access to knowledge (things) referred to in the database. This endpoint retrieves a list of all subject nodes in the database:

```
http://hostname:port/v1/graphs/things
```

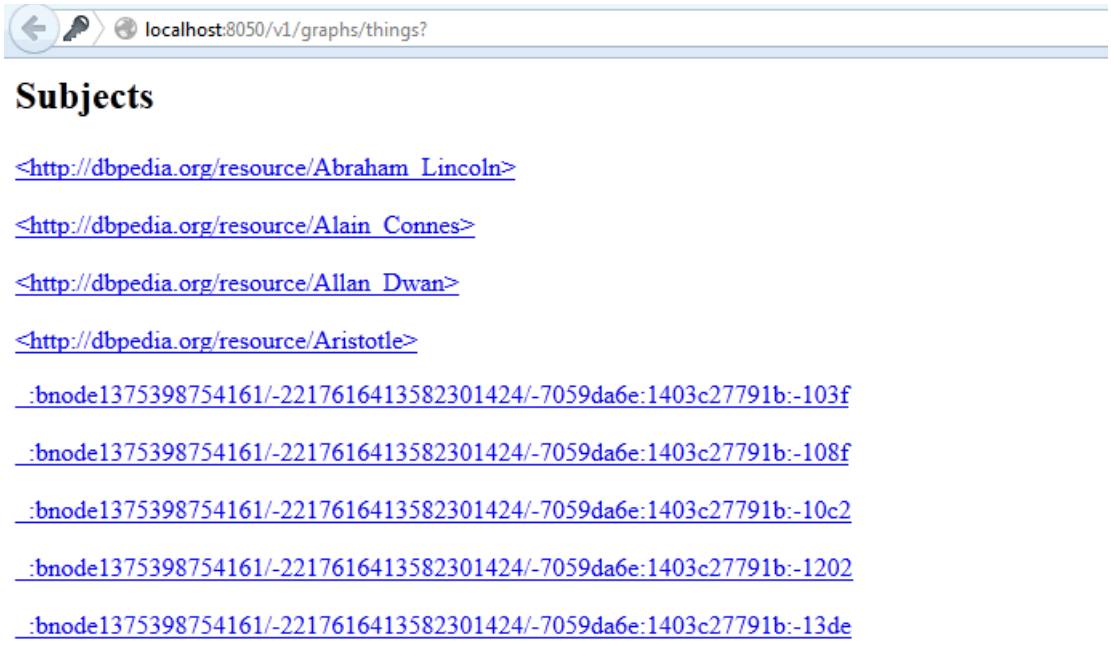
where the *hostname* and *port* are the host and port on which you are running MarkLogic.

For example:

```
http://localhost:8050/v1/graphs/things
```

You can also specify a set of subject nodes to be returned. When this endpoint is called with no parameters, a list of subject nodes in the database is returned for all triples in the database.

This example shows the response, a list of nodes as IRIs, in a Web browser:



Note: This endpoint has a hard-coded limit of 10,000 items to display, and does not support pagination.

You can traverse and navigate the triples in the database by clicking on the links and drilling down the nodes. Clicking on an IRI may return one or more related triples:

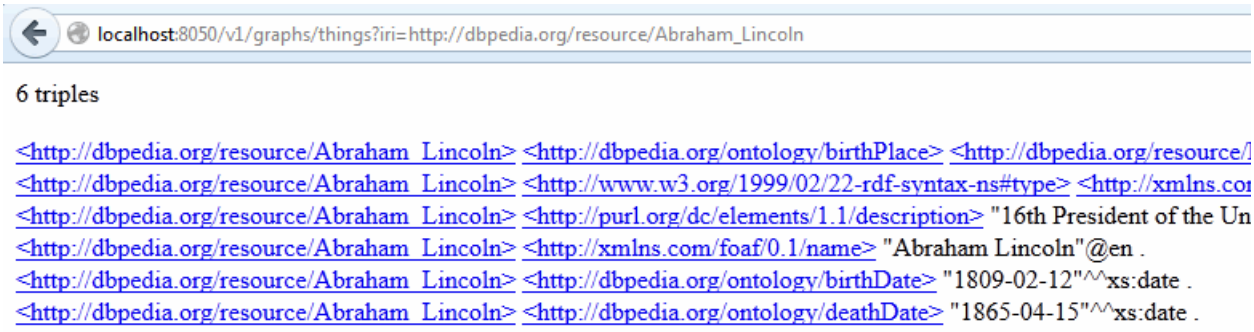


You can use an optional `iri` parameter to specify particular IRIs about which to return information, in Turtle triple serialization.

For example, you could paste this request into your browser:

```
http://localhost:8050/v1/graphs/things?iri=http://dbpedia.org/resource
/Abraham_Lincoln
```

The nodes selected by the IRI `http://dbpedia.org/resources/Abraham_Lincoln` are returned in Turtle serialization:



If you are using `curl` or an equivalent command-line tool for issuing HTTP requests, you can specify the following MIME types in the request Accept header:

- When no parameters are specified, use `text/html` in the request Accept header.
- When you use the `iri` parameter, use one of the MIME types listed in “SPARQL Query Types and Output Formats” on page 200. See “Supported RDF Triple Formats” on page 38 for additional information about RDF triple formats.

In this example, the `GET` request returns the nodes selected by the given `iri` parameter in Turtle triple serialization:

```
curl --anyauth --user admin:password -i -X GET \
-H "Accept: text/turtle" \
http://localhost:8051/v1/graphs/things?iri=http://dbpedia.org/resource
/Aristotle

=>

HTTP/1.1 200 OK
Content-type: text/turtle; charset=UTF-8
Server: MarkLogic
Content-Length: 628
Connection: Keep-Alive
Keep-Alive: timeout=5

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
<http://dbpedia.org/resource/Aristotle>
<http://dbpedia.org/ontology/deathPlace>
 <http://dbpedia.org/resource/Chalcis> .
<http://dbpedia.org/resource/Aristotle>
<http://dbpedia.org/ontology/birthPlace>
 <http://dbpedia.org/resource/Stagira_(ancient_city)> .
<http://dbpedia.org/resource/Aristotle> <http://www.w3.org/1999/02/22-
rdf-syntax
-ns#type/> <http://xmlns.com/foaf/0.1/Person> .
<http://dbpedia.org/resource/Aristotle>
<http://xmlns.com/foaf/0.1/name>
"Aristotle" .
<http://dbpedia.org/resource/Aristotle>
<http://purl.org/dc/elements/1.1/description>
"Greek philosopher" .
```

Note: If a given IRI does not exist, the response is “404 Not Found”. A `GET` request for a response in an unsupported serialization will yield “406 Not Acceptable”.

9.11 Managing Graph Permissions

This section covers the REST Client API support for setting, modifying, and retrieving graph permissions. If you are not already familiar with the MarkLogic security model, review the *Security Guide*.

The following topics are covered:

- Default Permissions and Required Privileges
- Setting Permissions as Part of Another Operation
- Setting Permissions Standalone
- Retrieving Graph Permissions

9.11.1 Default Permissions and Required Privileges

All graphs created and managed using the REST Client API grant "read" capability to the `rest-reader` role and "update" capability to the `rest-writer` role. These default permissions are always assigned to a graph, even if you do not explicitly specify them.

For example, if you create a new graph using `PUT /v1/graphs` and do not specify any permissions, the graph will have permissions similar to the following:

XML	JSON

`<metadata xmlns="http://marklogic.com/rest-api">`
`  <permissions>`
`  <permission>`
`  <role-name>rest-writer</role-name>`
`  <capability>update</capability>`
`  </permission>`
`  <permission>`
`  <role-name>rest-reader</role-name>`
`  <capability>read</capability>`
`  </permission>`
`  </permissions>`
`</metadata>`
	`{"permissions":[`
	`  {"role-name":"rest-writer",`
	`  "capabilities":["update"]`
	`  },`
	`  {"role-name":"rest-reader",`
	`  "capabilities":["read"]`
	`  }`
	`]}`

If you explicitly specify other permissions when creating the graph, the above default permissions are still set, as well as the permissions you specify.

You can use custom roles to limit access to selected users on a graph by graph basis. Your custom roles must include equivalent `rest-reader` and `rest-writer` privileges. Otherwise, users with these roles cannot use the REST Client API to manage or query semantic data.

For details, see Security Requirements in the *REST Application Developer's Guide*.

9.11.2 Setting Permissions as Part of Another Operation

Use the `perm` request parameter to set, overwrite, or add permissions as part of another graph operation. To manage permissions when not modifying graph content, use the `category` parameter instead. For details, see "Setting Permissions Standalone" on page 219.

The `perm` parameter has the following form:

```
perm:role=capability
```

Where *role* is the name of a role defined in MarkLogic and *capability* is one of "read", "insert", "update", or "execute".

You can specify the `perm` parameter multiple times to grant multiple capabilities to the same role or set permissions for multiple roles. For example, the following set of parameters grants the "readers" role the "read" capability and the "writers" role the "update" capability:

```
perm:readers=read&perm:writers=update
```

Note: Setting or changing the permissions on a graph does not effect the permissions on documents that contain embedded triples in that graph.

You can use the `perm` parameter with the following operations:

Operation	REST Client API Methods
Set or overwrite permissions on a named graph or the default graph while creating or overwritng the graph contents.	`PUT /v1/graphs?graph=uri&perm:`*`role=capability`* `PUT /v1/graphs?default&perm:`*`role=capability`* Request body contains new graph contents (triples).
During a SPARQL Update operation, set permissions on all graphs created as part of the update.	`POST /v1/graphs/sparql?perm:`*`role=capability`* Request body contains SPARQL Update.
Add permissions to a named graph while adding content to the graph.	`POST /v1/graphs?graph=uri&perm:`*`role=capability`* `POST /v1/graphs?default&perm:`*`role=capability`* Request body contains graph updates (triples)

The following restrictions apply:

- When you use the `perm` parameter with `/v1/graphs`, you must also include either the `graph` or the `default` request parameter.
- You cannot use the `perm` parameter in conjunction with `category=permissions` or `category=metadata`.
- When you use the `perm` parameter to specify permissions as part of a SPARQL Update operation, the permissions only affect graphs created as part of the update. The permissions on pre-existing graphs remain unchanged.

9.11.3 Setting Permissions Standalone

Set the `category` request parameter to `permissions` to manage permissions without affecting the contents of a graph. For example, a request of the following form that includes permissions metadata in the request body sets the permissions on the default graph, but does not change the graph contents.

```
PUT /v1/graphs?default&category=permissions
```

To set or add permissions along with your graph content, use the `perm` request parameter. For details, see "Setting Permissions as Part of Another Operation" on page 218.

You can set the `category` parameter to either `permissions` or `metadata`. They are equivalent in the context of graph management.

The request body must contain permissions metadata. In XML, the metadata can be rooted at either a `metadata` element or the `permissions` element. Also, in XML, the metadata must be in the namespace `http://marklogic.com/rest-api`.

For example, all of the following are acceptable:

XML	JSON

<metadata xmlns="http://marklogic.com/rest-api">
  <permissions>
  <permission>
  <role-name>roleA</role-name>
  <capability>read</capability>
  <capability>update</capability>
  </permission>
  <permission>
  <role-name>roleB</role-name>
  <capability>read</capability>
  </permission>
  </permissions>
</metadata>
	{"permissions":[
	  {"role-name":"roleA",
	  "capabilities":[
	  "read", "update"
	  ]
	  },
	  {"role-name":"roleB",
	  "capabilities":["read"]
	  }
	]}

<permissions xmlns="http://marklogic.com/rest-api"> <permission> <role-name>roleA</role-name> <capability>read</capability> <capability>update</capability> </permission> <permission> <role-name>roleB</role-name> <capability>read</capability> </permission> </permissions>	

Note: Setting or changing the permissions on a graph does not effect the permissions on documents that contain embedded triples in that graph.

You can use the `category=permissions` pattern to manage graph permissions with the following methods. In all cases, the graph contents are unaffected.

Operation	REST Client API Pattern

Set or overwrite permissions on a named graph or the default graph.	`PUT /v1/graphs?graph=uri&category=permissions`
	`PUT /v1/graphs?default&category=permissions`
	Request body contains permissions metadata. You can also use `category=metadata`.

Add permissions to a named graph or the default graph.	`POST /v1/graphs?graph=uri&category=permissions`
	`POST /v1/graphs?default&category=permissions`
	Request body contains permissions metadata. You can also use `category=metadata`.

Reset the permissions to default permissions on a named graph or the default graph.	`DELETE /v1/graphs?graph=uri&category=permissions` `DELETE /v1/graphs?default&category=permissions`

The following restrictions apply:

- When you use `category=permissions` or `category=metadata` with `/v1/graphs`, you must also include either the `graph` or the `default` request parameter.
- You cannot use `category=permissions` or `category=metadata` in conjunction with the `perm` parameter.

9.11.4 Retrieving Graph Permissions

To retrieve permissions metadata about a named graph, make a GET request of the following form:

```
GET /v1/graphs?graph=graphURI&category=permissions
```

To retrieve permissions metadata about the default graph, make a GET request of the following form:

```
GET /v1/graphs?default&category=permissions
```

You can request metadata in either XML or JSON. The default format is XML.

For example, the following command retrieves permissions for the graph with URI `/my/graph`, as XML. In this case, the graph includes both the default `rest-writer` and `read-reader` permissions and permissions for a custom role named “GroupA”.

```
curl --anyauth --user user:password -X GET -i \
  -H "Accept: application/xml" \
  'http://localhost:8000/v1/graphs?graph=/my/graph&category=permissions'

HTTP/1.1 200 OK
Content-type: application/xml; charset=utf-8
Server: MarkLogic
Content-Length: 868
Connection: Keep-Alive
Keep-Alive: timeout=5

<rapi:metadata uri="/my/graph" ...
     xmlns:rapi="http://marklogic.com/rest-api" ...>
  <rapi:permissions>
    <rapi:permission>
      <rapi:role-name>rest-writer</rapi:role-name>
      <rapi:capability>update</rapi:capability>
    </rapi:permission>
    <rapi:permission>
      <rapi:role-name>rest-reader</rapi:role-name>
      <rapi:capability>read</rapi:capability>
    </rapi:permission>
    <rapi:permission>
      <rapi:role-name>GroupA</rapi:role-name>
      <rapi:capability>read</rapi:capability>
      <rapi:capability>update</rapi:capability>
    </rapi:permission>
  </rapi:permissions>
</rapi:metadata>
```

The following data is the equivalent permissions metadata, expressed as JSON:

```
{"permissions":[
  {"role-name":"rest-writer","capabilities":["update"]},
  {"role-name":"rest-reader","capabilities":["read"]},
  {"role-name":"GroupA","capabilities":["read","update"]}
]}
```

10.0 XQuery and JavaScript Semantics APIs

This chapter describes the XQuery and JavaScript Semantics APIs, which include an XQuery library module, built-in semantics functions, and support for SPARQL, SPARQL Update, and RDF. The Semantics API is designed for large-scale, production triple stores and applications. The complete list of semantic functions can be found at http://docs.marklogic.com/sem/semantics.

This chapter includes examples of using the Semantics API, which is an API designed to create, query, update, and delete triples and graphs in MarkLogic.

Additionally, the following APIs support the MarkLogic Semantics features; XQuery API, REST API, Node.js Client API, and Java Client API, using a variety of query styles, as described in the Loading Semantic Triples, Semantic Queries and Inserting, Deleting, and Modifying Triples with XQuery and Server-Side JavaScript chapters of this guide.

This chapter includes the following sections:

- XQuery Library Module for Semantics
- Generating Triples
- Extracting Triples from Content
- Parsing Triples
- Exploring Data

Note: Semantics is a separately licensed product: you need a valid semantics license key to use semantics.

10.1 XQuery Library Module for Semantics

Some of the Semantics XQuery functions are built-in functions that do not require an import statement, while others are implemented in an XQuery library module that requires an import statement. To simplify things, MarkLogic recommends that you import the Semantics API library into every XQuery module or JavaScript module that uses the Semantics API.

10.1.1 Importing the Semantics Library Module with XQuery

You can use the Semantics API library module with XQuery by importing the module into your XQuery with the following prolog statement:

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

The prefix for all semantic functions in XQuery is `http://marklogic.com/semantics`. The Semantics API uses the prefixes `sem:` or `rdf:`, which are defined in the server. For details about the function signatures and descriptions, see the Semantics documentation under XQuery Library Modules in the *XQuery and XSLT Reference Guide* and the *MarkLogic XQuery and XSLT Function Reference*.

10.1.2 Importing the Semantics Library Module with JavaScript

For JavaScript you can use the Semantics API library module by importing the module into your JavaScript with this statement:

```
var sem = require("/marklogic/semantics.xqy");
```

The prefix for all semantic XQuery functions in JavaScript is `/marklogic.com/semantics.xqy`. With JavaScript, the Semantics API uses the prefix `sem.` , which is defined in the server. For details about the function signatures and descriptions, see the Semantics documentation under JavaScript Library Modules in the *JavaScript Reference Guide* and the *MarkLogic XQuery and XSLT Function Reference*.

10.2 Generating Triples

The XQuery `sem:rdf-builder` function is a powerful tool for dynamically generating triples in the Semantics API. (For JavaScript, the function is `sem.rdfBuilder`.)

The function builds triples from the CURIE and blank node syntaxes. Blank nodes specified with a leading underscore (_) are assigned blank node identifiers, and maintain state across multiple invocations; for example, "`_:person1`" refers to the same node as a later invocation that also mentions "`_:person1`". For example:

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

 let $builder := sem:rdf-builder((), sem:iri("my-named-graph"))
 let $t1 := $builder("_:person1", "a", "foaf:Person")
 let $t2 := $builder("_:person2", "a", "foaf:Person")
 let $t3 := $builder("_:person1", "foaf:knows", "_:person2")
 return ($t1,$t2,$t3)

=>

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .

<http://marklogic.com/semantics/blank/4892021155019117627>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type/>
<http://xmlns.com/foaf/0.1/Person> .

<http://marklogic.com/semantics/blank/6695700652332466909>
        <http://xmlns.com/foaf/0.1/knows>
        _:bnode4892021155019117627 ;
```

```
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type/>
<http://xmlns.com/foaf/0.1/Person> .
```

In the example, there are three triples generated in Turtle serialization using `sem:rdf-builder`. The triples represent the following facts; that person1 and person2 are people, and that their relationship is that person1 knows person2.

Note the following:

- The first parameter accepts an optional set of prefix mappings, which in this example is an empty argument. Since "empty" means default, the `$common-prefixes` are used for the first argument. The second argument is a named graph for the `sem:rdf-builder` output.
- In the predicate position, the special value of `"a"` is expanded to the full IRI for `rdf:type`.
- Human-readable CURIEs for common prefixes are used, such as `foaf:knows` instead of long IRIs. See "Working with CURIEs" on page 139.
- The blank nodes produced in the third triple match the identity of those defined in the first and second.

10.3 Extracting Triples from Content

With the `sem:rdf-builder` function you can easily extract triples from existing content or the results of a SPARQL query and quickly construct RDF graphs for querying or inserting into your database.

Assume that you have a web page that lists cities and countries that are sorted and ranked by the cost of living (COL), which is based on a Consumer Priced Index (CPI) and CPI-based inflation rate. The inflation rate is defined as the annual percent change in consumer prices compared with the previous year's consumer prices. Using a reference point of Monterrey, Mexico with an assigned a value of 100, the `Inflation` value of every other city in the database is calculated by comparing their COL to that of Monterrey. For example, an `Inflation` value of 150, means that the COL is 50% more expensive than living in Monterrey.

Ranking	City	Inflation *	Ranking	City	Inflation *
1	London (United Kingdom)	270	71	Minneapolis (United States)	134
2	Stockholm (Sweden)	266	72	Turin (Italy)	134
3	Zurich (Switzerland)	251	73	Shanghai (China)	133
4	Geneva (Switzerland)	247	74	Ljubljana (Slovenia)	132
5	New York City (United States)	225	75	Orlando (United States)	129
6	Singapore (Singapore)	219	76	Saint Petersburg (Russia)	125
7	San Francisco (United States)	209	77	Austin (United States)	125
8	Paris (France)	208	78	Phoenix (United States)	123
9	Sydney (Australia)	205	79	Almaty (Kazakhstan)	121
10	Brisbane (Australia)	203	80	Salt Lake City (United States)	120
11	Copenhagen (Denmark)	202	81	Beijing (China)	120
12	Oslo (Norway)	201	82	Lisbon (Portugal)	120
13	Tokyo (Japan)	200	83	Santiago (Chile)	120
14	Hong Kong (Hong Kong)	200	84	Montevideo (Uruguay)	119
15	Perth (Australia)	198	85	Belo Horizonte (Brazil)	116
16	Melbourne (Australia)	190	86	Brasília (Brazil)	116
17	Amsterdam (Netherlands)	189	87	Valencia (Spain)	116
18	Wellington (New Zealand)	186	88	Kansas City (United States)	116
19	Washington D.C. (United States)	185	89	Raleigh, North Carolina (United States)	115
20	Helsinki (Finland)	182	90	Istanbul (Turkey)	114
21	Dublin (Ireland)	182	91	Bangkok (Thailand)	114
22	Boston (United States)	174	92	Bogotá (Colombia)	111
23	Frankfurt am Main (Germany)	172	93	Zagreb (Croatia)	109
24	Toronto (Canada)	170	94	Oporto (Portugal)	108
25	Munich (Germany)	168	95	Taipei (Taiwan)	108
26	Manchester (United Kingdom)	166	96	Amman (Jordan)	107
27	Vancouver (Canada)	165	97	Curitiba (Brazil)	107
28	Tel Aviv (Israel)	165	98	Porto Alegre (Brazil)	107
29	Malmo (Sweden)	165	99	Kuala Lumpur (Malaysia)	105
30	Calgary (Canada)	164	100	Johannesburg (South Africa)	103

Note: These values are fictional and are not based on any official sources.

The underlying HTML code for the COL table may resemble the following:

```
<table class="city-index"
style="max-width:58%;float:left;margin-right:2em;">
    <thead>
        <tr>
            <th>Ranking</th>
            <th class="city-name">City</th>
            <th class="inflation">Inflation
            <a href="#inflation-explanation">*</a></th>
        </tr>
    </thead>

           <tbody><tr>
              <td class="ranking">1</td>
```

```
    <td class="city-name">
    <a href="http://www.example.org/IncreasedCoL/london">
    London (United Kingdom)</a></td>
    <td class="inflation">270</td>
  </tr>

  <tr>
    <td class="ranking">2</td>
    <td class="city-name">
    <a href="http://www.example.org/IncreasedCoL/stockholm">
    Stockholm (Sweden)</a></td>
    <td class="inflation">266</td>
  </tr>

  <tr>
    <td class="ranking">3</td>
    <td class="city-name">
    <a href="http://www.example.org/IncreasedCoL/zurich">
    Zurich (Switzerland)</a></td>
    <td class="inflation">251</td>
  </tr>

  <tr>
    <td class="ranking">4</td>
    <td class="city-name">
    <a href="http://www.example.org/IncreasedCoL/geneva">
    Geneva (Switzerland)</a></td>
    <td class="inflation">247</td>
  </tr>

  <tr>
    <td class="ranking">5</td>
    <td class="city-name">
    <ahref="http://www.example.org/IncreasedCoL/new-york">
    New York City (United States)</a></td>
    <td class="inflation">225</td>
  </tr>
```

This example uses the `sem:rdf-builder` function to extract triples from the HTML content. The function takes advantage of the fact that the HTML code is already well-formed and has a useful classification node (`@class`):

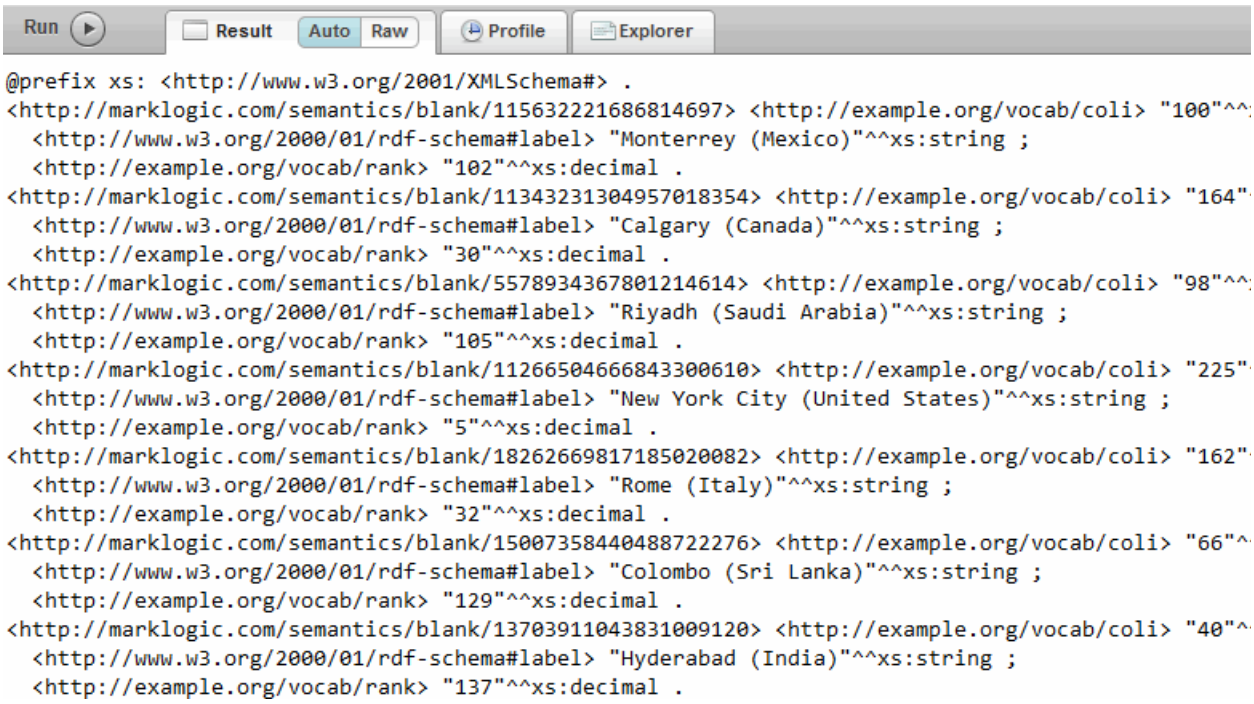
```
import module namespace sem = "http://marklogic.com/semantics"
  at "/MarkLogic/semantics.xqy";
declare namespace html="http://www.w3.org/1999/xhtml";

let $doc := xdmp:tidy(xdmp:document-get("C:\Temp\CoLIndex.html",
      <options xmlns="xdmp:document-get">
        <repair>none</repair>
        <format>text</format>
      </options>))[2]

let $rows := ($doc//html:tr)[html:td/@class eq 'ranking']
```

```
let $builder := sem:rdf-builder
                (sem:prefixes("my: http://example.org/vocab/"))
for $row in $rows
let $bnode-name := "_:" || $row/html:td[@class eq 'ranking']
return (
        $builder($bnode-name, "my:rank", xs:decimal(
                $row/html:td[@class eq 'ranking'] )),
        $builder($bnode-name, "rdfs:label", xs:string(
                $row/html:td[@class eq 'city-name'] )),
        $builder($bnode-name, "my:coli", xs:decimal(
                $row/html:td[@class eq 'inflation'] ))
        )
```

The results are returned as in-memory triples:



10.4 Parsing Triples

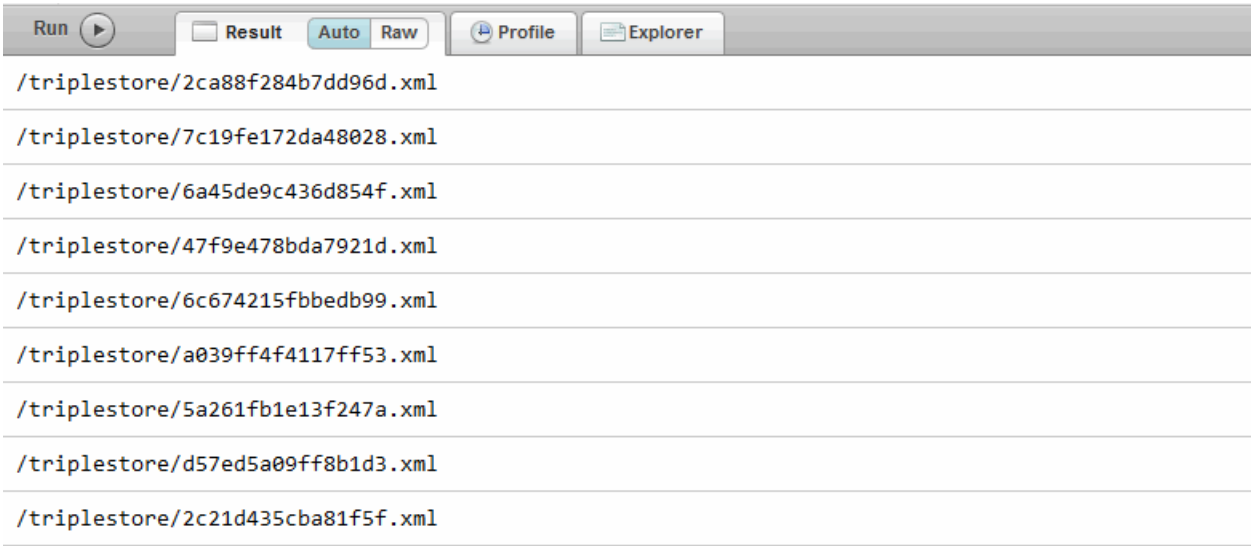
This example extends the previous example and inserts parsed triples into the database by using the `sem:rdf-insert` function:

```
import module namespace sem = "http://marklogic.com/semantics"
  at "/MarkLogic/semantics.xqy";
declare namespace html="http://www.w3.org/1999/xhtml";

let $doc := xdmp:tidy(xdmp:document-get("C:\Temp\CoLIndex.html",
      <options xmlns="xdmp:document-get">
        <repair>none</repair>
        <format>text</format>
      </options>))[2]
```

```
let $rows := ($doc//html:tr)[html:td/@class eq 'ranking']
let $builder := sem:rdf-builder(
                  sem:prefixes("my: http://example.org/vocab/"))
for $row in $rows
let $bnode-name := "_:" || $row/html:td[@class eq 'ranking']
let $triples := $row
return (sem:rdf-insert((
      $builder($bnode-name, "my:rank", xs:decimal
            ( $row/html:td[@class eq 'ranking'] )),
      $builder($bnode-name, "rdfs:label", xs:string
            ( $row/html:td[@class eq 'city-name'] )),
      $builder($bnode-name, "my:coli", xs:decimal
            ( $row/html:td[@class eq 'inflation'] )))
))
```

The document IRIs are returned as strings:

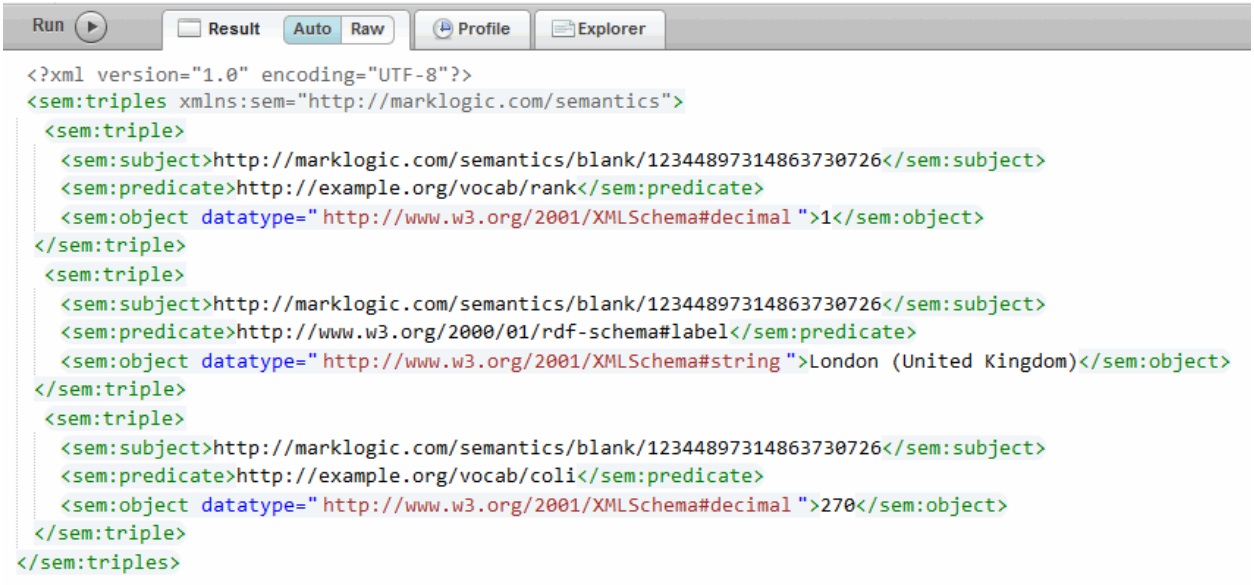


Note: For more information about inserting and parsing triples with XQuery, see “Loading Triples with XQuery” on page 51.

The parser ensures well-formed markup as the triples are inserted as schema-valid triples and indexed with the Triples index, provided it is enabled. See “Enabling the Triple Index” on page 66.

Use `fn:doc` to view the contents of the documents and verify the triples.

```
fn:doc("/triplestore/2ca88f284b7dd96d.xml")
```



One document is created for each blank node identifier (`$bnode-name`).

> **Note:** During the generation process `$builder` maintains state eliminating the need to keep track of every blank node label and ensuring that they map to the same `sem:blank` value.

The Semantics API includes a repair option for the N-Quad and Turtle parsers. During a normal operation, the RDF parsers perform these tasks:

- Turtle parsing uses the base IRI to resolve relative IRIs. If the result is relative, an error is raised.
- N-Quad parsing does not resolve using the base IRI. If a IRI in the document is relative, an error is raised.

During a repair operation the RDF parsers perform this task:

- Turtle parsing uses the base IRI to resolve relative IRIs. No error is raised for resultant relative IRIs.
- N-Quad parsing also uses the base IRI to resolve relative IRIs. No error is raised for resultant relative IRIs.

10.5 Exploring Data

The Semantics API provides functions to access RDF data in a database. This section focuses on the following topics:

- sem:triple Functions
- Transitive Closure

10.5.1 sem:triple Functions

This table describes the `sem:triple` functions used to define or search for triple data:

Function	Description
`sem:triple`	Creates a triple object that represents an RDF triple containing atomic values representing the subject, predicate, object, and optionally a graph identifier (graph IRI)
`sem:triple-subject`	Returns the subject from a `sem:triple` value
`sem:triple-predicate`	Returns the predicate from a `sem:triple` value
`sem:triple-object`	Returns the object from a `sem:triple` value
`sem:triple-graph`	Returns the graph identifier (graph IRI) from a `sem:triple` value

In this example, the `sem:triple` function is used to create a triple that includes a CURIE for the predicate and an `rdf:langString` value as the object, with English (`en`) as the given language tag:

```
sem:triple(sem:iri("http://id.loc.gov/authorities/subjects/
sh85040989"),
sem:curie-expand("skos:prefLabel"),
rdf:langString("Education", "en"))

=>
@prefix xs: <http://www.w3.org/2001/XMLSchema#> .

<http://id.loc.gov/authorities/subjects/sh85040989>
<http://www.w3.org/2004/02/skos/core#prefLabel/>
"Education"@en .
```

10.5.2 Transitive Closure

Transitive closure is a way to traverse a large section of a graph with a single lookup, applying a "follow relationship X" recursively.

10.5.2.1 Understanding Transitive Closure

A common use case is a thesaurus, where you have a term, and you want to find all broader terms, all broader terms for those terms, and all broader terms for those broader terms, and so forth. For example, if you have a taxonomy organized like this:

Mammal
Dog
Bichon

If you want to find all terms that are narrower terms for “mammal”, you can do a transitive closure of “mammal” over “narrower term” and find cat, dog, cow, Bichon, Siamese, Alsatian, chihuahua, Friesian, Jersey, and so forth.

Transitive closure queries are commonly used to explore taxonomies and ontologies such as the Simple Knowledge Organization System (SKOS). SKOS is “a common data model for knowledge organization systems such as thesauri, classification schemes, subject heading systems and taxonomies” as described by the W3C SKOS Simple Knowledge Organization System Reference:

http://www.w3.org/TR/skos-reference/

10.5.2.2 sem:transitive-closure

The sem:transitive-closure function has the following signature:

```
sem:transitive-closure(
   $seeds as sem:iri*,
   $predicates as sem:iri*,
   $limit as xs:integer
) as sem:iri*
```

This function takes seeds (subjects), predicates (relationships), and the depth to which to search, and returns all unique node IRIs.

Use the `sem:transitive-closure` function to traverse RDF graphs to answer reachability questions and discover more information about your RDF data. (In JavaScript, you would use the `sem.transitiveClosure` function.)

For example, assume that you have a file composed of triples for subject headings that relate to US Congress bills and that the triples are marked up with the SKOS vocabulary. The triples may look similar to this extract:

```
<http://id.loc.gov/authorities/subjects/sh85002310/>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type/>
<http://www.w3.org/2004/02/skos/core#Concept/> .

<http://id.loc.gov/authorities/subjects/sh85002310/>
<http://www.w3.org/2004/02/skos/core#prefLabel/>
"Agricultural education"@en .

<http://id.loc.gov/authorities/subjects/sh85002310/>
<http://www.w3.org/2008/05/skos-xl#altLabel/>
_:bnode7authoritiessubjectssh85002310 .

_:bnode7authoritiessubjectssh85002310
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type/>
<http://www.w3.org/2008/05/skos-xl#Label/> .
```

```
_:bnode7authoritiessubjectssh85002310
<http://www.w3.org/2008/05/skos-xl#literalForm/>
"Education, Agricultural"@en .

<http://id.loc.gov/authorities/subjects/sh85002310/>
<http://www.w3.org/2004/02/skos/core#broader/>
<http://id.loc.gov/authorities/subjects/sh85133121/> .

<http://id.loc.gov/authorities/subjects/sh85002310/>
<http://www.w3.org/2004/02/skos/core#narrower/>
<http://id.loc.gov/authorities/subjects/sh85118332/> .

<http://id.loc.gov/authorities/subjects/sh85133121/>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type/>
<http://www.w3.org/2004/02/skos/core#Concept/> .

<http://id.loc.gov/authorities/subjects/sh85133121/>
<http://www.w3.org/2004/02/skos/core#prefLabel/>
"Technical education"@en .
```

In this dataset, “Technical education” is a broader subject heading for “Agricultural education” as defined by the `skos:broader` predicate:

```
<http://id.loc.gov/authorities/subjects/sh85002310/>
<http://www.w3.org/2004/02/skos/core#broader/>
<http://id.loc.gov/authorities/subjects/sh85133121/> .
```

```
<sem:triple>
  <sem:subject>http://id.loc.gov/authorities/subjects/sh85002310</sem:subject>
  <sem:predicate>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</sem:predicate>
  <sem:object>http://www.w3.org/2004/02/skos/core#Concept</sem:object>
</sem:triple>
<sem:triple>
  <sem:subject>http://id.loc.gov/authorities/subjects/sh85002310</sem:subject>
  <sem:predicate>http://www.w3.org/2004/02/skos/core#prefLabel</sem:predicate>
  <sem:object xml:lang="en">Agricultural education</sem:object>
</sem:triple>
<sem:triple>
  <sem:subject>http://id.loc.gov/authorities/subjects/sh85002310</sem:subject>
  <sem:predicate>http://www.w3.org/2008/05/skos-xl#altLabel</sem:predicate>
  <sem:object>http://marklogic.com/semantics/blank/17142585114552908287</sem:object>
</sem:triple>
<sem:triple>

<sem:subject>http://marklogic.com/semantics/blank/17142585114552908287</sem:subject>
  <sem:predicatehttp://www.w3.org/1999/02/22-rdf-syntax-ns#type></sem:predicate>
  <sem:object>http://www.w3.org/2008/05/skos-xl#Label</sem:object>
</sem:triple>
<sem:triple>

<sem:subject>http://marklogic.com/semantics/blank/17142585114552908287</sem:subject>
  <sem:predicate>http://www.w3.org/2008/05/skos-xl#literalForm</sem:predicate>
  <sem:object xml:lang="en">Education, Agricultural</sem:object>
</sem:triple>
<sem:triple>
  <sem:subject>http://id.loc.gov/authorities/subjects/sh85002310</sem:subject>
  <sem:predicate>http://www.w3.org/2004/02/skos/core#broader</sem:predicate>
  <sem:object>http://id.loc.gov/authorities/subjects/sh85002310</sem:object>
</sem:triple>
```

This example uses `cts:triples` to find the subject IRI for a triple where the predicate is a CURIE for `skos:prefLabel` and the object is `Agricultural education`. The subject IRI found in the `cts:triples` query is subsequently used with `skos:broader` to determine broader subject terms to a depth of `3`:

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

let $triple-subject := sem:triple-subject(cts:triples((), sem:curie-
expand("skos:prefLabel"),
rdf:langString("Agricultural education", "en")))
return
sem:transitive-closure($triple-subject, sem:curie-
expand("skos:broader"), 3)

=>
<http://id.loc.gov/authorities/subjects/sh85133121/>
<http://id.loc.gov/authorities/subjects/sh85002310/>
<http://id.loc.gov/authorities/subjects/sh85026423/>
<http://id.loc.gov/authorities/subjects/sh85040989/>
```

Notice that in addition to the expected IRIs, for the following subjects:

- `<http://id.loc.gov/authorities/subjects/sh85002310/>`
- `<http://id.loc.gov/authorities/subjects/sh85133121/>`

IRIs were returned also in the results for the following subjects:

- `<http://id.loc.gov/authorities/subjects/sh85040989/>`
- `<http://id.loc.gov/authorities/subjects/sh85026423/>`

When we take a closer look at the dataset, the IRIs for “Education” and “Civilization” are also returned, since they are broader subjects still to “Agricultural education” and “Technical Education”:

```
<http://id.loc.gov/authorities/subjects/sh85040989/>
<http://www.w3.org/2004/02/skos/core#prefLabel/>
"Education"@en .
<http://id.loc.gov/authorities/subjects/sh85040989/>
<http://www.w3.org/2004/02/skos/core#broader/>
<http://id.loc.gov/authorities/subjects/sh85026423/> .
...
<http://id.loc.gov/authorities/subjects/sh85026423/>
<http://www.w3.org/2004/02/skos/core#prefLabel/>
"Civilization"@en .
...
```

11.0 Client-Side APIs for Semantics

MarkLogic Semantics can be accessed through client-side APIs that provide support for management of triples and graphs, SPARQL and SPARQL Update, and access to the search features of MarkLogic server.

The chapter includes the following sections:

- Java Client API
- Node.js Client API
- Queries Using Optic API

11.1 Java Client API

The Java Client API enables you to create client-side Java applications that interact with MarkLogic. Semantics related features include support for graph and triple management, SPARQL Query, SPARQL Update, and Optic queries.

For details, see Working With Semantic Data in the *Java Application Developer's Guide* and the following interfaces and classes in the `com.marklogic.client.semantics` package in the *Java Client API Documentation*.

- `GraphManager`
- `SPARQLQueryManager`
- `SPARQLQueryDefinitions`

11.2 Node.js Client API

The Node.js Client API can be used for CRUD (Create, Read, Update, and Delete) operations on graphs; creating, reading, updating, and deleting triples and graphs. The `DatabaseClient.graphs.write` function can be used to create a graph containing triples, the `DatabaseClient.graphs.read` function reads from a graph. The `DatabaseClient.graphs.remove` function removes a graph. The `DatabaseClient.graphs.sparql` function queries semantic data.

See Working With Semantic Data in the *Node.js Application Developer's Guide* for more details. The Node.js Client source can be found on GitHub at http://github.com/marklogic/node-client-api. For additional operations, see the *Node.js Client API Reference*.

> **Note:** These operations only work with managed triples contained in a graph. Embedded triples cannot be manipulated using the Node.js Client API.

11.3 Queries Using Optic API

The Optic API can be used to search and work with semantic triples in both client-side queries and server-side side queries. Optic can be used for triple data client-side queries with the Java Client API and the REST Client API, but not with Node.js. See Optic Java API for Relational Operations in the *Java Application Developer's Guide* and Invoking an Optic API Query Plan in the *REST Application Developer's Guide* for more details.

For server-side queries using the Optic API, see "Querying Triples with the Optic API" on page 145 for more information. Also, see the `op:from-triples` or `op.fromTriples` functions in the Optic API and the Data Access Functions section in the *Application Developer's Guide* .

12.0 Inserting, Deleting, and Modifying Triples with XQuery and Server-Side JavaScript

Triples can be modified with XQuery or Server-side JavaScript, using MarkLogic `xdmp` built-ins. Triples managed by MarkLogic - those triples having a *document root* of `sem:triples` - can be modified using SPARQL Update. See "Using SPARQL Update" on page 170 for more information about modifying managed triples.

"Unmanaged" triples, those triples embedded in another document with an *element node* of `sem:triple`, can only be modified using XQuery or Server-Side JavaScript and `xdmp` built-ins. To perform updates on triples in your datastore (for either managed or unmanaged triples), you insert a new triple and delete the existing one. You are not updating the existing triple; the update operation is actually an `INSERT`/`DELETE` procedure.

This chapter includes the following sections:

- Updating Triples
- Deleting Triples

12.1 Updating Triples

You can use XQuery or Server-Side JavaScript functions to update existing triples in a database, by using `INSERT`/`DELETE` to replace nodes. For a managed triple, the `sem:database-nodes` (`sem.databaseNode` in Server-Side JavaScript) and the `xdmp:node-replace` (`xdmp.nodeReplace` in Server-Side JavaScript) functions are used to correct inaccurate data.

Assume the database contains a document containing the following unmanaged triple, with the resource "John Doe" entered as "John_Doe":

```
<sem:triples xmlns:sem="http://marklogic.com/semantics">
 <sem:triple>
  <sem:subject>http://dbpedia.org/resource/John_Doe</sem:subject>
  <sem:predicate>http://www.w3.org/1999/02/22-rdf-syntax-
ns#type</sem:predicate>
  <sem:object>http://xmlns.com/foaf/0.1/Person/</sem:object>
 </sem:triple>
</sem:triples>
```

The following example replaces the subject with "http://dbpedia.org/resource/John_Doe" using the `xdmp:node-replace` function. The example uses `sem:rdf-builder` to construct a triple that matches the one we want to change. Using this triple with `sem:database-nodes` finds the matching nodes in the database to be changed.

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

(: construct the triple to match against :)
```

```
let $builder := sem:rdf-builder(
  sem:prefixes("dbpedia: http://dbpedia.org/resource/"))
let $triple := $builder(
  "dbpedia:John_Doe", "a", "foaf:Person")
(: find matching unmanaged triples in the database :)
let $node := sem:database-nodes($triple)
(: construct the replacement triple with a new subject :)
let $replace :=
  <sem:triple>
    <sem:subject>http://dbpedia.org/resource/John_Doe</sem:subject>
    {$node[1]/sem:predicate, $node[1]/sem:object}
  </sem:triple>

(: replace the old triple with the new one in all matched nodes :)
return $node ! xdmp:node-replace($node, $replace);
```

The following example performs the same operation using Server-Side JavaScript. The example uses the NodeBuilder interface to construct the replacement node.

```
declareUpdate();
const sem = require('/MarkLogic/semantics');

// construct the triple to find in the database
const builder =
  sem.rdfBuilder(sem.prefixes('dbpedia: http://dbpedia.org/resource/'));
const triple = xdmp.apply(builder,
  'dbpedia:John_Doe', 'a', 'foaf:Person');

for (let node of sem.databaseNodes(triple)) {
  // construct the replacement triple with the new subject
  const pred = fn.head(
    node.xpath('sem:predicate',
               {'sem': 'http://marklogic.com/semantics'}));
  const obj = fn.head(
    node.xpath('sem:object',
               {'sem': 'http://marklogic.com/semantics'}));
  const replacement = new NodeBuilder()
    .startElement('sem:triple','http://marklogic.com/semantics/')
      .addElement(
        'sem:subject',
        'http://dbpedia.org/resource/John_Doe',
        'http://marklogic.com/semantics')
      .addNode(pred)
      .addNode(obj)
    .endElement()
    .toNode();
  // replace the old triple with the new one
  xdmp.nodeReplace(node, replacement)
}
```

When you have multiple triples to update, you can use XQuery or Server-Side JavaScript (or if they are managed triples, SPARQL Update), to find matching triples, and then iterate over the nodes to replace them.

In this example, a `cts:triples` call finds all triples with “John_Doe” in the subject position and replaces each occurrence with “Jane_Roe”:

```
xquery version "1.0-ml";
import module namespace sem = "http://marklogic.com/semantics"
  at "/Marklogic/semantics.xqy";

let $triples :=
cts:triples(sem:iri("http://dbpedia.org/resource/John_Doe"),(),())
for $triple in  $triples
let $node := sem:database-nodes($triple)
let $replace :=
    <sem:triple>
       <sem:subject>http://dbpedia.org/resource/Jane_Roe</sem:subject>
       {$node/sem:predicate, $node/sem:object}
    </sem:triple>
return $node ! xdmp:node-replace(., $replace)
```

An empty sequence is returned for both of the examples because the replacements have been made. Use a simple `cts:triples` call to verify that the updates have been made:

```
cts:triples(sem:iri("http://dbpedia.org/resource/John_Doe"),
           (),())
```

Note: Using the `xdmp:node-replace` function results in creating a new fragment and deleting the old fragment. When the system performs a merge, the deleted fragments are removed permanently. The system performs automatic merges, unless this feature has been disabled by an administrator.

12.2 Deleting Triples

This section discusses methods for deleting RDF data in MarkLogic and includes the following topics:

- Deleting Triples with XQuery or Server-Side JavaScript
- Deleting Triples with REST API

12.2.1 Deleting Triples with XQuery or Server-Side JavaScript

There are several functions you can use to delete triples from a database. This section discusses the following functions:

- sem:graph-delete
- xdmp:node-delete
- xdmp:document-delete

12.2.1.1 sem:graph-delete

Note: This function only works for managed triples.

You can use the `sem:graph-delete` function to delete all managed triple documents in a named graph. You specify the graph IRI as the parameter.

For example:

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

sem:graph-delete(sem:iri("mynamedgraph"))
```

In Server-Side JavaScript the command would be:

```
const sem = require("/marklogic/semantics.xqy");

sem.graphDelete(sem.iri("mynamedgraph"));
```

The following example deletes all managed triples in the default graph. If no other named graphs exist, this might remove all triples from the database:

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

sem:graph-delete(
  sem:iri("http://marklogic.com/semantics#default-graph"))
```

Note: The `sem:graph-delete` function will only delete triples inserted by the Graph Store API, which have a document root element of `sem:triple`. If you delete a specific named graph, it will not affect documents with embedded triples (with a `sem:triples` element node), so the graph might still exist.

12.2.1.2 xdmp:node-delete

To delete a set of triples from the database, use the `sem:database-nodes` function with `xdmp:node-delete` in XQuery, or `sem.databaseNodes` with `xdmp.nodeDelete` in Server-Side JavaScript. This function works with managed or unmanaged triples.

For example:

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

let $triples :=
cts:triples(sem:iri("http://www.rdfabout.com/rdf/usgov/congress/people
/D000596"),()())

for $triple in $triples
return (sem:database-nodes($triple) ! xdmp:node-delete(.))
```

Note: This query will not delete empty `sem:triple` document elements if all the triples are deleted from a single document.

In Server-Side JavaScript, the example would look like this:

```
const sem = require('/MarkLogic/semantics');
const triples = cts.triples(
  sem.iri('http://www.rdfabout.com/rdf/usgov/congress/people/D000596'),
  null, null);

for (let triple of triples) {
  for (let node of xdm.databaseNodes(triple)) {
    xdmp.nodeDelete(node)
  }
}
```

12.2.1.3 xdmp:document-delete

You can remove documents containing triples from the database with the `xdmp:document-delete` function. This function deletes a document and all of its properties, and works with both managed and unmanaged triples. Specify the IRI of the document to be deleted as the parameter.

The following XQuery example deletes the document with URI “example.xml”:

```
xquery version "1.0-ml";
xdmp:document-delete("example.xml")
```

The following example performs the equivalent operation in Server-Side JavaScript:

```
declareUpdate();
const sem = require('/MarkLogic/semantics.xqy');

xdmp.documentDelete('example.xml');
```

Deleting a document deletes the document, any triples embedded in the document, and the document properties.

To delete all documents in a directory, use the `xdmp:directory-delete` function.

12.2.2 Deleting Triples with REST API

You can use the REST API to delete triples in the default graph or a named graph by sending a `DELETE` request to the `DELETE:/v1/graphs` service. To delete triples from a named graph, use `curl` to send the `DELETE` request in the following form:

```
http://host:port/version/graphs?graph=graph-iri
```

where *graph-iri* is the IRI of your named graph.

The IRI for the named graph in the request is `http://host:port/version/graphs?default`. For example, this `DELETE` request removes all triples from the default graph at port 8321:

```
#Windows users, see Modifying the Example Commands for Windows

$ curl --anyauth --user user:password -X DELETE \
http://localhost:8321/v1/graphs?default
```

Note: Use caution when specifying the graph, since there is no confirmation check before deleting the dataset.

This `curl` command will delete the triples in the graph named `mygraph`.

```
#Windows users, see Modifying the Example Commands for Windows

$ curl --anyauth --user user:password -X DELETE \
http://localhost:8321/v1/graphs?graph=http://marklogic.com/semantics#m
ygraph/
```

As with the `sem:graph-delete` function, the `DELETE` request removes triples from graphs where `sem:triples` is the root element of the containing document (managed triples). XML documents that contain embedded triples are unaffected. Graphs may still exist after the `DELETE` operation if the graph contained both types of documents.

When you send a `PUT` request, triples are replaced in a named graph or added to an empty graph if the graph did not exist. This is the equivalent of a `DELETE` followed by `POST`.

For example:

```
# Windows users, see Modifying the Example Commands for Windows

$ curl --digest --user admin:password -s -X PUT
-H "Content-type:text/turtle" --data-binary '@./example.ttl'
    "http://localhost:8033/v1/graphs?graph=mynamed-graph"
```

To perform the equivalent of a `DELETE` operation using the REST API, use `curl` to send a `PUT` request with an empty graph.

13.0 Using a Template to Identify Triples in a Document

You can define a template to identify data to be indexed as triples in an existing document. Documents with any type of data that you want to represent as triples can be indexed using a template. The triples identified by the template are similar to unmanaged triples, sometimes called *embedded triples*.

Once you have indexed these triples, you can query them in all the same ways you can query unmanaged triples; with SPARQL, with `xdmp.sparql()`, with combination queries, with the new Optic API, and with `cts:triple-range-query`. For more about working with these triples, see "Unmanaged Triples" on page 73. For a more complete discussion of creating and using templates, see Template Driven Extraction (TDE) in the *Application Developer's Guide*.

This chapter covers the following topics:

- Creating a Template
- Template Elements
- Examples
- Triples Generated With TDE and SQL

13.1 Creating a Template

Here is an example of a simple template to identify triples. It includes a definition for a namespace and context for the template. It contains descriptions for the subject, object, predicate of the triples, and data mappings for the values:

```
<template xmlns="http://marklogic.com/xdmp/tde">
  <context>/article/topic</context>
    <vars>
      <var>
        <name>EX</name>
        <val>"http://example.org/ex#"</val>
      </var>
    </vars>
    <triples>
     <triple>
      <subject>
        <val>sem:iri( $EX || who )</val>
      </subject>
      <predicate>
        <val>sem:iri( $EX || what )</val>
      </predicate>
      <object>
        <val>xs:string( $EX || where )</val>
      </object>
    </triple>
   </triples>
  </template>
```

For triples, the subject and predicate descriptions must have a value of `sem:iri`. Here the template incorporates using `vars` as a short-hand, to save typing when you specify IRIs. When creating templates to identify triples, you can specify the types of values that you extract using a subset of XQuery language expressions. See Template Dialect and Data Transformation Functions in the *Application Developer's Guide* for more information.

Note: Triples identified using templates cannot be modified directly or modified as triples (for example, using SPARQL Update). You can disable and then delete a template so that the triples no longer exist, or you can modify the underlying document data to modify the triple.

Security for templates can be controlled by setting protected collections. See Security on TDE Documents in the *Application Developer's Guide*.

13.2 Template Elements

A template contains the following elements and their child elements:

Element	Description
`context`	The lookup node that is used for template activation and data extraction. See Context in the *Application Developer's Guide* for more details.
`description`	Optional description of the template.

`collections`
`  collection`
`  collections-and`
`  collection`
	Optional collection scopes. Multiple collection scopes can be ORed or ANDed.
	
	A `<collections>` section is a top level OR of a sequence of:
	
	• `<collection>` that scope the template to a specific collection.
	• `<collections-and>` that contains a sequence of `<collection>` that are ANDed together.
	
	See Collections in the *Application Developer's Guide* for more details.

`directories` `  directory`	Optional directory scopes. Multiple directory scopes are ORed together.

Element	Description
`vars` `  var`	Optional intermediate variables extracted at the current context level. This element can be used as a short hand for IRIs (prefixes) in triples. See Variables in the *Application Developer's Guide* for more details.

`triples`
`  triple`
`  subject`
`  val`
`  invalid-values`
`  predicate`
`  val`
`  invalid-values`
`  object`
`  val`
`  invalid-values`
	These elements are used for triple-extraction templates.
	
	`triples` contains a sequence of triple extraction descriptions. Each `triple` description defines the data mapping for the `subject`, `predicate` and `object`.
	
	An extracted triples graph cannot be specified through the template. The graph is implicitly defined by the document's collection similar to embedded triples.

`templates` `  template`	Optional sequence of sub-templates. For details, see Creating Views from Multiple Templates and Creating Views from Nested Templates in the *SQL Data Modeling Guide*.
`path-namespaces` `  path-namespace`	Optional sequence of namespace bindings. See path-namespaces in the *Application Developer's Guide* for more details.
`enabled`	A boolean that specifies whether the template is enabled (`true`) or disabled (`false`). The default value is `true`.

The `context`, `vars`, and `triples` elements identify XQuery elements or JSON properties by means of path expressions. The `var` element can be used to specify a prefix for elements in the triple.

For example:

```
<vars>
 <var>
   <name>ex</name>
   <val>"http://example.org/ex#"</val>
 </var>
</vars>
```

Path expressions are based on XPath, which is described in XPath Quick Reference in the *XQuery and XSLT Reference Guide* and Traversing JSON Documents Using XPath in the *Application Developer's Guide*.

13.2.1 Reindexing Triggered by Templates

When adding or modifying a triple template, reindexing is triggered and the triples extracted by the template are available as soon as they start to appear in the triple index. Note that only documents matching the context element, and the directory and collection scopes will be re-indexed, so choose these carefully to avoid unnecessary (re)indexing work.

- For a new template, triples appear in the index as documents are indexed.
- For modified templates (and until reindexing is complete), there could be a mix of existing triples extracted with the previous version of the template (for the documents that haven't been reindexed yet) along with new triples extracted by the newer version of the template (for those documents that have been reindexed).

13.3 Examples

This section contains examples of different ways that you can validate and use templates to identify triples in documents.

- Validate and Insert a Template
- Validate and Insert in One Step
- Use a JSON Template
- Identify Potential Triples

13.3.1 Validate and Insert a Template

For this example, insert this document into the Documents database using the Query Console. This document is used as the source of the triples.

```
xdmp:document-insert("APNews.xml",
<article>
 <info>APNewswire - Nixon went to China</info>
 <triples-context>
  <confidence>80</confidence>
  <published>2011-10-14</published>
  <source>AP News</source>
 </triples-context>
 <topic>
  <who>Nixon</who>
  <what>wentTo</what>
  <where>China</where>
  </topic>
 <body>
  In 1974, Richard Nixon went to China.
 </body>
</article>
)
```

Using the Query Console, we will validate this template (`APtemplate.xml`) and then insert it into a collection called `http://marklogic.com/xdmp/tde` in the Schemas database. First validate the template:

```
let $t1 :=
<template xmlns="http://marklogic.com/xdmp/tde">
  <context>/article/topic</context>
    <vars>
      <var>
        <name>EX</name>
        <val>"http://example.org/ex#"</val>
      </var>
    </vars>
  <triples>
    <triple>
      <subject>
        <val>sem:iri( $EX || who)</val>
      </subject>
      <predicate>
        <val>sem:iri( $EX || what)</val>
      </predicate>
      <object>
        <val>xs:string( $EX || where)</val>
      </object>
    </triple>
  </triples>
  </template>

return tde:validate($t1)
=>
<map:map xmlns:map="http://marklogic.com/xdmp/map"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <map:entry key="valid">
    <map:value xsi:type="xs:boolean">true
    </map:value>
  </map:entry>
</map:map>
```

Next insert the valid template. Use `tde:template-insert`. This takes care of putting the template into the Schemas database and into the correct collection:

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

let $t1 :=
<template xmlns="http://marklogic.com/xdmp/tde">
  <context>/article/topic</context>
    <vars>
      <var>
        <name>EX</name>
        <val>"http://example.org/ex#"</val>
```

```
      </var>
    </vars>
  <triples>
    <triple>
      <subject>
        <val>sem:iri( $EX || who)</val>
      </subject>
      <predicate>
        <val>sem:iri( $EX || what)</val>
      </predicate>
      <object>
        <val>xs:string( $EX || where)</val>
      </object>
    </triple>
  </triples>
  </template>

return tde:template-insert(
"APtemplate.xml",$t1, (), "http://marklogic.com/xdmp/tde")
```

When you use the template, content in the document will be indexed as a triple. The triple is not added to the original document. To see the triple, run this query in Query Console:

```
tde:node-data-extract(fn:doc("APNews.xml"));
```

This returns the name of the document and the content that was indexed as a triple.

```
=>
{"APNews.xml": [
 {
  "triple": {
   "subject": "http://example.org/ex#Nixon",
   "predicate": "http://example.org/ex#wentTo",
   "object": {
    "datatype": "http://www.w3.org/2001/XMLSchema#string",
    "value": "http://example.org/ex#China"
   }
  }
 }
]}
```

Use this SPARQL query to verify that the triple is in the triple index:

```
SELECT ?country
WHERE {
  <http://example.org/ex#Nixon> <http://example.org/ex#wentTo>
  ?country
}

=> China
```

13.3.2 Validate and Insert in One Step

The next example uses `tde:template-insert` to both validate and insert the template into the Schemas database associated with this content database in one step. For this example, we'll insert a document described in "Unmanaged Triples" on page 73.

The following code inserts the document into the Documents database in a "SAR" collection:

```
xquery version "1.0-ml";
xdmp:document-insert("SAR_report.xml",
<SAR>
 <title>Suspicious vehicle...Suspicious vehicle near airport</title>
  <date>2015-11-12Z</date>
  <type>observation/surveillance</type>
  <threat>
   <type>suspicious activity</type>
    <category>suspicious vehicle</category>
  </threat>
  <location>
    <lat>37.497075</lat>
    <long>-122.363319</long>
  </location>
  <description>A blue van with license plate ABC 123 was observed
parked behind the airport sign...
    <sem:triple>
      <sem:subject>IRIID</sem:subject>
      <sem:predicate>isa</sem:predicate>
       <sem:object
datatype="http://www.w3.org/2001/XMLSchema#string">license-
plate</sem:object>
    </sem:triple>
    <sem:triple>
      <sem:subject>IRIID</sem:subject>
      <sem:predicate>value</sem:predicate>
      <sem:object
datatype="http://www.w3.org/2001/XMLSchema#string">ABC
123</sem:object>
    </sem:triple>
  </description>
</SAR>,(),
"SAR")
```

This document already has two embedded triples. Now let us identify another triple describing the date and type of threat described in the report. We will create a template to identify the triple and insert it using `tde:template-insert`, which validates the template and then inserts it into the Schemas database.

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

let $template :=
<template xmlns="http://marklogic.com/xdmp/tde">
  <context>/SAR</context>
    <triples>
    <triple>
      <subject>
        <val>sem:iri(threat/type)</val>
      </subject>
      <predicate>
        <val>sem:iri("http://example.org/on-date")</val>
      </predicate>
      <object>
        <val>xs:date(date)</val>
      </object>
    </triple>
    </triples>
</template>
return tde:template-insert("SARtemplate.xml", $template)
```

To see the new triple, run this query using `tde:node-data-extract` in Query Console:

```
tde:node-data-extract(fn:doc("SAR_report.xml"));

=>
{
 "SAR_report.xml": [
  {
   "triple": {
    "subject": "suspicious activity",
    "predicate": "http://example.org/on-date",
    "object": {
      "datatype": "http://www.w3.org/2001/XMLSchema#date",
      "value": "2015-11-12Z"
    }
   }
  }
 ]
}
```

To see all the triples in this document, run this SPARQL query restricted to the “SAR” collection, in the Query Console:

```
SELECT *
FROM <SAR>
WHERE {
  ?s ?p ?o
}
```

This returns all of the triples in the `SAR_report.xml` document:

```
s                           p                              o
<suspicious activity>    <http://example.org/on-date>  2014-11-
12Z"^^xs:date
<IRIID>                  <isa>                         <license-plate>
<IRIID>                  <value>                       <ABC 123>
```

13.3.3 Use a JSON Template

You can use a JSON template to identify triples in a JSON document.

> **Note:** Any template (XML or JSON) will extract triples from any document (XML or JSON).

Insert this document into the Documents database:

```
declareUpdate();
xdmp.documentInsert("/medlineCitation.json", ({
   "MedlineCitation": {
    "Status": "Completed",
    "MedlineID": 69152893,
    "PMID": 5717905,
    "Article": {
      "Journal": {
        "ISSN": "0043-5341"
      },
      "ArticleTitle": "[On the influence of calcium ... on cholesterol
in human serum]",
      "AuthorList": {
        "Author": [
          {
            "LastName": "Doe",
            "ForeName": "John"
          },
          {
            "LastName": "Smith",
            "ForeName": "Jane"
          }
        ]
      }
    }, "collections" : "http://marklogic.com/xdmp/tde"}
}));
```

Now validate and insert a JSON template. The `tde.templateInsert` command validates the template and inserts it into the Schemas database.

```
declareUpdate();
var tde = require ("/MarkLogic/tde.xqy");

var template = xdmp.toJSON({
  "template":{
    "context":"/MedlineCitation/Article",
    "vars":[
      {
        "name":"prefix1",
        "val":"\"http://marklogic.com/example/\""
      }
    ],
    "triples":[{
        "subject":{
        "val":"sem:iri($prefix1||'person/'||AuthorList/Author[1] \
                    /ForeName||'_'||AuthorList/Author[1]/LastName)"},
        "predicate":{
          "val":"sem:iri(($prefix1||'authored'))"},
        "object":{
          "val":"xs:string(Journal/ISSN)"}
         }
        ] }});

tde.templateInsert("medlineTemplate.json", template);

// After validating the template, this inserts template into the
Schemas
database as medlineTemplate.json
```

Run this query against the Documents database in the Query Console. This query identifies the first author in the document in the form of a triple:

```
tde.nodeDataExtract([fn.doc("/medlineCitation.json")]);
=>
{
 "/medlineCitation.json": [
  {
  "triple": {
    "subject": "http://marklogic.com/example/person/John_Doe",
    "predicate": "http://marklogic.com/example/authored",
    "object": {
      "datatype": "http://www.w3.org/2001/XMLSchema#string",
      "value": "0043-5341"
    }
   }
  }
 ]
}
```

Note: The `nodeDataExtract` command is a helper utility to show you how the template view looks. Normally you would run a SQL or SPARQL query against the generated view.

This template only extracts the first author's name along with the ISSN number. You can change the `[1]` to a `[2]` in the template to extract the second author's name.

13.3.4 Identify Potential Triples

This next example includes both the document and the template used to identify two triples as part of one query that you can paste into Query Console. The `tde:node-data-extract` is a helping function to show you what *would* be indexed if you did insert this document and template.

```
let $doc1 :=
<MedlineCitation Status="Completed">
  <MedlineID>69152893</MedlineID>
  <PMID>5717905</PMID>
  <Article>
    <Journal>
      <ISSN>0043-5341</ISSN>
      <JournalIssue>
        <Volume>118</Volume>
        <Issue>49</Issue>
        <PubDate>
          <Year>1968</Year>
          <Month>Dec</Month>
          <Day>7</Day>
        </PubDate>
      </JournalIssue>
    </Journal>
    <ArticleTitle>[On the influence of calcium ... on cholesterol in
human serum]</ArticleTitle>
    <AuthorList>
      <Author>
        <LastName>Doe</LastName>
        <ForeName>John</ForeName>
      </Author>
      <Author>
        <LastName>Smith</LastName>
        <ForeName>Jane</ForeName>
      </Author>
    </AuthorList>
  </Article>
</MedlineCitation>

let $template1 :=
<template xmlns="http://marklogic.com/xdmp/tde">
  <context>/MedlineCitation/Article/AuthorList/Author</context>
  <triples>
    <triple>
      <subject>
        <val>sem:iri(concat(ForeName,' ',LastName))</val>
```

```
      </subject>
      <predicate>
        <val>sem:iri('authored')</val>
      </predicate>
      <object>
        <val>xs:string(../../ArticleTitle)</val>
      </object>
    </triple>
  </triples>
</template>

return tde:node-data-extract (($doc1), ($template1))
```

This query returns the two triples that would be added to the triple index in JSON format:

```
{
 "document1": [
   {
   "triple": {
   "subject": "John Doe",
   "predicate": "authored",
   "object": {
     "value": "[On the influence of calcium ... on cholesterol in human
serum]"
    }
   }
  },
  {
   "triple": {
   "subject": "Jane Smith",
   "predicate": "authored",
   "object": {
     "datatype": "http://www.w3.org/2001/XMLSchema#string",
     "value": "[On the influence of calcium ... on cholesterol in human
serum]"
    }
   }
  }
 ]
}
```

These triples in this example have *not* been added to the triple index, but you can see how the template works and what triples would be indexed if you inserted the document and template.

Note: The graph for these triples cannot be specified through the template. The graph is implicitly defined by the document's collection, similar to embedded triples.

13.4 Triples Generated With TDE and SQL

Some TDE views created for SQL will generate index entries that are present, visible, and usable as triples due to the underlying implementation of SQL using the triples index. Those triples may then appear in SPARQL query results.

These triples have very distinctive subject and predicate URIs, so as long as a SPARQL query includes some subject or some predicate filter, the triples generated by a row template will not appear in your results.

This is an example of a triple generated from a row template:

```
<http://marklogic.com/row/09CA32CBA69361E5/8FD41B78E884B48E>
  <http://marklogic.com/column/id/81C579F95CEA957B>
  "George Washington"
```

Some SPARQL operations where these row triples may appear include:

1. A SPARQL query for “show me all triples”. When you are initially trying out SPARQL, you might load 10 triples and run this SPARQL query:

   ```
   SELECT *
   WHERE {
      ?s ?p ?o }
   ```

   Note: For performance reasons, do not run this query on any database with numerous triples because the query will return all of the triples in the database.

2. A SPARQL query to “count all triples”. This is similar to the preceding query, and would also access all of the triples in the database.

3. A SPARQL query to “show me all distinct predicates”. This is another common way to explore your triples data.

To avoid seeing row triples returned as part of these queries, insert and query triples from a named graph, or include a subject or predicate filter to exclude the row triples.

Note: A best practice is to insert triples into a named graph and query from that graph.

For more information about using the Optic API with triples for server-side queries see “Querying Triples with the Optic API” on page 145, the `op:from-triples` or `op.fromTriples` functions, and Data Access Functions and Optic API for Multi-Model Data Access in the *Application Developer’s Guide*. For information about using the Optic API for client-side queries, see “Queries Using Optic API” on page 238 and Optic Java API for Relational Operations in the *Java Application Developer’s Guide*. Also see /REST/client/row-management in the Client API reference and the row manager and rows endpoint in the *REST Application Developer’s Guide*.

For information about using templates with SQL content, see Creating Template Views in the *SQL Data Modeling Guide*.

14.0 Execution Plan

This section describes how to interpret a query execution plan output from the `sem:sparql-plan` function. The generated query execution plan shows how the supplied query will be handled by the SPARQL parser. The query execution plan for SPARQL is designed to work with all `SPARQL` queries, including `SELECT`, `CONSTRUCT`, `ASK` and `DESCRIBE`.

14.1 Generating an Execution Plan

You can use `sem:sparql-plan` to generate a query execution plan for a `SPARQL` query to see how the query will be handled internally.

For example, this SPARQL query produces an execution plan:

```
sem:sparql-plan("select * { ?s ?p ?o }",(),"optimize=1")
```

The query outputs the following execution plan:

```
<plan:plan xmlns:plan="http://marklogic.com/plan">
 <plan:select>
  <plan:project order="1,0,2">
   <plan:variable name="s" column-index="0" static-type="NONE">
   </plan:variable>
   <plan:variable name="p" column-index="1" static-type="NONE">
   </plan:variable>
   <plan:variable name="o" column-index="2" static-type="NONE">
   </plan:variable>
   <plan:triple-index order="1,0,2" permutation="PSO" dedup="true">
    <plan:subject>
      <plan:variable name="s" column-index="0" static-type="NONE">
     </plan:variable>
    </plan:subject>
    <plan:predicate>
     <plan:variable name="p" column-index="1" static-type="NONE">
     </plan:variable>
    </plan:predicate>
    <plan:object>
     <plan:variable name="o" column-index="2" static-type="NONE">
     </plan:variable>
    </plan:object>
   </plan:triple-index>
  </plan:project>
 </plan:select>
</plan:plan>
```

14.2 Parsing an Execution Plan

This section breaks down and describes each portion of the execution plan.

The intro specifies the type of plan, in this case for a SELECT statement:

```
<plan:plan xmlns:plan="http://marklogic.com/plan">
 <plan:select>
```

This section identifies the projected order of the elements of the triples–subject, predicate, object–with their variable names (s,p,o) and column indexes:

```
<plan:project order="1,0,2">
   <plan:variable name="s" column-index="0" static-type="NONE">
   </plan:variable>
   <plan:variable name="p" column-index="1" static-type="NONE">
   </plan:variable>
   <plan:variable name="o" column-index="2" static-type="NONE">
   </plan:variable>
```

At the end is the order of the triple variables in the triple index - predicate, subject, object (p,s,o).

```
   <plan:triple-index order="1,0,2" permutation="PSO" dedup="true">
    <plan:subject>
      <plan:variable name="s" column-index="0" static-type="NONE">
     </plan:variable>
    </plan:subject>
    <plan:predicate>
     <plan:variable name="p" column-index="1" static-type="NONE">
     </plan:variable>
    </plan:predicate>
    <plan:object>
     <plan:variable name="o" column-index="2" static-type="NONE">
     </plan:variable>
    </plan:object>
   </plan:triple-index>
  </plan:project>
 </plan:select>
</plan:plan>
```

If you run this query, the variables and the values are projected in three columns (s, p, o):

```
[{"s":"<http://example.com/ns/directory#jp>","p":"<http://example.com/
ns/person#firstName>","o":"\"John-Paul\""}]
```

Here is an example of a more complicated SPARQL `SELECT` query, which includes prefixes:

```
sem:sparql-plan("
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX prod:   <http://example.com/products/>
PREFIX ex:     <http://example.com/>

SELECT ?product
FROM <http://marklogic.com/semantics/products/>
WHERE
  {
    ?product  rdf:type  ex:Shirt ;
              ex:color  'blue' }")
```

Here is the output for this query execution plan. The intro specifies the namespace and the type of plan:

```
<plan:plan xmlns:plan="http://marklogic.com/plan">
 <plan:select>
```

The first section identifies the order of projected values, in this case just one "`product`".

```
  <plan:project order="order(0 ASC)">
   <plan:variable name="product" column-index="0" static-type="NONE">
    </plan:variable>
```

This section describes what sort of hash join order.

```
   <plan:hash-join order="order(0 ASC)">
    <plan:hash left="0" right="0" operator="=">
     </plan:hash>
```

Here is the projected order of triple elements–object, predicate, subject (o, p, s)–first for the triple for product of type "Shirt":

```
<plan:triple-index order="order(0 ASC)" permutation="OPS"
dedup="true">
  <plan:subject>
    <plan:variable name="product" column-index="0" static-type="NONE">
    </plan:variable>
  </plan:subject>
  <plan:predicate>
    <plan:iri name="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
static-type="NONE">
    </plan:iri>
  </plan:predicate>
  <plan:object>
    <plan:iri name="http://example.com/Shirt" static-type="NONE">
    </plan:iri>
  </plan:object>
</plan:triple-index>
```

And for the product with the color value “blue”:

```
<plan:triple-index order="order(0 ASC)" permutation="OPS"
dedup="true">
  <plan:subject>
    <plan:variable name="product" column-index="0" static-type="NONE">
    </plan:variable>
  </plan:subject>
  <plan:predicate>
    <plan:iri name="http://example.com/color" static-type="NONE">
    </plan:iri>
  </plan:predicate>
  <plan:object>
    <plan:value datatype="http://www.w3.org/2001/XMLSchema#string"
value="blue">
    </plan:value>
  </plan:object>
</plan:triple-index>
```

And then the close of the plan:

```
    </plan:hash-join>
   </plan:project>
  </plan:select>
</plan:plan>
```

For more about query execution plans, see Execution Plan in the *SQL Data Modeling Guide*.

15.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at http://help.marklogic.com to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the Support Handbook for instructions on registering support contacts and on working with the MarkLogic Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at http://developer.marklogic.com. For technical questions, we encourage you to ask your question on Stack Overflow.

16.0 Copyright

MarkLogic Server 9.0 and supporting products.
Last updated: November 18, 2019

COPYRIGHT

© 2019 MarkLogic Corporation. All rights reserved.
This technology is protected by U.S. Patent No. 7,127,469B2, U.S. Patent No. 7,171,404B2, U.S. Patent No. 7,756,858 B2, and U.S. Patent No 7,962,474 B2, US 8,892,599, and US 8,935,267.

The MarkLogic software is protected by United States and international copyright laws, and incorporates certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers set forth below.

For all copyright notices, including third-party copyright notices, see the Combined Product Notices for your version of MarkLogic.