W3C Recommendation

SPARQL Query Language for RDF

W3C Recommendation 15 January 2008

New Version Available: SPARQL 1.1 (Document Status Update, 26 March 2013)

The SPARQL Working Group has produced a W3C Recommendation for a new version of SPARQL which adds features to this 2008 version. Please see SPARQL 1.1 Overview for an introduction to SPARQL 1.1 and a guide to the SPARQL 1.1 document set.

This version:
http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/
Latest version:
http://www.w3.org/TR/rdf-sparql-query/
Previous version:
http://www.w3.org/TR/2007/PR-rdf-sparql-query-20071112/
Editors:
Eric Prud'hommeaux, W3C <eric@w3.org>
Andy Seaborne, Hewlett-Packard Laboratories, Bristol <andy.seaborne@hp.com>

Please refer to the **errata** for this document, which may include some normative corrections.

See also **translations**.

Abstract

RDF is a directed, labeled graph data format for representing information in the Web. This specification defines the syntax and semantics of the SPARQL query language for RDF. SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports extensible value testing and constraining queries by source RDF graph. The results of SPARQL queries can be results sets or RDF graphs.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This is a W3C Recommendation.

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

Comments on this document should be sent to public-rdf-dawg-comments@w3.org, a mailing list with a public archive. Questions and comments about SPARQL that are not related to this specification, including extensions and features, can be discussed on the mailing list public-sparql-dev@w3.org, (public archive).

This document was produced by the RDF Data Access Working Group, which is part of the W3C Semantic Web Activity. The first release of this document as a Working Draft was 12 October 2004 and the Working Group has addressed a number of comments received and issues since then. Two changes have been made and logged since the publication of the November 2007 Proposed Recommendation.

The Working Group's SPARQL Query Language For RDF Implementation Report demonstrates that the goals for interoperable implementations, set in the June 2007 Candidate Recommendation , were achieved.

The Data Access Working Group has postponed 12 issues, including aggregate functions, and an update language.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

Appendices

---

1 Introduction

RDF is a directed, labeled graph data format for representing information in the Web. RDF is often used to represent, among other things, personal information, social networks, metadata about digital artifacts, as well as to provide a means of integration over disparate sources of information. This specification defines the syntax and semantics of the SPARQL query language for RDF.

The SPARQL query language for RDF is designed to meet the use cases and requirements identified by the RDF Data Access Working Group in *RDF Data Access Use Cases and Requirements* [UCNR].

The SPARQL query language is closely related to the following specifications:

- The SPARQL Protocol for RDF [SPROT] specification defines the remote protocol for issuing SPARQL queries and receiving the results.
- The SPARQL Query Results XML Format [RESULTS] specification defines an XML document format for representing the results of SPARQL SELECT and ASK queries.

1.1 Document Outline

Unless otherwise noted in the section heading, all sections and appendices in this document are normative.

This section of the document, section 1, introduces the SPARQL query language specification. It presents the organization of this specification document and the conventions used throughout the specification.

Section 2 of the specification introduces the SPARQL query language itself via a series of example queries and query results. Section 3 continues the introduction of the SPARQL query language with more examples that demonstrate SPARQL's ability to express constraints on the RDF terms that appear in a query's results.

Section 4 presents details of the SPARQL query language's syntax. It is a companion to the full grammar of the language and defines how grammatical constructs represent IRIs, blank nodes, literals, and variables. Section 4 also defines the meaning of several grammatical constructs that serve as syntactic sugar for more verbose expressions.

Section 5 introduces basic graph patterns and group graph patterns, the building blocks from which more complex SPARQL query patterns are constructed. Sections 6, 7, and 8 present constructs that combine SPARQL graph patterns into larger graph patterns. In particular, Section 6 introduces the ability to make portions of a query optional; Section 7 introduces the ability to express the disjunction of alternative graph patterns; and Section 8 introduces the ability to constrain portions of a query to particular source graphs. Section 8 also presents SPARQL's mechanism for defining the source graphs for a query.

Section 9 defines the constructs that affect the solutions of a query by ordering, slicing, projecting, limiting, and removing duplicates from a sequence of solutions.

Section 10 defines the four types of SPARQL queries that produce results in different forms.

Section 11 defines SPARQL's extensible value testing framework. It also presents the functions and operators that can be used to constrain the values that appear in a query's results.

Section 12 is a formal definition of the evaluation of SPARQL graph patterns and solution modifiers.

Appendix A contains the normative definition of the SPARQL query language's syntax, as given by a grammar expressed in EBNF notation.

1.2 Document Conventions

1.2.1 Namespaces

In this document, examples assume the following namespace prefix bindings unless otherwise stated:

Prefix	IRI
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:	http://www.w3.org/2000/01/rdf-schema#
xsd:	http://www.w3.org/2001/XMLSchema#
fn:	http://www.w3.org/2005/xpath-functions#

1.2.2 Data Descriptions

This document uses the Turtle [TURTLE] data format to show each triple explicitly. Turtle allows IRIs to be abbreviated with prefixes:

```
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix :     <http://example.org/book/> .
:book1  dc:title  "SPARQL Tutorial" .
```

1.2.3 Result Descriptions

Result sets are illustrated in tabular form.

x	y	z
"Alice"	<http://example/a>	

A 'binding' is a pair (variable, RDF term). In this result set, there are three variables: `x`, `y` and `z` (shown as column headers). Each solution is shown as one row in the body of the table. Here, there is a single solution, in which variable `x` is bound to `"Alice"`, variable `y` is bound to `<http://example/a>`, and variable `z` is not bound to an RDF term. Variables are not required to be bound in a solution.

1.2.4 Terminology

The SPARQL language includes IRIs, a subset of RDF URI References that omits spaces. Note that all IRIs in SPARQL queries are absolute; they may or may not include a fragment identifier [RFC3987, section 3.1]. IRIs include URIs [RFC3986] and URLs. The abbreviated forms (relative IRIs and prefixed names) in the SPARQL syntax are resolved to produce absolute IRIs.

The following terms are defined in RDF Concepts and Abstract Syntax [CONCEPTS] and used in SPARQL:

- IRI (corresponds to the Concepts and Abstract Syntax term "`RDF URI reference`")
- literal
- lexical form
- plain literal
- language tag
- typed literal
- datatype IRI (corresponds to the Concepts and Abstract Syntax term "`datatype URI`")
- blank node

2 Making Simple Queries (Informative)

Most forms of SPARQL query contain a set of triple patterns called a *basic graph pattern*. Triple patterns are like RDF triples except that each of the subject, predicate and object may be a variable. A basic 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 RDF graph equivalent to the subgraph.

2.1 Writing a Simple Query

The example below shows a SPARQL query to find the title of a book from the given data graph. The query consists of two parts: the `SELECT` clause identifies the variables to appear in the query results, and the `WHERE` clause provides the basic graph pattern to match against the data graph. The basic graph pattern in this example consists of a single triple pattern with a single variable (`?title`) in the object position.

Data:

```
<http://example.org/book/book1> <http://purl.org/dc/elements/1.1/title> "SPARQL Tutorial" .
```

Query:

```
SELECT ?title
WHERE
{
  <http://example.org/book/book1> <http://purl.org/dc/elements/1.1/title> ?title .
}
```

This query, on the data above, has one solution:

Query Result:

title
"SPARQL Tutorial"

2.2 Multiple Matches

The result of a query is a solution sequence, corresponding to the ways in which the query's graph pattern matches the data. There may be zero, one or multiple solutions to a query.

Data:

```
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name   "Johnny Lee Outlaw" .
_:a  foaf:mbox   <mailto:jlow@example.com> .
_:b  foaf:name   "Peter Goodguy" .
_:b  foaf:mbox   <mailto:peter@example.org> .
_:c  foaf:mbox   <mailto:carol@example.org> .
```

Query:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE
  { ?x foaf:name ?name .
    ?x foaf:mbox ?mbox }
```

Query Result:

name	mbox
"Johnny Lee Outlaw"	<mailto:jlow@example.com>
"Peter Goodguy"	<mailto:peter@example.org>

Each solution gives one way in which the selected variables can be bound to RDF terms so that the query pattern matches the data. The result set gives all the possible solutions. In the above example, the following two subsets of the data provided the two matches.

```
 _:a foaf:name  "Johnny Lee Outlaw" .
 _:a foaf:box   <mailto:jlow@example.com> .
```

```
 _:b foaf:name  "Peter Goodguy" .
 _:b foaf:box   <mailto:peter@example.org> .
```

This is a basic graph pattern match; all the variables used in the query pattern must be bound in every solution.

2.3 Matching RDF Literals

The data below contains three RDF literals:

```
@prefix dt:   <http://example.org/datatype#> .
@prefix ns:   <http://example.org/ns#> .
@prefix :     <http://example.org/ns#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

:x   ns:p     "cat"@en .
:y   ns:p     "42"^^xsd:integer .
:z   ns:p     "abc"^^dt:specialDatatype .
```

Note that, in Turtle, `"cat"@en` is an RDF literal with a lexical form "cat" and a language `en`; `"42"^^xsd:integer` is a typed literal with the datatype `http://www.w3.org/2001/XMLSchema#integer`; and `"abc"^^dt:specialDatatype` is a typed literal with the datatype `http://example.org/datatype#specialDatatype`.

This RDF data is the data graph for the query examples in sections 2.3.1–2.3.3.

2.3.1 Matching Literals with Language Tags

Language tags in SPARQL are expressed using `@` and the language tag, as defined in Best Common Practice 47 [BCP47].

This following query has no solution because `"cat"` is not the same RDF literal as `"cat"@en`:

```
SELECT ?v WHERE { ?v ?p "cat" }
```

v

but the query below will find a solution where variable `v` is bound to `:x` because the language tag is specified and matches the given data:

```
SELECT ?v WHERE { ?v ?p "cat"@en }
```

v
<http://example.org/ns#x>

2.3.2 Matching Literals with Numeric Types

Integers in a SPARQL query indicate an RDF typed literal with the datatype `xsd:integer`. For example: `42` is a shortened form of `"42"^^<http://www.w3.org/2001/XMLSchema#integer>`.

The pattern in the following query has a solution with variable `v` bound to `:y`.

```
SELECT ?v WHERE { ?v ?p 42 }
```

v
<http://example.org/ns#y>

Section 4.1.2 defines SPARQL shortened forms for `xsd:float` and `xsd:double`.

2.3.3 Matching Literals with Arbitrary Datatypes

The following query has a solution with variable `v` bound to `:z`. The query processor does not have to have any understanding of the values in the space of the datatype. Because the lexical form and datatype IRI both match, the literal matches.

```
SELECT ?v WHERE { ?v ?p "abc"^^<http://example.org/datatype#specialDatatype> }
```

v
<http://example.org/ns#z>

2.4 Blank Node Labels in Query Results

Query results can contain blank nodes. Blank nodes in the example result sets in this document are written in the form "_:" followed by a blank node label.

Blank node labels are scoped to a result set (as defined in "SPARQL Query Results XML Format") or, for the `CONSTRUCT` query form, the result graph. Use of the same label within a result set indicates the same blank node.

Data:

```
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name   "Alice" .
_:b  foaf:name   "Bob" .
```

Query:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?x ?name
WHERE  { ?x foaf:name ?name }
```

x	name
_:c	"Alice"
_:d	"Bob"

The results above could equally be given with different blank node labels because the labels in the results only indicate whether RDF terms in the solutions are the same or different.

x	name
_:r	"Alice"
_:s	"Bob"

These two results have the same information: the blank nodes used to match the query are different in the two solutions. There need not be any relation between a label `_:a` in the result set and a blank node in the data graph with the same label.

An application writer should not expect blank node labels in a query to refer to a particular blank node in the data.

2.5 Building RDF Graphs

SPARQL has several query forms. The `SELECT` query form returns variable bindings. The `CONSTRUCT` query form returns an RDF graph. The graph is built based on a template which is used to generate RDF triples based on the results of matching the graph pattern of the query.

Data:

```
@prefix org:    <http://example.com/ns#> .

_:a  org:employeeName   "Alice" .
_:a  org:employeeId     12345 .

_:b  org:employeeName   "Bob" .
_:b  org:employeeId     67890 .
```

Query:

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

CONSTRUCT { ?x foaf:name ?name }
WHERE  { ?x org:employeeName ?name }
```

Results:

```
@prefix org: <http://example.com/ns#> .

_:x foaf:name "Alice" .
_:y foaf:name "Bob" .
```

which can be serialized in RDF/XML as:

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    >
  <rdf:Description>
    <foaf:name>Alice</foaf:name>
  </rdf:Description>
  <rdf:Description>
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
</rdf:RDF>
```

3 RDF Term Constraints (Informative)

Graph pattern matching produces a solution sequence, where each solution has a set of bindings of variables to RDF terms. SPARQL `FILTER`s restrict solutions to those for which the filter expression evaluates to `TRUE`.

This section provides an informal introduction to SPARQL `FILTER`s; their semantics are defined in Section 11. Testing Values. The examples in this section share one input graph:

Data:

```
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix :     <http://example.org/book/> .
@prefix ns:   <http://example.org/ns#> .

:book1  dc:title  "SPARQL Tutorial" .
:book1  ns:price  42 .
:book2  dc:title  "The Semantic Web" .
:book2  ns:price  23 .
```

3.1 Restricting the Values of Strings

SPARQL `FILTER` functions like `regex` can test RDF literals. `regex` matches only plain literals with no language tag.

`regex` can be used to match the lexical forms of other literals by using the str function.

Query:

```
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
SELECT  ?title
WHERE   { ?x dc:title ?title
          FILTER regex(?title, "^SPARQL")
        }
```

Query Result:

title
"SPARQL Tutorial"

Regular expression matches may be made case-insensitive with the "`i`" flag.

Query:

```
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
SELECT  ?title
WHERE   { ?x dc:title ?title
          FILTER regex(?title, "web", "i" )
        }
```

Query Result:

title
"The Semantic Web"

The regular expression language is defined by XQuery 1.0 and XPath 2.0 Functions and Operators and is based on XML Schema Regular Expressions.

3.2 Restricting Numeric Values

SPARQL `FILTER`s can restrict on arithmetic expressions.

Query:

```
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
PREFIX  ns:  <http://example.org/ns#>
SELECT  ?title ?price
WHERE   { ?x ns:price ?price .
          FILTER (?price < 30.5)
          ?x dc:title ?title . }
```

Query Result:

title	price
"The Semantic Web"	23

By constraining the `price` variable, only `:book2` matches the query because only `:book2` has a price less than `30.5`, as the filter condition requires.

3.3 Other Term Constraints

In addition to numeric types, SPARQL supports types `xsd:string`, `xsd:boolean` and `xsd:dateTime` (see 11.1 Operand Data Types). 11.3 Operator Mapping lists a set of test functions, including `BOUND`, `isLITERAL` and `langMATCHES` and accessors, including `STR`, `LANG` and `DATATYPE`. 11.5 Constructor Functions lists a set of XML Schema constructor functions that are in the SPARQL language to cast values from one type to another.

4 SPARQL Syntax

This section covers the syntax used by SPARQL for RDF terms and triple patterns. The full grammar is given in appendix A.

4.1 RDF Term Syntax

4.1.1 Syntax for IRIs

The IRIref production designates the set of IRIs [RFC3987]; IRIs are a generalization of URIs [RFC3986] and are fully

compatible with URIs and URLs. The PrefixedName production designates a prefixed name. The mapping from a prefixed name to an IRI is described below. IRI references (relative or absolute IRIs) are designated by the IRI_REF production, where the '<' and '>' delimiters do not form part of the IRI reference. Relative IRIs match the irelative-ref reference in section 2.2 ABNF for IRI References and IRIs in [RFC3987] and are resolved to IRIs as described below.

Grammar rules:

[67]	*IRIref*	::=	IRI_REF | PrefixedName
[68]	*PrefixedName*	::=	PNAME_LN | PNAME_NS
[69]	*BlankNode*	::=	BLANK_NODE_LABEL | ANON
[70]	*IRI_REF*	::=	'<' ([^<>"{}|^`\]-[#x00-#x20])* '>'
[71]	*PNAME_NS*	::=	PN_PREFIX? ':'
[72]	*PNAME_LN*	::=	PNAME_NS PN_LOCAL

The set of RDF terms defined in RDF Concepts and Abstract Syntax includes RDF URI references while SPARQL terms include IRIs. RDF URI references containing "<", ">", '"' (double quote), space, "{", "}", "|", "\", "^", and "`" are not IRIs. The behavior of a SPARQL query against RDF statements composed of such RDF URI references is not defined.

Prefixed names

The `PREFIX` keyword associates a prefix label with an IRI. A prefixed name is a prefix label and a local part, separated by a colon ":". A prefixed name is mapped to an IRI by concatenating the IRI associated with the prefix and the local part. The prefix label or the local part may be empty. Note that SPARQL local names allow leading digits while XML local names do not.

Relative IRIs

Relative IRIs are combined with base IRIs as per Uniform Resource Identifier (URI): Generic Syntax [RFC3986] using only the basic algorithm in Section 5.2 . Neither Syntax-Based Normalization nor Scheme-Based Normalization (described in sections 6.2.2 and 6.2.3 of RFC3986) are performed. Characters additionally allowed in IRI references are treated in the same way that unreserved characters are treated in URI references, per section 6.5 of Internationalized Resource Identifiers (IRIs) [RFC3987].

The `BASE` keyword defines the Base IRI used to resolve relative IRIs per RFC3986 section 5.1.1, "Base URI Embedded in Content". Section 5.1.2, "Base URI from the Encapsulating Entity" defines how the Base IRI may come from an encapsulating document, such as a SOAP envelope with an xml:base directive or a mime multipart document with a Content-Location header. The "Retrieval URI" identified in 5.1.3, Base "URI from the Retrieval URI", is the URL from which a particular SPARQL query was retrieved. If none of the above specifies the Base URI, the default Base URI (section 5.1.4, "Default Base URI") is used.

The following fragments are some of the different ways to write the same IRI:

```
<http://example.org/book/book1>
```

```
BASE <http://example.org/book/>
<book1>
```

```
PREFIX book: <http://example.org/book/>
book:book1
```

4.1.2 Syntax for Literals

The general syntax for literals is a string (enclosed in either double quotes, `"..."`, or single quotes, `'...'`), with either an optional language tag (introduced by `@`) or an optional datatype IRI or prefixed name (introduced by `^^`).

As a convenience, integers can be written directly (without quotation marks and an explicit datatype IRI) and are interpreted as typed literals of datatype `xsd:integer`; decimal numbers for which there is '.' in the number but no exponent are interpreted as `xsd:decimal`; and numbers with exponents are interpreted as `xsd:double`. Values of type `xsd:boolean` can also be written as `true` or `false`.

To facilitate writing literal values which themselves contain quotation marks or which are long and contain newline characters, SPARQL provides an additional quoting construct in which literals are enclosed in three single- or double-quotation marks.

Examples of literal syntax in SPARQL include:

- `"chat"`

- `'chat'@fr` with language tag "fr"
- `"xyz"^^<http://example.org/ns/userDatatype>`
- `"abc"^^appNS:appDataType`
- `'''The librarian said, "Perhaps you would enjoy 'War and Peace'."'''`
- `1`, which is the same as `"1"^^xsd:integer`
- `1.3`, which is the same as `"1.3"^^xsd:decimal`
- `1.300`, which is the same as `"1.300"^^xsd:decimal`
- `1.0e6`, which is the same as `"1.0e6"^^xsd:double`
- `true`, which is the same as `"true"^^xsd:boolean`
- `false`, which is the same as `"false"^^xsd:boolean`

Grammar rules:

[60]	*RDFLiteral*	::=	String (LANGTAG | ('^^' IRIref))?
[61]	*NumericLiteral*	::=	NumericLiteralUnsigned | NumericLiteralPositive | NumericLiteralNegative
[62]	*NumericLiteralUnsigned*	::=	INTEGER | DECIMAL | DOUBLE
[63]	*NumericLiteralPositive*	::=	INTEGER_POSITIVE | DECIMAL_POSITIVE | DOUBLE_POSITIVE
[64]	*NumericLiteralNegative*	::=	INTEGER_NEGATIVE | DECIMAL_NEGATIVE | DOUBLE_NEGATIVE
[65]	*BooleanLiteral*	::=	'true' | 'false'
[66]	*String*	::=	STRING_LITERAL1 | STRING_LITERAL2 | STRING_LITERAL_LONG1 | STRING_LITERAL_LONG2
[76]	*LANGTAG*	::=	'@' [a-zA-Z]+ ('-' [a-zA-Z0-9]+)*
[77]	*INTEGER*	::=	[0-9]+
[78]	*DECIMAL*	::=	[0-9]+ '.' [0-9]* | '.' [0-9]+
[79]	*DOUBLE*	::=	[0-9]+ '.' [0-9]* EXPONENT | '.' ([0-9])+ EXPONENT | ([0-9])+ EXPONENT
[80]	*INTEGER_POSITIVE*	::=	'+' INTEGER
[81]	*DECIMAL_POSITIVE*	::=	'+' DECIMAL
[82]	*DOUBLE_POSITIVE*	::=	'+' DOUBLE
[83]	*INTEGER_NEGATIVE*	::=	'-' INTEGER
[84]	*DECIMAL_NEGATIVE*	::=	'-' DECIMAL
[85]	*DOUBLE_NEGATIVE*	::=	'-' DOUBLE
[86]	*EXPONENT*	::=	[eE] [+-]? [0-9]+
[87]	*STRING_LITERAL1*	::=	"'" (([^#x27#x5C#xA#xD]) | ECHAR)* "'"
[88]	*STRING_LITERAL2*	::=	'"' (([^#x22#x5C#xA#xD]) | ECHAR)* '"'

Tokens matching the productions INTEGER, DECIMAL, DOUBLE and BooleanLiteral are equivalent to a typed literal with the lexical value of the token and the corresponding datatype (`xsd:integer`, `xsd:decimal`, `xsd:double`, `xsd:boolean`).

4.1.3 Syntax for Query Variables

Query variables in SPARQL queries have global scope; use of a given variable name anywhere in a query identifies the same variable. Variables are prefixed by either "?" or "$"; the "?" or "$" is not part of the variable name. In a query, `$abc` and `?abc` identify the same variable. The possible names for variables are given in the SPARQL grammar.

Grammar rules:

[44]	*Var*	::=	VAR1 | VAR2
[74]	*VAR1*	::=	'?' VARNAME
[75]	*VAR2*	::=	'$' VARNAME
[97]	*VARNAME*	::=	(PN_CHARS_U | [0-9]) (PN_CHARS_U | [0-9] | #x00B7 | [#x0300-#x036F] | [#x203F-#x2040])*

4.1.4 Syntax for Blank Nodes

Blank nodes in graph patterns act as non-distinguished variables, not as references to specific blank nodes in the data being queried.

Blank nodes are indicated by either the label form, such as "_:abc", or the abbreviated form "[]". A blank node that is used in only one place in the query syntax can be indicated with []. A unique blank node will be used to form the triple pattern. Blank node labels are written as "_:abc" for a blank node with label "abc". The same blank node label cannot be used in two different basic graph patterns in the same query.

The [:p :v] construct can be used in triple patterns. It creates a blank node label which is used as the subject of all contained predicate-object pairs. The created blank node can also be used in further triple patterns in the subject and object positions.

The following two forms

```
[ :p "v" ] .
```

```
[] :p "v" .
```

allocate a unique blank node label (here "b57") and are equivalent to writing:

```
_:b57 :p "v" .
```

This allocated blank node label can be used as the subject or object of further triple patterns. For example, as a subject:

```
[ :p "v" ] :q "w" .
```

which is equivalent to the two triples:

```
_:b57 :p "v" .
_:b57 :q "w" .
```

and as an object:

```
:x :q [ :p "v" ] .
```

which is equivalent to the two triples:

```
:x  :q _:b57 .
_:b57 :p "v" .
```

Abbreviated blank node syntax can be combined with other abbreviations for common subjects and common predicates.

```
  [ foaf:name  ?name ;
    foaf:mbox  <mailto:alice@example.org> ]
```

This is the same as writing the following basic graph pattern for some uniquely allocated blank node label, "b18":

```
  _:b18  foaf:name  ?name .
  _:b18  foaf:mbox  <mailto:alice@example.org> .
```

Grammar rules:

[39]	*BlankNodePropertyList*	::=	'['PropertyListNotEmpty']'
[69]	*BlankNode*	::=	BLANK_NODE_LABEL | ANON
[73]	*BLANK_NODE_LABEL*	::=	'_:' PN_LOCAL
[94]	*ANON*	::=	'[' WS* ']'

4.2 Syntax for Triple Patterns

Triple Patterns are written as a whitespace-separated list of a subject, predicate and object; there are abbreviated ways of writing some common triple pattern constructs.

The following examples express the same query:

```
PREFIX  dc: <http://purl.org/dc/elements/1.1/>
SELECT  ?title
WHERE   { <http://example.org/book/book1> dc:title ?title }
```

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

SELECT  $title
WHERE   { :book1  dc:title  $title }
```

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

SELECT  $title
WHERE   { <book1>  dc:title  ?title }
```

Grammar rules:

[32]	*TriplesSameSubject*	::=	VarOrTerm PropertyListNotEmpty | TriplesNode PropertyList
[33]	*PropertyListNotEmpty*	::=	Verb ObjectList (';' (Verb ObjectList)?)*
[34]	*PropertyList*	::=	PropertyListNotEmpty?
[35]	*ObjectList*	::=	Object (',' Object)*
[37]	*Verb*	::=	VarOrIRIref | 'a'

4.2.1 Predicate-Object Lists

Triple patterns with a common subject can be written so that the subject is only written once and is used for more than one triple pattern by employing the ";" notation.

```
    ?x  foaf:name  ?name ;
        foaf:mbox  ?mbox .
```

This is the same as writing the triple patterns:

```
    ?x  foaf:name  ?name .
    ?x  foaf:mbox  ?mbox .
```

4.2.2 Object Lists

If triple patterns share both subject and predicate, the objects may be separated by ",".

```
    ?x foaf:nick  "Alice" , "Alice_" .
```

is the same as writing the triple patterns:

```
   ?x  foaf:nick  "Alice" .
   ?x  foaf:nick  "Alice_" .
```

Object lists can be combined with predicate-object lists:

```
   ?x  foaf:name ?name ; foaf:nick  "Alice" , "Alice_" .
```

is equivalent to:

```
   ?x  foaf:name  ?name .
   ?x  foaf:nick  "Alice" .
   ?x  foaf:nick  "Alice_" .
```

4.2.3 RDF Collections

RDF collections can be written in triple patterns using the syntax "(element1 element2 ...)". The form "`()`" is an alternative for the IRI `http://www.w3.org/1999/02/22-rdf-syntax-ns#nil`. When used with collection elements, such as `(1 ?x 3 4)`, triple patterns with blank nodes are allocated for the collection. The blank node at the head of the collection can be used as a subject or object in other triple patterns. The blank nodes allocated by the collection syntax do not occur elsewhere in the query.

```
(1 ?x 3 4) :p "w" .
```

is syntactic sugar for (noting that `b0`, `b1`, `b2` and `b3` do not occur anywhere else in the query):

```
    _:b0  rdf:first  1 ;
          rdf:rest   _:b1 .
    _:b1  rdf:first  ?x ;
          rdf:rest   _:b2 .
    _:b2  rdf:first  3 ;
          rdf:rest   _:b3 .
    _:b3  rdf:first  4 ;
          rdf:rest   rdf:nil .
    _:b0  :p         "w" .
```

RDF collections can be nested and can involve other syntactic forms:

```
(1 [:p :q] ( 2 ) ) .
```

is syntactic sugar for:

```
    _:b0  rdf:first  1 ;
          rdf:rest   _:b1 .
    _:b1  rdf:first  _:b2 .
    _:b2  :p         :q .
    _:b1  rdf:rest   _:b3 .
    _:b3  rdf:first  _:b4 .
    _:b4  rdf:first  2 ;
          rdf:rest   rdf:nil .
    _:b3  rdf:rest   rdf:nil .
```

Grammar rules:

[40]	*Collection*	::=	'(' GraphNode+ ')'
[92]	*NIL*	::=	'(' WS* ')'

4.2.4 rdf:type

The keyword "`a`" can be used as a predicate in a triple pattern and is an alternative for the IRI `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`. This keyword is case-sensitive.

```
  ?x  a  :Class1 .
  [ a :appClass ] :p "v" .
```

is syntactic sugar for:

```
  ?x    rdf:type  :Class1 .
  _:b0  rdf:type  :appClass .
  _:b0  :p        "v" .
```

5 Graph Patterns

SPARQL is based around graph pattern matching. More complex graph patterns can be formed by combining smaller patterns in various ways:

- Basic Graph Patterns, where a set of triple patterns must match
- Group Graph Pattern, where a set of graph patterns must all match
- Optional Graph patterns, where additional patterns may extend the solution
- Alternative Graph Pattern, where two or more possible patterns are tried
- Patterns on Named Graphs, where patterns are matched against named graphs

In this section we describe the two forms that combine patterns by conjunction: basic graph patterns, which combine triples patterns, and group graph patterns, which combine all other graph patterns.

The outer-most graph pattern in a query is called the query pattern. It is grammatically identified by `GroupGraphPattern` in

[13]	*WhereClause*	::=	'WHERE'? GroupGraphPattern

5.1 Basic Graph Patterns

Basic graph patterns are sets of triple patterns. SPARQL graph pattern matching is defined in terms of combining the results from matching basic graph patterns.

A sequence of triple patterns interrupted by a filter comprises a single basic graph pattern. Any graph pattern terminates a basic graph pattern.

5.1.1 Blank Node Labels

When using blank nodes of the form `_:abc`, labels for blank nodes are scoped to the basic graph pattern. A label can be used in only a single basic graph pattern in any query.

5.1.2 Extending Basic Graph Pattern Matching

SPARQL is defined for matching RDF graphs with simple entailment. SPARQL can be extended to other forms of entailment given certain conditions as described below.

5.2 Group Graph Patterns

In a SPARQL query string, a group graph pattern is delimited with braces: `{}`. For example, this query's query pattern is a group graph pattern of one basic graph pattern.

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE  {
          ?x foaf:name ?name .
          ?x foaf:mbox ?mbox .
       }
```

The same solutions would be obtained from a query that grouped the triple patterns into two basic graph patterns. For example, the query below has a different structure but would yield the same solutions as the previous query:

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE  { { ?x foaf:name ?name . }
         { ?x foaf:mbox ?mbox . }
       }
```

Grammar rules:

[20]	*GroupGraphPattern*	::=	'{' TriplesBlock? ((GraphPatternNotTriples | Filter) '.'? TriplesBlock?)* '}'
[21]	*TriplesBlock*	::=	TriplesSameSubject ('.' TriplesBlock?)?
[22]	*GraphPatternNotTriples*	::=	OptionalGraphPattern | GroupOrUnionGraphPattern | GraphGraphPattern

5.2.1 Empty Group Pattern

The group pattern:

```
{ }
```

matches any graph (including the empty graph) with one solution that does not bind any variables. For example:

```
SELECT ?x
WHERE {}
```

matches with one solution in which variable `x` is not bound.

5.2.2 Scope of Filters

A constraint, expressed by the keyword `FILTER`, is a restriction on solutions over the whole group in which the filter appears. The following patterns all have the same solutions:

```
{  ?x foaf:name ?name .
   ?x foaf:mbox ?mbox .
   FILTER regex(?name, "Smith")
}
```

```
{  FILTER regex(?name, "Smith")
   ?x foaf:name ?name .
   ?x foaf:mbox ?mbox .
}
```

```
{  ?x foaf:name ?name .
   FILTER regex(?name, "Smith")
   ?x foaf:mbox ?mbox .
}
```

5.2.3 Group Graph Pattern Examples

```
 {
   ?x foaf:name ?name .
   ?x foaf:mbox ?mbox .
 }
```

is a group of one basic graph pattern and that basic graph pattern consists of two triple patterns.

```
 {
   ?x foaf:name ?name . FILTER regex(?name, "Smith")
   ?x foaf:mbox ?mbox .
 }
```

is a group of one basic graph pattern and a filter, and that basic graph pattern consists of two triple patterns; the filter does not break the basic graph pattern into two basic graph patterns.

```
{
  ?x foaf:name ?name .
  {}
  ?x foaf:mbox ?mbox .
}
```

is a group of three elements, a basic graph pattern of one triple pattern, an empty group, and another basic graph pattern of one triple pattern.

6 Including Optional Values

Basic graph patterns allow applications to make queries where the entire query pattern must match for there to be a solution. For every solution of a query containing only group graph patterns with at least one basic graph pattern, every variable is bound to an RDF Term in a solution. However, regular, complete structures cannot be assumed in all RDF graphs. It is useful to be able to have queries that allow information to be added to the solution where the information is available, but do not reject the solution because some part of the query pattern does not match. Optional matching provides this facility: if the optional part does not match, it creates no bindings but does not eliminate the solution.

6.1 Optional Pattern Matching

Optional parts of the graph pattern may be specified syntactically with the OPTIONAL keyword applied to a graph pattern:

```
pattern OPTIONAL { pattern }
```

The syntactic form:

```
{ OPTIONAL { pattern } }
```

is equivalent to:

```
{ { } OPTIONAL { pattern } }
```

Grammar rule:

[23]	*OptionalGraphPattern*	::=	'OPTIONAL' GroupGraphPattern

The `OPTIONAL` keyword is left-associative :

```
pattern OPTIONAL { pattern } OPTIONAL { pattern }
```

is the same as:

```
{ pattern OPTIONAL { pattern } } OPTIONAL { pattern }
```

In an optional match, either the optional graph pattern matches a graph, thereby defining and adding bindings to one or more solutions, or it leaves a solution unchanged without adding any additional bindings.

Data:

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

_:a  rdf:type        foaf:Person .
_:a  foaf:name       "Alice" .
_:a  foaf:mbox       <mailto:alice@example.com> .
_:a  foaf:mbox       <mailto:alice@work.example> .

_:b  rdf:type        foaf:Person .
_:b  foaf:name       "Bob" .
```

Query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE  { ?x foaf:name  ?name .
         OPTIONAL { ?x  foaf:mbox  ?mbox }
       }
```

With the data above, the query result is:

```
```

name	mbox
"Alice"	<mailto:alice@example.com>
"Alice"	<mailto:alice@work.example>
"Bob"	

There is no value of `mbox` in the solution where the name is `"Bob"`.

This query finds the names of people in the data. If there is a triple with predicate `mbox` and the same subject, a solution will contain the object of that triple as well. In this example, only a single triple pattern is given in the optional match part of the query but, in general, the optional part may be any graph pattern. The entire optional graph pattern must match for the optional graph pattern to affect the query solution.

6.2 Constraints in Optional Pattern Matching

Constraints can be given in an optional graph pattern. For example:

```
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix :     <http://example.org/book/> .
@prefix ns:   <http://example.org/ns#> .

:book1  dc:title  "SPARQL Tutorial" .
:book1  ns:price  42 .
:book2  dc:title  "The Semantic Web" .
:book2  ns:price  23 .
```

```
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
PREFIX  ns:  <http://example.org/ns#>
SELECT  ?title ?price
WHERE   { ?x dc:title ?title .
          OPTIONAL { ?x ns:price ?price . FILTER (?price < 30) }
        }
```

title	price
"SPARQL Tutorial"	
"The Semantic Web"	23

No price appears for the book with title "SPARQL Tutorial" because the optional graph pattern did not lead to a solution involving the variable "`price`".

6.3 Multiple Optional Graph Patterns

Graph patterns are defined recursively. A graph pattern may have zero or more optional graph patterns, and any part of a query pattern may have an optional part. In this example, there are two optional graph patterns.

Data:

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Alice" .
_:a  foaf:homepage   <http://work.example.org/alice/> .

_:b  foaf:name       "Bob" .
_:b  foaf:mbox       <mailto:bob@work.example> .
```

Query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox ?hpage
WHERE  { ?x foaf:name  ?name .
         OPTIONAL { ?x foaf:mbox ?mbox } .
         OPTIONAL { ?x foaf:homepage ?hpage }
       }
```

Query result:

name	mbox	hpage
"Alice"		<http://work.example.org/alice/>
"Bob"	<mailto:bob@work.example>	

7 Matching Alternatives

SPARQL provides a means of combining graph patterns so that one of several alternative graph patterns may match. If more than one of the alternatives matches, all the possible pattern solutions are found.

Pattern alternatives are syntactically specified with the `UNION` keyword.

Data:

```
@prefix dc10:  <http://purl.org/dc/elements/1.0/> .
@prefix dc11:  <http://purl.org/dc/elements/1.1/> .

_:a  dc10:title     "SPARQL Query Language Tutorial" .
_:a  dc10:creator   "Alice" .

_:b  dc11:title     "SPARQL Protocol Tutorial" .
_:b  dc11:creator   "Bob" .

_:c  dc10:title     "SPARQL" .
_:c  dc11:title     "SPARQL (updated)" .
```

Query:

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX dc11:  <http://purl.org/dc/elements/1.1/>

SELECT ?title
WHERE  { { ?book dc10:title  ?title } UNION { ?book dc11:title  ?title } }
```

Query result:

title
"SPARQL Protocol Tutorial"
"SPARQL"
"SPARQL (updated)"
"SPARQL Query Language Tutorial"

This query finds titles of the books in the data, whether the title is recorded using Dublin Core properties from version 1.0 or version 1.1. To determine exactly how the information was recorded, a query could use different variables for the two alternatives:

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX dc11:  <http://purl.org/dc/elements/1.1/>

SELECT ?x ?y
WHERE  { { ?book dc10:title ?x } UNION { ?book dc11:title  ?y } }
```

x	y
	"SPARQL (updated)"
	"SPARQL Protocol Tutorial"
"SPARQL"	
"SPARQL Query Language Tutorial"	

This will return results with the variable `x` bound for solutions from the left branch of the `UNION`, and `y` bound for the solutions from the right branch. If neither part of the `UNION` pattern matched, then the graph pattern would not match.

The `UNION` pattern combines graph patterns; each alternative possibility can contain more than one triple pattern:

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX dc11:  <http://purl.org/dc/elements/1.1/>

SELECT ?title ?author
WHERE  { { ?book dc10:title ?title .  ?book dc10:creator ?author }
         UNION
         { ?book dc11:title ?title .  ?book dc11:creator ?author }
       }
```

author	title
"Alice"	"SPARQL Protocol Tutorial"
"Bob"	"SPARQL Query Language Tutorial"

This query will only match a book if it has both a title and creator predicate from the same version of Dublin Core.

Grammar rule:

[25]	*GroupOrUnionGraphPattern*	::=	GroupGraphPattern ('UNION' GroupGraphPattern)*

8 RDF Dataset

The RDF data model expresses information as graphs consisting of triples with subject, predicate and object. Many RDF data stores hold multiple RDF graphs and record information about each graph, allowing an application to make queries that involve information from more than one graph.

A SPARQL query is executed against an *RDF Dataset* which represents a collection of graphs. An RDF Dataset comprises one graph, the default graph, which does not have a name, and zero or more named graphs, where each named graph is identified by an IRI. A SPARQL query can match different parts of the query pattern against different graphs as described in section 8.3 Querying the Dataset.

An RDF Dataset may contain zero named graphs; an RDF Dataset always contains one default graph. A query does not need to involve matching the default graph; the query can just involve matching named graphs.

The graph that is used for matching a basic graph pattern is the *active graph*. In the previous sections, all queries have been shown executed against a single graph, the default graph of an RDF dataset as the active graph. The `GRAPH` keyword is used to make the active graph one of all of the named graphs in the dataset for part of the query.

8.1 Examples of RDF Datasets

The definition of RDF Dataset does not restrict the relationships of named and default graphs. Information can be repeated in different graphs; relationships between graphs can be exposed. Two useful arrangements are:

- to have information in the default graph that includes provenance information about the named graphs
- to include the information in the named graphs in the default graph as well.

Example 1:

```
# Default graph
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://example.org/bob>    dc:publisher  "Bob" .
<http://example.org/alice>  dc:publisher  "Alice" .
```

```
# Named graph: http://example.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Bob" .
_:a foaf:mbox <mailto:bob@oldcorp.example.org> .
```

```
# Named graph: http://example.org/alice
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Alice" .
_:a foaf:mbox <mailto:alice@work.example.org> .
```

In this example, the default graph contains the names of the publishers of two named graphs. The triples in the named graphs are not visible in the default graph in this example.

Example 2:

RDF data can be combined by the RDF merge [RDF-MT] of graphs. One possible arrangement of graphs in an RDF Dataset is to have the default graph be the RDF merge of some or all of the information in the named graphs.

In this next example, the named graphs contain the same triples as before. The RDF dataset includes an RDF merge of the named graphs in the default graph, re-labeling blank nodes to keep them distinct.

```
# Default graph
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:x foaf:name "Bob" .
_:x foaf:mbox <mailto:bob@oldcorp.example.org> .

_:y foaf:name "Alice" .
_:y foaf:mbox <mailto:alice@work.example.org> .
```

```
# Named graph: http://example.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Bob" .
_:a foaf:mbox <mailto:bob@oldcorp.example.org> .
```

```
# Named graph: http://example.org/alice
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Alice" .
_:a foaf:mbox <mailto:alice@work.example> .
```

In an RDF merge, blank nodes in the merged graph are not shared with blank nodes from the graphs being merged.

8.2 Specifying RDF Datasets

A SPARQL query may specify the dataset to be used for matching by using the `FROM` clause and the `FROM NAMED` clause to describe the RDF dataset. If a query provides such a dataset description, then it is used in place of any

dataset that the query service would use if no dataset description is provided in a query. The RDF dataset may also be specified in a SPARQL protocol request, in which case the protocol description overrides any description in the query itself. A query service may refuse a query request if the dataset description is not acceptable to the service.

The FROM and FROM NAMED keywords allow a query to specify an RDF dataset by reference; they indicate that the dataset should include graphs that are obtained from representations of the resources identified by the given IRIs (i.e. the absolute form of the given IRI references). The dataset resulting from a number of FROM and FROM NAMED clauses is:

- a default graph consisting of the RDF merge of the graphs referred to in the FROM clauses, and
- a set of (IRI, graph) pairs, one from each FROM NAMED clause.

If there is no FROM clause, but there is one or more FROM NAMED clauses, then the dataset includes an empty graph for the default graph.

Grammar rules:

[9]	*DatasetClause*	::=	'FROM' (DefaultGraphClause | NamedGraphClause)
[10]	*DefaultGraphClause*	::=	SourceSelector
[11]	*NamedGraphClause*	::=	'NAMED' SourceSelector
[12]	*SourceSelector*	::=	IRIref

8.2.1 Specifying the Default Graph

Each FROM clause contains an IRI that indicates a graph to be used to form the default graph. This does not put the graph in as a named graph.

In this example, the RDF Dataset contains a single default graph and no named graphs:

```
# Default graph (stored at http://example.org/foaf/aliceFoaf)
@prefix  foaf:  <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name     "Alice" .
_:a  foaf:mbox     <mailto:alice@work.example> .
```

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT  ?name
FROM    <http://example.org/foaf/aliceFoaf>
WHERE   { ?x foaf:name ?name }
```

name
"Alice"

If a query provides more than one FROM clause, providing more than one IRI to indicate the default graph, then the default graph is based on the RDF merge of the graphs obtained from representations of the resources identified by the given IRIs.

8.2.2 Specifying Named Graphs

A query can supply IRIs for the named graphs in the RDF Dataset using the FROM NAMED clause. Each IRI is used to provide one named graph in the RDF Dataset. Using the same IRI in two or more FROM NAMED clauses results in one named graph with that IRI appearing in the dataset.

```
# Graph: http://example.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Bob" .
_:a foaf:mbox <mailto:bob@oldcorp.example.org> .
```

```
# Graph: http://example.org/alice
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Alice" .
_:a foaf:mbox <mailto:alice@work.example> .
```

```
...
FROM NAMED <http://example.org/alice>
FROM NAMED <http://example.org/bob>
...
```

The FROM NAMED syntax suggests that the IRI identifies the corresponding graph, but the relationship between an IRI and a graph in an RDF dataset is indirect. The IRI identifies a resource, and the resource is represented by a graph (or, more precisely: by a document that serializes a graph). For further details see [WEBARCH].

8.2.3 Combining FROM and FROM NAMED

The FROM clause and FROM NAMED clause can be used in the same query.

```
# Default graph (stored at http://example.org/dft.ttl)
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://example.org/bob>    dc:publisher  "Bob Hacker" .
<http://example.org/alice>  dc:publisher  "Alice Hacker" .
```

```
# Named graph: http://example.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Bob" .
_:a foaf:mbox <mailto:bob@oldcorp.example.org> .
```

```
# Named graph: http://example.org/alice
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Alice" .
_:a foaf:mbox <mailto:alice@work.example.org> .
```

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

The RDF Dataset for this query contains a default graph and two named graphs. The GRAPH keyword is described below.

The actions required to construct the dataset are not determined by the dataset description alone. If an IRI is given twice in a dataset description, either by using two FROM clauses, or a FROM clause and a FROM NAMED clause, then it does not assume that exactly one or exactly two attempts are made to obtain an RDF graph associated with the IRI. Therefore, no assumptions can be made about blank node identity in triples obtained from the two occurrences in the dataset description. In general, no assumptions can be made about the equivalence of the graphs.

8.3 Querying the Dataset

When querying a collection of graphs, the GRAPH keyword is used to match patterns against named graphs. GRAPH can provide an IRI to select one graph or use a variable which will range over the IRI of all the named graphs in the query's RDF dataset.

The use of GRAPH changes the active graph for matching basic graph patterns within part of the query. Outside the use of GRAPH, the default graph is matched by basic graph patterns.

The following two graphs will be used in examples:

```
# Named graph: http://example.org/foaf/aliceFoaf
@prefix  foaf:     <http://xmlns.com/foaf/0.1/> .
@prefix  rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix  rdfs:     <http://www.w3.org/2000/01/rdf-schema#> .

_:a  foaf:name     "Alice" .
_:a  foaf:mbox     <mailto:alice@work.example> .
_:a  foaf:knows    _:b .

_:b  foaf:name     "Bob" .
_:b  foaf:mbox     <mailto:bob@work.example> .
_:b  foaf:nick     "Bobby" .
_:b  rdfs:seeAlso  <http://example.org/foaf/bobFoaf> .

<http://example.org/foaf/bobFoaf>
     rdf:type      foaf:PersonalProfileDocument .
```

```
# Named graph: http://example.org/foaf/bobFoaf
@prefix  foaf:     <http://xmlns.com/foaf/0.1/> .
@prefix  rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix  rdfs:     <http://www.w3.org/2000/01/rdf-schema#> .

_:z  foaf:mbox     <mailto:bob@work.example> .
_:z  rdfs:seeAlso  <http://example.org/foaf/bobFoaf> .
_:z  foaf:nick     "Robert" .

<http://example.org/foaf/bobFoaf>
     rdf:type      foaf:PersonalProfileDocument .
```

Grammar rule:

[24]	*GraphGraphPattern*	::=	'GRAPH' VarOrIRIref GroupGraphPattern

8.3.1 Accessing Graph Names

The query below matches the graph pattern against each of the named graphs in the dataset and forms solutions which have the `src` variable bound to IRIs of the graph being matched. The graph pattern is matched with the active graph being each of the named graphs in the dataset.

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

SELECT ?src ?bobNick
FROM NAMED <http://example.org/foaf/aliceFoaf>
FROM NAMED <http://example.org/foaf/bobFoaf>
WHERE
  {
    GRAPH ?src
    { ?x foaf:mbox <mailto:bob@work.example> .
      ?x foaf:nick ?bobNick
    }
  }
```

The query result gives the name of the graphs where the information was found and the value for Bob's nick:

src	bobNick
<http://example.org/foaf/aliceFoaf>	"Bobby"
<http://example.org/foaf/bobFoaf>	"Robert"

8.3.2 Restricting by Graph IRI

The query can restrict the matching applied to a specific graph by supplying the graph IRI. This sets the active graph to the graph named by the IRI. This query looks for Bob's nick as given in the graph `http://example.org/foaf/bobFoaf`.

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

SELECT ?nick
FROM NAMED <http://example.org/foaf/aliceFoaf>
FROM NAMED <http://example.org/foaf/bobFoaf>
WHERE
  {
     GRAPH data:bobFoaf {
         ?x foaf:mbox <mailto:bob@work.example> .
         ?x foaf:nick ?nick }
  }
```

which yields a single solution:

nick
"Robert"

8.3.3 Restricting Possible Graph IRIs

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.

The query below uses the graph with IRI `http://example.org/foaf/aliceFoaf` to find the profile document for Bob; it then matches another pattern against that graph. The pattern in the second `GRAPH` clause finds the blank node (variable `w`) for the person with the same mail box (given by variable `mbox`) as found in the first `GRAPH` clause (variable `whom`), because the blank node used to match for variable `whom` from Alice's FOAF file is not the same as the blank node in the profile document (they are in different graphs).

```
PREFIX  data:  <http://example.org/foaf/>
PREFIX  foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX  rdfs:  <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?mbox ?nick ?ppd
FROM NAMED <http://example.org/foaf/aliceFoaf>
FROM NAMED <http://example.org/foaf/bobFoaf>
WHERE
{
  GRAPH data:aliceFoaf
  {
    ?alice foaf:mbox <mailto:alice@work.example> ;
           foaf:knows ?whom .
    ?whom  foaf:mbox ?mbox ;
           rdfs:seeAlso ?ppd .
    ?ppd  a foaf:PersonalProfileDocument .
  } .
  GRAPH ?ppd
  {
      ?w foaf:mbox ?mbox ;
         foaf:nick ?nick
  }
}
```

mbox	nick	ppd
<mailto:bob@work.example>	"Robert"	<http://example.org/foaf/bobFoaf>

Any triple in Alice's FOAF file giving Bob's `nick` is not used to provide a nick for Bob because the pattern involving variable `nick` is restricted by `ppd` to a particular Personal Profile Document.

8.3.4 Named and Default Graphs

Query patterns can involve both the default graph and the named graphs. In this example, an aggregator has read in a Web resource on two different occasions. Each time a graph is read into the aggregator, it is given an IRI by the local system. The graphs are nearly the same but the email address for "Bob" has changed.

In this example, the default graph is being used to record the provenance information and the RDF data actually read is kept in two separate graphs, each of which is given a different IRI by the system. The RDF dataset consists of two named graphs and the information about them.

RDF Dataset:

```
# Default graph
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix g:  <tag:example.org,2005-06-06:> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

g:graph1 dc:publisher "Bob" .
g:graph1 dc:date "2004-12-06"^^xsd:date .

g:graph2 dc:publisher "Bob" .
g:graph2 dc:date "2005-01-10"^^xsd:date .
```

```
# Graph: locally allocated IRI: tag:example.org,2005-06-06:graph1
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Alice" .
_:a foaf:mbox <mailto:alice@work.example> .

_:b foaf:name "Bob" .
_:b foaf:mbox <mailto:bob@oldcorp.example.org> .
```

```
# Graph: locally allocated IRI: tag:example.org,2005-06-06:graph2
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Alice" .
_:a foaf:mbox <mailto:alice@work.example> .

_:b foaf:name "Bob" .
_:b foaf:mbox <mailto:bob@newcorp.example.org> .
```

This query finds email addresses, detailing the name of the person and the date the information was discovered.

```
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 }
  }
```

The results show that the email address for "Bob" has changed.

name	mbox	date
"Bob"	<mailto:bob@oldcorp.example.org>	"2004-12-06"^^xsd:date
"Bob"	<mailto:bob@newcorp.example.org>	"2005-01-10"^^xsd:date

The IRI for the date datatype has been abbreviated in the results for clarity.

9 Solution Sequences and Modifiers

Query patterns generate an unordered collection of solutions, each solution being a partial function from variables to RDF terms. These solutions are then treated as a sequence (a solution sequence), initially in no specific order; any sequence modifiers are then applied to create another sequence. Finally, this latter sequence is used to generate one of the results of a SPARQL query form.

A **solution sequence modifier** is one of:

- Order modifier: put the solutions in order
- Projection modifier: choose certain variables
- Distinct modifier: ensure solutions in the sequence are unique
- Reduced modifier: permit elimination of some non-unique solutions
- Offset modifier: control where the solutions start from in the overall sequence of solutions
- Limit modifier: restrict the number of solutions

Modifiers are applied in the order given by the list above.

Grammar rules:

[5]	*SelectQuery*	::=	'SELECT' ('DISTINCT' | 'REDUCED')? (Var+ | '*') DatasetClause* WhereClause SolutionModifier
[14]	*SolutionModifier*	::=	OrderClause? LimitOffsetClauses?
[15]	*LimitOffsetClauses*	::=	(LimitClause OffsetClause? | OffsetClause LimitClause?)
[16]	*OrderClause*	::=	'ORDER' 'BY' OrderCondition+

9.1 ORDER BY

The `ORDER BY` clause establishes the order of a solution sequence.

Following the `ORDER BY` clause is a sequence of order comparators, composed of an expression and an optional order modifier (either `ASC()` or `DESC()`). Each ordering comparator is either ascending (indicated by the `ASC()` modifier or by no modifier) or descending (indicated by the `DESC()` modifier).

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

SELECT ?name
WHERE { ?x foaf:name ?name }
ORDER BY ?name
```

```
PREFIX     :    <http://example.org/ns#>
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX xsd:     <http://www.w3.org/2001/XMLSchema#>

SELECT ?name
WHERE { ?x foaf:name ?name ; :empId ?emp }
ORDER BY DESC(?emp)
```

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

SELECT ?name
WHERE { ?x foaf:name ?name ; :empId ?emp }
ORDER BY ?name DESC(?emp)
```

The "<" operator (see the Operator Mapping and 11.3.1 Operator Extensibility) defines the relative order of pairs of `numerics`, `simple literals`, `xsd:strings`, `xsd:booleans` and `xsd:dateTimes`. Pairs of IRIs are ordered by comparing them as `simple literals`.

SPARQL also fixes an order between some kinds of RDF terms that would not otherwise be ordered:

1. (Lowest) no value assigned to the variable or expression in this solution.
2. Blank nodes
3. IRIs
4. RDF literals

A plain literal is lower than an RDF literal with type `xsd:string` of the same lexical form.

SPARQL does not define a total ordering of all possible RDF terms. Here are a few examples of pairs of terms for which the relative order is undefined:

- "a" and "a"@en_gb (a simple literal and a literal with a language tag)
- "a"@en_gb and "b"@en_gb (two literals with language tags)
- "a" and "a"^^xsd:string (a simple literal and an xsd:string)
- "a" and "1"^^xsd:integer (a simple literal and a literal with a supported data type)
- "1"^^my:integer and "2"^^my:integer (two unsupported data types)
- "1"^^xsd:integer and "2"^^my:integer (a supported data type and an unsupported data type)

This list of variable bindings is in ascending order:

RDF Term	Reason
	Unbound results sort earliest.
_:z	Blank nodes follow unbound.
_:a	There is no relative ordering of blank nodes.
<http://script.example/Latin>	IRIs follow blank nodes.
<http://script.example/Кириллица>	The character in the 23rd position, "K", has a unicode codepoint 0x41A, which is higher than 0x4C ("L").
<http://script.example/漢字>	The character in the 23rd position, "漢", has a unicode codepoint 0x6F22, which is higher than 0x41A ("K").
"http://script.example/Latin"	Simple literals follow IRIs.
"http://script.example/Latin"^^xsd:string	xsd:strings follow simple literals.

The ascending order of two solutions with respect to an ordering comparator is established by substituting the solution bindings into the expressions and comparing them with the "<" operator. The descending order is the reverse of the ascending order.

The relative order of two solutions is the relative order of the two solutions with respect to the first ordering comparator in the sequence. For solutions where the substitutions of the solution bindings produce the same RDF term, the order is the relative order of the two solutions with respect to the next ordering comparator. The relative order of two solutions is undefined if no order expression evaluated for the two solutions produces distinct RDF terms.

Ordering a sequence of solutions always results in a sequence with the same number of solutions in it.

Using `ORDER BY` on a solution sequence for a `CONSTRUCT` or `DESCRIBE` query has no direct effect because only `SELECT` returns a sequence of results. Used in combination with `LIMIT` and `OFFSET`, `ORDER BY` can be used to return results generated from a different slice of the solution sequence. An `ASK` query does not include `ORDER BY`, `LIMIT` or `OFFSET`.

Grammar rules:

[16]	*OrderClause*	::=	'ORDER' 'BY' OrderCondition+
[17]	*OrderCondition*	::=	(('ASC' | 'DESC') BrackettedExpression) | (Constraint | Var)
[18]	*LimitClause*	::=	'LIMIT' INTEGER
[19]	*OffsetClause*	::=	'OFFSET' INTEGER

9.2 Projection

The solution sequence can be transformed into one involving only a subset of the variables. For each solution in the sequence, a new solution is formed using a specified selection of the variables using the SELECT query form.

The following example shows a query to extract just the names of people described in an RDF graph using FOAF properties.

```
@prefix foaf:        <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Alice" .
_:a  foaf:mbox       <mailto:alice@work.example> .

_:b  foaf:name       "Bob" .
_:b  foaf:mbox       <mailto:bob@work.example> .
```

```
PREFIX foaf:       <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE
 { ?x foaf:name ?name }
```

name
"Bob"

"Alice"

9.3 Duplicate Solutions

A solution sequence with no DISTINCT or REDUCED query modifier will preserve duplicate solutions.

```
@prefix  foaf:  <http://xmlns.com/foaf/0.1/> .

_:x    foaf:name   "Alice" .
_:x    foaf:mbox   <mailto:alice@example.com> .

_:y    foaf:name   "Alice" .
_:y    foaf:mbox   <mailto:asmith@example.com> .

_:z    foaf:name   "Alice" .
_:z    foaf:mbox   <mailto:alice.smith@example.com> .
```

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT ?name WHERE { ?x foaf:name ?name }
```

name
"Alice"
"Alice"
"Alice"

The modifiers DISTINCT and REDUCED affect whether duplicates are included in the query results.

9.3.1 DISTINCT

The DISTINCT solution modifier eliminates duplicate solutions. Specifically, each solution that binds the same variables to the same RDF terms as another solution is eliminated from the solution set.

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?name WHERE { ?x foaf:name ?name }
```

name
"Alice"

Note that, per the order of solution sequence modifiers, duplicates are eliminated before either limit or offset is applied.

9.3.2 REDUCED

While the DISTINCT modifier ensures that duplicate solutions are eliminated from the solution set, REDUCED simply permits them to be eliminated. The cardinality of any set of variable bindings in an REDUCED solution set is at least one and not more than the cardinality of the solution set with no DISTINCT or REDUCED modifier. For example, using the data above, the query

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT REDUCED ?name WHERE { ?x foaf:name ?name }
```

may have one, two (shown here) or three solutions:

name
"Alice"
"Alice"

9.4 OFFSET

OFFSET causes the solutions generated to start after the specified number of solutions. An OFFSET of zero has no effect.

Using LIMIT and OFFSET to select different subsets of the query solutions will not be useful unless the order is made predictable by using ORDER BY.

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

SELECT  ?name
WHERE   { ?x foaf:name ?name }
ORDER BY ?name
LIMIT   5
OFFSET  10
```

9.5 LIMIT

The `LIMIT` clause puts an upper bound on the number of solutions returned. If the number of actual solutions is greater than the limit, then at most the limit number of solutions will be returned.

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

SELECT ?name
WHERE { ?x foaf:name ?name }
LIMIT 20
```

A `LIMIT` of 0 would cause no results to be returned. A limit may not be negative.

10 Query Forms

SPARQL has four query forms. These query forms use the solutions from pattern matching to form result sets or RDF graphs. The query forms are:

SELECT
: Returns all, or a subset of, the variables bound in a query pattern match.

CONSTRUCT
: Returns an RDF graph constructed by substituting variables in a set of triple templates.

ASK
: Returns a boolean indicating whether a query pattern matches or not.

DESCRIBE
: Returns an RDF graph that describes the resources found.

The SPARQL Variable Binding Results XML Format can be used to serialize the result set from a `SELECT` query or the boolean result of an `ASK` query.

10.1 SELECT

The SELECT form of results returns variables and their bindings directly. The syntax `SELECT *` is an abbreviation that selects all of the variables in a query.

```
@prefix  foaf:  <http://xmlns.com/foaf/0.1/> .

_:a    foaf:name   "Alice" .
_:a    foaf:knows  _:b .
_:a    foaf:knows  _:c .

_:b    foaf:name   "Bob" .

_:c    foaf:name   "Clare" .
_:c    foaf:nick   "CT" .
```

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT ?nameX ?nameY ?nickY
WHERE
  { ?x foaf:knows ?y ;
       foaf:name ?nameX .
    ?y foaf:name ?nameY .
    OPTIONAL { ?y foaf:nick ?nickY }
  }
```

nameX	nameY	nickY
"Alice"	"Bob"	
"Alice"	"Clare"	"CT"

Result sets can be accessed by a local API but also can be serialized into either XML or an RDF graph. An XML format is described in SPARQL Query Results XML Format, and gives for this example:

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="nameX"/>
    <variable name="nameY"/>
    <variable name="nickY"/>
  </head>
  <results>
    <result>
      <binding name="nameX">
        <literal>Alice</literal>
      </binding>
      <binding name="nameY">
        <literal>Bob</literal>
      </binding>
   </result>
    <result>
      <binding name="nameX">
        <literal>Alice</literal>
      </binding>
      <binding name="nameY">
        <literal>Clare</literal>
      </binding>
      <binding name="nickY">
        <literal>CT</literal>
      </binding>
    </result>
  </results>
</sparql>
```

Grammar rule:

[5]	*SelectQuery*	::=	'SELECT' ('DISTINCT' | 'REDUCED')? (Var+ | '*') DatasetClause* WhereClause SolutionModifier

10.2 CONSTRUCT

The CONSTRUCT query form returns a single RDF graph specified by a graph template. The result is an RDF graph formed by taking each query solution in the solution sequence, substituting for the variables in the graph template, and combining the triples into a single RDF graph by set union.

If any such instantiation produces a triple containing an unbound variable or an illegal RDF construct, such as a literal in subject or predicate position, then that triple is not included in the output RDF graph. The graph template can contain triples with no variables (known as ground or explicit triples), and these also appear in the output RDF graph returned by the CONSTRUCT query form.

```
@prefix  foaf:  <http://xmlns.com/foaf/0.1/> .

_:a    foaf:name   "Alice" .
_:a    foaf:mbox   <mailto:alice@example.org> .
```

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX vcard:   <http://www.w3.org/2001/vcard-rdf/3.0#>
CONSTRUCT   { <http://example.org/person#Alice> vcard:FN ?name }
WHERE       { ?x foaf:name ?name }
```

creates vcard properties from the FOAF information:

```
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .

<http://example.org/person#Alice> vcard:FN "Alice" .
```

10.2.1 Templates with Blank Nodes

A template can create an RDF graph containing blank nodes. The blank node labels are scoped to the template for each solution. If the same label occurs twice in a template, then there will be one blank node created for each query solution, but there will be different blank nodes for triples generated by different query solutions.

```
@prefix  foaf:  <http://xmlns.com/foaf/0.1/> .

_:a    foaf:givenname   "Alice" .
_:a    foaf:family_name "Hacker" .

_:b    foaf:firstname   "Bob" .
_:b    foaf:surname     "Hacker" .
```

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX vcard:   <http://www.w3.org/2001/vcard-rdf/3.0#>

CONSTRUCT { ?x  vcard:N _:v .
            _:v vcard:givenName ?gname .
            _:v vcard:familyName ?fname }
WHERE
 {
    { ?x foaf:firstname ?gname } UNION  { ?x foaf:givenname   ?gname } .
    { ?x foaf:surname   ?fname } UNION  { ?x foaf:family_name ?fname } .
 }
```

creates vcard properties corresponding to the FOAF information:

```
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .

_:v1 vcard:N          _:x .
_:x vcard:givenName  "Alice" .
_:x vcard:familyName "Hacker" .

_:v2 vcard:N          _:z .
_:z vcard:givenName  "Bob" .
_:z vcard:familyName "Hacker" .
```

The use of variable `x` in the template, which in this example will be bound to blank nodes with labels `_:a` and `_:b` in the data, causes different blank node labels (`_:v1` and `_:v2`) in the resulting RDF graph.

10.2.2 Accessing Graphs in the RDF Dataset

Using `CONSTRUCT`, it is possible to extract parts or the whole of graphs from the target RDF dataset. This first example returns the graph (if it is in the dataset) with IRI label `http://example.org/aGraph`; otherwise, it returns an empty graph.

```
CONSTRUCT { ?s ?p ?o } WHERE { GRAPH <http://example.org/aGraph> { ?s ?p ?o } . }
```

The access to the graph can be conditional on other information. For example, if the default graph contains metadata about the named graphs in the dataset, then a query like the following one can extract one graph based on information about the named graph:

```
PREFIX  dc: <http://purl.org/dc/elements/1.1/>
PREFIX app: <http://example.org/ns#>
CONSTRUCT { ?s ?p ?o } WHERE
 {
   GRAPH ?g { ?s ?p ?o } .
   { ?g dc:publisher <http://www.w3.org/> } .
   { ?g dc:date ?date } .
   FILTER ( app:customDate(?date) > "2005-02-28T00:00:00Z"^^xsd:dateTime ) .
 }
```

where `app:customDate` identified an extension function to turn the data format into an `xsd:dateTime` RDF term.

Grammar rule:

[6]	*ConstructQuery*	::=	'CONSTRUCT' ConstructTemplate DatasetClause* WhereClause SolutionModifier

10.2.3 Solution Modifiers and CONSTRUCT

The solution modifiers of a query affect the results of a `CONSTRUCT` query. In this example, the output graph from the `CONSTRUCT` template is formed from just two of the solutions from graph pattern matching. The query outputs a graph with the names of the people with the top two sites, rated by hits. The triples in the RDF graph are not ordered.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix site: <http://example.org/stats#> .

_:a foaf:name "Alice" .
_:a site:hits 2349 .

_:b foaf:name "Bob" .
_:b site:hits 105 .

_:c foaf:name "Eve" .
_:c site:hits 181 .
```

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

CONSTRUCT { [] foaf:name ?name }
WHERE
{ [] foaf:name ?name ;
     site:hits ?hits .
}
ORDER BY desc(?hits)
LIMIT 2
```

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:x foaf:name "Alice" .
_:y foaf:name "Eve" .
```

10.3 ASK

Applications can use the `ASK` form to test whether or not a query pattern has a solution. No information is returned about the possible query solutions, just whether or not a solution exists.

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Alice" .
_:a  foaf:homepage   <http://work.example.org/alice/> .

_:b  foaf:name       "Bob" .
_:b  foaf:mbox       <mailto:bob@work.example> .
```

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
ASK  { ?x foaf:name  "Alice" }
```

```
yes
```

The SPARQL Query Results XML Format form of this result set gives:

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head></head>
  <results>
    <boolean>true</boolean>
  </results>
</sparql>
```

On the same data, the following returns no match because Alice's `mbox` is not mentioned.

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
ASK  { ?x foaf:name  "Alice" ;
          foaf:mbox  <mailto:alice@work.example> }
```

```
no
```

Grammar rule:

[8]	*AskQuery*	::=	'ASK' DatasetClause* WhereClause

10.4 DESCRIBE (Informative)

The `DESCRIBE` form returns a single result RDF graph containing RDF data about resources. This data is not prescribed by a SPARQL query, where the query client would need to know the structure of the RDF in the data source, but, instead, is determined by the SPARQL query processor. The query pattern is used to create a result set. The `DESCRIBE` form takes each of the resources identified in a solution, together with any resources directly named by IRI, and assembles a single RDF graph by taking a "description" which can come from any information available including the target RDF Dataset. The description is determined by the query service. The syntax `DESCRIBE *` is an abbreviation that describes all of the variables in a query.

10.4.1 Explicit IRIs

The `DESCRIBE` clause itself can take IRIs to identify the resources. The simplest `DESCRIBE` query is just an IRI in the `DESCRIBE` clause:

```
DESCRIBE <http://example.org/>
```

10.4.2 Identifying Resources

The resources to be described can also be taken from the bindings to a query variable in a result set. This enables description of resources whether they are identified by IRI or by blank node in the dataset:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
DESCRIBE ?x
WHERE    { ?x foaf:mbox <mailto:alice@org> }
```

The property `foaf:mbox` is defined as being an inverse function property in the FOAF vocabulary. If treated as such, this query will return information about at most one person. If, however, the query pattern has multiple solutions, the RDF data for each is the union of all RDF graph descriptions.

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
DESCRIBE ?x
WHERE    { ?x foaf:name "Alice" }
```

More than one IRI or variable can be given:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
DESCRIBE ?x ?y <http://example.org/>
WHERE    {?x foaf:knows ?y}
```

10.4.3 Descriptions of Resources

The RDF returned is determined by the information publisher. It is the useful information the service has about a resource. It may include information about other resources: for example, the RDF data for a book may also include details about the author.

A simple query such as

```
PREFIX ent:  <http://org.example.com/employees#>
DESCRIBE ?x WHERE { ?x ent:employeeId "1234" }
```

might return a description of the employee and some other potentially useful details:

```
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .
@prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0> .
@prefix exOrg:  <http://org.example.com/employees#> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl:    <http://www.w3.org/2002/07/owl#>

_:a     exOrg:employeeId    "1234" ;

        foaf:mbox_sha1sum   "ABCD1234" ;
        vcard:N
          [ vcard:Family       "Smith" ;
            vcard:Given        "John"  ] .

foaf:mbox_sha1sum  rdf:type  owl:InverseFunctionalProperty .
```

which includes the blank node closure for the vcard vocabulary vcard:N. Other possible mechanisms for deciding what information to return include Concise Bounded Descriptions [CBD].

For a vocabulary such as FOAF, where the resources are typically blank nodes, returning sufficient information to identify a node such as the InverseFunctionalProperty `foaf:mbox_sha1sum` as well as information like name and other details recorded would be appropriate. In the example, the match to the WHERE clause was returned, but this is not required.

Grammar rule:

[7]	*DescribeQuery*	::=	'DESCRIBE' (VarOrIRIref+ | '*') DatasetClause* WhereClause? SolutionModifier

11 Testing Values

SPARQL `FILTERs` restrict the solutions of a graph pattern match according to a given expression. Specifically, `FILTERs` eliminate any solutions that, when substituted into the expression, either result in an effective boolean value of `false` or produce an error. Effective boolean values are defined in section 11.2.2 *Effective Boolean Value* and errors are defined in XQuery 1.0: An XML Query Language [XQUERY] section 2.3.1, *Kinds of Errors*. These errors have no affect outside of `FILTER` evaluation.

RDF literals may have a datatype IRI:

```
@prefix a:          <http://www.w3.org/2000/10/annotation-ns#> .
@prefix dc:         <http://purl.org/dc/elements/1.1/> .

_:a   a:annotates   <http://www.w3.org/TR/rdf-sparql-query/> .
_:a   dc:date       "2004-12-31T19:00:00-05:00" .

_:b   a:annotates   <http://www.w3.org/TR/rdf-sparql-query/> .
_:b   dc:date       "2004-12-31T19:01:00-05:00"^^<http://www.w3.org/2001/XMLSchema#dateTime> .
```

The object of the first `dc:date` triple has no type information. The second has the datatype `xsd:dateTime`.

SPARQL expressions are constructed according to the grammar and provide access to functions (named by IRI) and operator functions (invoked by keywords and symbols in the SPARQL grammar). SPARQL operators can be used to compare the values of typed literals:

```
PREFIX a:      <http://www.w3.org/2000/10/annotation-ns#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>

SELECT ?annot
WHERE { ?annot  a:annotates  <http://www.w3.org/TR/rdf-sparql-query/> .
        ?annot  dc:date      ?date .
        FILTER ( ?date > "2005-01-01T00:00:00Z"^^xsd:dateTime ) }
```

The SPARQL operators are listed in section 11.3 and are associated with their productions in the grammar.

In addition, SPARQL provides the ability to invoke arbitrary functions, including a subset of the XPath casting functions, listed in section 11.5. These functions are invoked by name (an IRI) within a SPARQL query. For example:

```
... FILTER ( xsd:dateTime(?date) < xsd:dateTime("2005-01-01T00:00:00Z") ) ...
```

The following typographical conventions are used in this section:

- XPath operators are labeled with the prefix `op:`. XPath operators have no namespace; `op:` is a labeling convention.
- Operators introduced by this specification are indicated with the SPARQLoperator class.

11.1 Operand Data Types

SPARQL functions and operators operate on RDF terms and SPARQL variables. A subset of these functions and operators are taken from the XQuery 1.0 and XPath 2.0 Functions and Operators [FUNCOP] and have XML Schema typed value arguments and return types. RDF `typed literals` passed as arguments to these functions and operators are mapped to XML Schema typed values with a string value of the `lexical form` and an atomic datatype corresponding to the datatype IRI. The returned typed values are mapped back to RDF `typed literals` the same way.

SPARQL has additional operators which operate on specific subsets of RDF terms. When referring to a type, the following terms denote a `typed literal` with the corresponding XML Schema [XSDT] datatype IRI:

- `xsd:integer`
- `xsd:decimal`
- `xsd:float`
- `xsd:double`
- `xsd:string`
- `xsd:boolean`
- `xsd:dateTime`

The following terms identify additional types used in SPARQL value tests:

- numeric denotes `typed literals` with datatypes `xsd:integer`, `xsd:decimal`, `xsd:float`, and `xsd:double`.
- simple literal denotes a `plain literal` with no `language tag`.
- RDF term denotes the types `IRI`, `literal`, and `blank node`.
- variable denotes a SPARQL variable.

The following types are derived from numeric types and are valid arguments to functions and operators taking numeric arguments:

- `xsd:nonPositiveInteger`
- `xsd:negativeInteger`
- `xsd:long`
- `xsd:int`
- `xsd:short`
- `xsd:byte`
- `xsd:nonNegativeInteger`

- xsd:unsignedLong
- xsd:unsignedInt
- xsd:unsignedShort
- xsd:unsignedByte
- xsd:positiveInteger

SPARQL language extensions may treat additional types as being derived from XML schema data types.

11.2 Filter Evaluation

SPARQL provides a subset of the functions and operators defined by XQuery Operator Mapping. XQuery 1.0 section 2.2.3 Expression Processing describes the invocation of XPath functions. The following rules accommodate the differences in the data and execution models between XQuery and SPARQL:

- Unlike XPath/XQuery, SPARQL functions do not process node sequences. When interpreting the semantics of XPath functions, assume that each argument is a sequence of a single node.
- Functions invoked with an argument of the wrong type will produce a type error. Effective boolean value arguments (labeled "xsd:boolean (EBV)" in the operator mapping table below), are coerced to `xsd:boolean` using the EBV rules in section 11.2.2 .
- Apart from BOUND, all functions and operators operate on RDF Terms and will produce a type error if any arguments are unbound.
- Any expression other than logical-or (`||`) or logical-and (`&&`) that encounters an error will produce that error.
- A logical-or that encounters an error on only one branch will return TRUE if the other branch is TRUE and an error if the other branch is FALSE.
- A logical-and that encounters an error on only one branch will return an error if the other branch is TRUE and FALSE if the other branch is FALSE.
- A logical-or or logical-and that encounters errors on both branches will produce *either* of the errors.

The logical-and and logical-or truth table for true (`T`), false (`F`), and error (`E`) is as follows:

A	B	A || B	A && B
T	T	T	T
T	F	T	F
F	T	T	F
F	F	F	F
T	E	T	E
E	T	T	E
F	E	E	F
E	F	E	F
E	E	E	E

11.2.1 Invocation

SPARQL defines a syntax for invoking functions and operators on a list of arguments. These are invoked as follows:

- Argument expressions are evaluated, producing argument values. The order of argument evaluation is not defined.
- Numeric arguments are promoted as necessary to fit the expected types for that function or operator.
- The function or operator is invoked on the argument values.

If any of these steps fails, the invocation generates an error. The effects of errors are defined in Filter Evaluation.

11.2.2 Effective Boolean Value (EBV)

Effective boolean value is used to calculate the arguments to the logical functions logical-and, logical-or, and fn:not, as well as evaluate the result of a `FILTER` expression.

The XQuery Effective Boolean Value rules rely on the definition of XPath's fn:boolean. The following rules reflect the rules for `fn:boolean` applied to the argument types present in SPARQL Queries:

- The EBV of any literal whose type is `xsd:boolean` or numeric is false if the lexical form is not valid for that datatype (e.g. "abc"^^xsd:integer).
- If the argument is a typed literal with a datatype of `xsd:boolean`, the EBV is the value of that argument.
- If the argument is a plain literal or a typed literal with a datatype of `xsd:string`, the EBV is false if the operand value has zero length; otherwise the EBV is true.

- If the argument is a numeric type or a typed literal with a datatype derived from a numeric type, the EBV is false if the operand value is NaN or is numerically equal to zero; otherwise the EBV is true.
- All other arguments, including unbound arguments, produce a type error.

An EBV of `true` is represented as a typed literal with a datatype of `xsd:boolean` and a lexical value of "true"; an EBV of false is represented as a typed literal with a datatype of `xsd:boolean` and a lexical value of "false".

11.3 Operator Mapping

The SPARQL grammar identifies a set of operators (for instance, &&, *, isIRI) used to construct constraints. The following table associates each of these grammatical productions with the appropriate operands and an operator function defined by either XQuery 1.0 and XPath 2.0 Functions and Operators [FUNCOP] or the SPARQL operators specified in section 11.4. When selecting the operator definition for a given set of parameters, the definition with the most specific parameters applies. For instance, when evaluating `xsd:integer = xsd:signedInt`, the definition for `=` with two `numeric` parameters applies, rather than the one with two RDF terms. The table is arranged so that the upper-most viable candiate is the most specific. Operators invoked without appropriate operands result in a type error.

SPARQL follows XPath's scheme for numeric type promotions and subtype substitution for arguments to numeric operators. The XPath Operator Mapping rules for numeric operands (`xsd:integer`, `xsd:decimal`, `xsd:float`, `xsd:double`, and types derived from a numeric type) apply to SPARQL operators as well (see XML Path Language (XPath) 2.0 [XPATH20] for defintions of numeric type promotions and subtype substitution). Some of the operators are associated with nested function expressions, e.g. `fn:not(op:numeric-equal(A, B))`. Note that per the XPath definitions, `fn:not` and `op:numeric-equal` produce an error if their argument is an error.

The collation for `fn:compare` is defined by XPath and identified by `http://www.w3.org/2005/xpath-functions/collation/codepoint`. This collation allows for string comparison based on code point values. Codepoint string equivalence can be tested with RDF term equivalence.

SPARQL Unary Operators

Operator	Type(A)	Function	Result type
XQuery Unary Operators			
! A	xsd:boolean (EBV)	fn:not(A)	xsd:boolean
+ A	numeric	op:numeric-unary-plus(A)	numeric
- A	numeric	op:numeric-unary-minus(A)	numeric
SPARQL Tests, defined in section 11.4			
BOUND(A)	variable	bound(A)	xsd:boolean
isIRI(A) **isURI(A)**	RDF term	isIRI(A)	xsd:boolean
isBLANK(A)	RDF term	isBlank(A)	xsd:boolean
isLITERAL(A)	RDF term	isLiteral(A)	xsd:boolean
SPARQL Accessors, defined in section 11.4			
STR(A)	literal	str(A)	simple literal
STR(A)	IRI	str(A)	simple literal
LANG(A)	literal	lang(A)	simple literal
DATATYPE(A)	typed literal	datatype(A)	IRI
DATATYPE(A)	simple literal	datatype(A)	IRI

SPARQL Binary Operators

Operator	Type(A)	Type(B)	Function	Result type
Logical Connectives, defined in section 11.4				
A || B	xsd:boolean (EBV)	xsd:boolean (EBV)	logical-or(A, B)	xsd:boolean
A && B	xsd:boolean (EBV)	xsd:boolean (EBV)	logical-and(A, B)	xsd:boolean
XPath Tests				
A = B	numeric	numeric	op:numeric-equal(A, B)	xsd:boolean

A = B	simple literal	simple literal	op:numeric-equal(fn:compare(A, B), 0)	xsd:boolean
A = B	xsd:string	xsd:string	op:numeric-equal(fn:compare(STR(A), STR(B)), 0)	xsd:boolean
A = B	xsd:boolean	xsd:boolean	op:boolean-equal(A, B)	xsd:boolean
A = B	xsd:dateTime	xsd:dateTime	op:dateTime-equal(A, B)	xsd:boolean
A != B	numeric	numeric	fn:not(op:numeric-equal(A, B))	xsd:boolean
A != B	simple literal	simple literal	fn:not(op:numeric-equal(fn:compare(A, B), 0))	xsd:boolean
A != B	xsd:string	xsd:string	fn:not(op:numeric-equal(fn:compare(STR(A), STR(B)), 0))	xsd:boolean
A != B	xsd:boolean	xsd:boolean	fn:not(op:boolean-equal(A, B))	xsd:boolean
A != B	xsd:dateTime	xsd:dateTime	fn:not(op:dateTime-equal(A, B))	xsd:boolean
A < B	numeric	numeric	op:numeric-less-than(A, B)	xsd:boolean
A < B	simple literal	simple literal	op:numeric-equal(fn:compare(A, B), -1)	xsd:boolean
A < B	xsd:string	xsd:string	op:numeric-equal(fn:compare(STR(A), STR(B)), -1)	xsd:boolean
A < B	xsd:boolean	xsd:boolean	op:boolean-less-than(A, B)	xsd:boolean
A < B	xsd:dateTime	xsd:dateTime	op:dateTime-less-than(A, B)	xsd:boolean
A > B	numeric	numeric	op:numeric-greater-than(A, B)	xsd:boolean
A > B	simple literal	simple literal	op:numeric-equal(fn:compare(A, B), 1)	xsd:boolean
A > B	xsd:string	xsd:string	op:numeric-equal(fn:compare(STR(A), STR(B)), 1)	xsd:boolean
A > B	xsd:boolean	xsd:boolean	op:boolean-greater-than(A, B)	xsd:boolean
A > B	xsd:dateTime	xsd:dateTime	op:dateTime-greater-than(A, B)	xsd:boolean
A <= B	numeric	numeric	logical-or(op:numeric-less-than(A, B), op:numeric-equal(A, B))	xsd:boolean
A <= B	simple literal	simple literal	fn:not(op:numeric-equal(fn:compare(A, B), 1))	xsd:boolean
A <= B	xsd:string	xsd:string	fn:not(op:numeric-equal(fn:compare(STR(A), STR(B)), 1))	xsd:boolean
A <= B	xsd:boolean	xsd:boolean	fn:not(op:boolean-greater-than(A, B))	xsd:boolean
A <= B	xsd:dateTime	xsd:dateTime	fn:not(op:dateTime-greater-than(A, B))	xsd:boolean
A >= B	numeric	numeric	logical-or(op:numeric-greater-than(A, B), op:numeric-equal(A, B))	xsd:boolean
A >= B	simple literal	simple literal	fn:not(op:numeric-equal(fn:compare(A, B), -1))	xsd:boolean
A >= B	xsd:string	xsd:string	fn:not(op:numeric-equal(fn:compare(STR(A), STR(B)), -1))	xsd:boolean
A >= B	xsd:boolean	xsd:boolean	fn:not(op:boolean-less-than(A, B))	xsd:boolean
A >= B	xsd:dateTime	xsd:dateTime	fn:not(op:dateTime-less-than(A, B))	xsd:boolean
XPath Arithmetic				
A * B	numeric	numeric	op:numeric-multiply(A, B)	numeric
A / B	numeric	numeric	op:numeric-divide(A, B)	numeric; but xsd:decimal if both operands are xsd:integer
A + B	numeric	numeric	op:numeric-add(A, B)	numeric
A - B	numeric	numeric	op:numeric-subtract(A, B)	numeric
SPARQL Tests, defined in section 11.4				

A = B	RDF term	RDF term	RDFterm-equal(A, B)	xsd:boolean
A != B	RDF term	RDF term	fn:not(RDFterm-equal(A, B))	xsd:boolean
sameTERM(A)	RDF term	RDF term	sameTerm(A, B)	xsd:boolean
langMATCHES(A, B)	simple literal	simple literal	langMatches(A, B)	xsd:boolean
REGEX(STRING, PATTERN)	simple literal	simple literal	fn:matches(STRING, PATTERN)	xsd:boolean

SPARQL Trinary Operators

Operator	Type(A)	Type(B)	Type(C)	Function	Result type
SPARQL Tests, defined in section 11.4					
REGEX(STRING, PATTERN, FLAGS)	simple literal	simple literal	simple literal	fn:matches(STRING, PATTERN, FLAGS)	xsd:boolean

xsd:boolean function arguments marked with "(EBV)" are coerced to xsd:boolean by evaluating the effective boolean value of that argument.

11.3.1 Operator Extensibility

SPARQL language extensions may provide additional associations between operators and operator functions; this amounts to adding rows to the table above. No additional operator may yield a result that replaces any result other than a type error in the semantics defined above. The consequence of this rule is that SPARQL extensions will produce *at least* the same solutions as an unextended implementation, and may, for some queries, produce more solutions.

Additional mappings of the '<' operator are expected to control the relative ordering of the operands, specifically, when used in an `ORDER BY` clause.

11.4 Operators Definitions

This section defines the operators introduced by the SPARQL Query language. The examples show the behavior of the operators as invoked by the appropriate grammatical constructs.

11.4.1 bound

```
xsd:boolean   BOUND (variable var)
```

Returns `true` if `var` is bound to a value. Returns false otherwise. Variables with the value NaN or INF are considered bound.

Data:

```
@prefix foaf:        <http://xmlns.com/foaf/0.1/> .
@prefix dc:          <http://purl.org/dc/elements/1.1/> .
@prefix xsd:          <http://www.w3.org/2001/XMLSchema#> .

_:a  foaf:givenName  "Alice".

_:b  foaf:givenName  "Bob" .
_:b  dc:date         "2005-04-04T04:04:04Z"^^xsd:dateTime .
```

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>
PREFIX xsd:   <http://www.w3.org/2001/XMLSchema#>
SELECT ?name
 WHERE { ?x foaf:givenName  ?givenName .
         OPTIONAL { ?x dc:date ?date } .
         FILTER ( bound(?date) ) }
```

Query result:

givenName
"Bob"

One may test that a graph pattern is *not* expressed by specifying an `OPTIONAL` graph pattern that introduces a variable and testing to see that the variable is `not bound`. This is called *Negation as Failure* in logic programming.

This query matches the people with a `name` but *no* expressed `date`:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>
SELECT ?name
 WHERE { ?x foaf:givenName  ?name .
         OPTIONAL { ?x dc:date ?date } .
         FILTER (!bound(?date)) }
```

Query result:

name
"Alice"

Because Bob's `dc:date` was known, `"Bob"` was not a solution to the query.

11.4.2 isIRI

```
 xsd:boolean   ISIRI (RDF term term)
 xsd:boolean   ISURI (RDF term term)
```

Returns `true` if `term` is an IRI. Returns `false` otherwise. `ISURI` is an alternate spelling for the `ISIRI` operator.

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Alice".
_:a  foaf:mbox       <mailto:alice@work.example> .

_:b  foaf:name       "Bob" .
_:b  foaf:mbox       "bob@work.example" .
```

This query matches the people with a `name` and an `mbox` which is an IRI:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
 WHERE { ?x foaf:name  ?name ;
            foaf:mbox  ?mbox .
         FILTER isIRI(?mbox) }
```

Query result:

name	mbox
"Alice"	<mailto:alice@work.example>

11.4.3 isBlank

```
 xsd:boolean   ISBLANK (RDF term term)
```

Returns `true` if `term` is a blank node. Returns `false` otherwise.

```
@prefix a:          <http://www.w3.org/2000/10/annotation-ns#> .
@prefix dc:         <http://purl.org/dc/elements/1.1/> .
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .

_:a   a:annotates   <http://www.w3.org/TR/rdf-sparql-query/> .
_:a   dc:creator    "Alice B. Toeclips" .

_:b   a:annotates   <http://www.w3.org/TR/rdf-sparql-query/> .
_:b   dc:creator    _:c .
_:c   foaf:given    "Bob".
_:c   foaf:family   "Smith".
```

This query matches the people with a `dc:creator` which uses predicates from the FOAF vocabulary to express the name.

```
PREFIX a:      <http://www.w3.org/2000/10/annotation-ns#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>

SELECT ?given ?family
 WHERE { ?annot  a:annotates  <http://www.w3.org/TR/rdf-sparql-query/> .
         ?annot  dc:creator   ?c .
         OPTIONAL { ?c  foaf:given   ?given ; foaf:family  ?family } .
         FILTER isBlank(?c)
       }
```

Query result:

given	family
"Bob"	"Smith"

In this example, there were two objects of `foaf:knows` predicates, but only one (`_:c`) was a blank node.

11.4.4 isLiteral

```
xsd:boolean   ISLITERAL (RDF term term)
```

Returns `true` if `term` is a literal. Returns `false` otherwise.

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Alice".
_:a  foaf:mbox       <mailto:alice@work.example> .

_:b  foaf:name       "Bob" .
_:b  foaf:mbox       "bob@work.example" .
```

This query is similar to the one in 11.4.2 except that is matches the people with a `name` and an `mbox` which is a literal. This could be used to look for erroneous data (`foaf:mbox` should only have an IRI as its object).

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
 WHERE { ?x foaf:name  ?name ;
           foaf:mbox  ?mbox .
         FILTER isLiteral(?mbox) }
```

Query result:

name	mbox
"Bob"	"bob@work.example"

11.4.5 str

```
simple literal   STR (literal ltrl)
simple literal   STR (IRI rsrc)
```

Returns the `lexical form` of `ltrl` (a literal); returns the codepoint representation of `rsrc` (an IRI). This is useful for examining parts of an IRI, for instance, the host-name.

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Alice".
_:a  foaf:mbox       <mailto:alice@work.example> .

_:b  foaf:name       "Bob" .
_:b  foaf:mbox       <mailto:bob@home.example> .
```

This query selects the set of people who use their `work.example` address in their foaf profile:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
 WHERE { ?x foaf:name  ?name ;
           foaf:mbox  ?mbox .
         FILTER regex(str(?mbox), "@work.example") }
```

Query result:

name	mbox
"Alice"	<mailto:alice@work.example>

11.4.6 lang

```
simple literal   LANG (literal ltrl)
```

Returns the `language tag` of `ltrl`, if it has one. It returns `""` if `ltrl` has no `language tag`. Note that the RDF data model does not include literals with an empty `language tag`.

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Robert"@EN.
_:a  foaf:name       "Roberto"@ES.
_:a  foaf:mbox       <mailto:bob@work.example> .
```

This query finds the Spanish `foaf:name` and `foaf:mbox`:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
 WHERE { ?x foaf:name  ?name ;
           foaf:mbox  ?mbox .
         FILTER ( lang(?name) = "ES" ) }
```

Query result:

name	mbox
"Roberto"@ES	<mailto:bob@work.example>

11.4.7 datatype

```
 IRI   DATATYPE (typed literal typedLit)
 IRI   DATATYPE (simple literal simpleLit)
```

Returns the `datatype IRI` of `typedLit`; returns `xsd:string` if the parameter is a simple literal.

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .
@prefix eg:         <http://biometrics.example/ns#> .
@prefix xsd:        <http://www.w3.org/2001/XMLSchema#> .

_:a  foaf:name       "Alice".
_:a  eg:shoeSize     "9.5"^^xsd:float .

_:b  foaf:name       "Bob".
_:b  eg:shoeSize     "42"^^xsd:integer .
```

This query finds the `foaf:name` and `foaf:shoeSize` of everyone with a shoeSize that is an integer:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX eg:   <http://biometrics.example/ns#>
SELECT ?name ?shoeSize
 WHERE { ?x foaf:name  ?name ; eg:shoeSize  ?shoeSize .
         FILTER ( datatype(?shoeSize) = xsd:integer ) }
```

Query result:

name	shoeSize
"Bob"	42

11.4.8 logical-or

```
 xsd:boolean   xsd:boolean left || xsd:boolean right
```

Returns a logical `OR` of `left` and `right`. Note that `logical-or` operates on the effective boolean value of its arguments.

Note: see section 11.2, Filter Evaluation, for the `||` operator's treatment of errors.

11.4.9 logical-and

```
 xsd:boolean   xsd:boolean left && xsd:boolean right
```

Returns a logical `AND` of `left` and `right`. Note that `logical-and` operates on the effective boolean value of its arguments.

Note: see section 11.2, Filter Evaluation, for the `&&` operator's treatment of errors.

11.4.10 RDFterm-equal

```
 xsd:boolean   RDF term term1 = RDF term term2
```

Returns TRUE if `term1` and `term2` are the same RDF term as defined in Resource Description Framework (RDF): Concepts and Abstract Syntax [CONCEPTS]; produces a type error if the arguments are both literal but are not the same RDF term [*]; returns FALSE otherwise. `term1` and `term2` are the same if any of the following is true:

- `term1` and `term2` are equivalent IRIs as defined in 6.4 RDF URI References of [CONCEPTS].
- `term1` and `term2` are equivalent literals as defined in 6.5.1 Literal Equality of [CONCEPTS].
- `term1` and `term2` are the same blank node as described in 6.6 Blank Nodes of [CONCEPTS].

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Alice".
_:a  foaf:mbox       <mailto:alice@work.example> .

_:b  foaf:name       "Ms A.".
_:b  foaf:mbox       <mailto:alice@work.example> .
```

This query finds the people who have multiple `foaf:name` triples:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name1 ?name2
 WHERE { ?x foaf:name  ?name1 ;
            foaf:mbox  ?mbox1 .
         ?y foaf:name  ?name2 ;
            foaf:mbox  ?mbox2 .
         FILTER (?mbox1 = ?mbox2 && ?name1 != ?name2)
       }
```

Query result:

name1	name2
"Alice"	"Ms A."
"Ms A."	"Alice"

In this query for documents that were annotated on New Year's Day (2004 or 2005), the RDF terms are not the same, but have equivalent values:

```
@prefix a:          <http://www.w3.org/2000/10/annotation-ns#> .
@prefix dc:         <http://purl.org/dc/elements/1.1/> .

_:b   a:annotates   <http://www.w3.org/TR/rdf-sparql-query/> .
_:b   dc:date       "2004-12-31T19:00:00-05:00"^^<http://www.w3.org/2001/XMLSchema#dateTime> .
```

```
PREFIX a:      <http://www.w3.org/2000/10/annotation-ns#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>

SELECT ?annotates
WHERE { ?annot  a:annotates  ?annotates .
        ?annot  dc:date      ?date .
        FILTER ( ?date = xsd:dateTime("2005-01-01T00:00:00Z") ) }
```

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

* Invoking RDFterm-equal on two typed literals tests for equivalent values. An extended implementation may have support for additional datatypes. An implementation processing a query that tests for equivalence on unsupported datatypes (and non-identical lexical form and datatype IRI) returns an error, indicating that it was unable to determine whether or not the values are equivalent. For example, an unextended implementation will produce an error when testing either `"iiii"^^my:romanNumeral = "iv"^^my:romanNumeral` or `"iiii"^^my:romanNumeral != "iv"^^my:romanNumeral`.

11.4.11 sameTerm

```
xsd:boolean   SAMETERM (RDF term term1, RDF term term2)
```

Returns TRUE if `term1` and `term2` are the same RDF term as defined in Resource Description Framework (RDF): Concepts and Abstract Syntax [CONCEPTS]; returns FALSE otherwise.

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Alice".
_:a  foaf:mbox       <mailto:alice@work.example> .

_:b  foaf:name       "Ms A.".
_:b  foaf:mbox       <mailto:alice@work.example> .
```

This query finds the people who have multiple `foaf:name` triples:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name1 ?name2
 WHERE { ?x foaf:name  ?name1 ;
            foaf:mbox  ?mbox1 .
         ?y foaf:name  ?name2 ;
            foaf:mbox  ?mbox2 .
         FILTER (sameTerm(?mbox1, ?mbox2) && !sameTerm(?name1, ?name2))
       }
```

Query result:

name1	name2
"Alice"	"Ms A."
"Ms A."	"Alice"

Unlike `RDFterm-equal`, `sameTerm` can be used to test for non-equivalent typed literals with unsupported data types:

```
@prefix :          <http://example.org/WMterms#> .
@prefix t:         <http://example.org/types#> .

_:c1  :label        "Container 1" .
_:c1  :weight       "100"^^t:kilos .
_:c1  :displacement  "100"^^t:liters .

_:c2  :label        "Container 2" .
_:c2  :weight       "100"^^t:kilos .
_:c2  :displacement  "85"^^t:liters .

_:c3  :label        "Container 3" .
_:c3  :weight       "85"^^t:kilos .
_:c3  :displacement  "85"^^t:liters .
```

```
PREFIX  :      <http://example.org/WMterms#>
PREFIX  t:     <http://example.org/types#>

SELECT ?aLabel1 ?bLabel
WHERE { ?a  :label        ?aLabel .
        ?a  :weight       ?aWeight .
        ?a  :displacement ?aDisp .

        ?b  :label        ?bLabel .
        ?b  :weight       ?bWeight .
        ?b  :displacement ?bDisp .

        FILTER ( sameTerm(?aWeight, ?bWeight) && !sameTerm(?aDisp, ?bDisp) }
```

aLabel	bLabel
"Container 1"	"Container 2"
"Container 2"	"Container 1"

The test for boxes with the same weight may also be done with the '=' operator (RDFterm-equal) as the test for `"100"^^t:kilos = "85"^^t:kilos` will result in an error, eliminating that potential solution.

11.4.12 langMatches

```
xsd:boolean   LANGMATCHES (simple literal language-tag, simple literal language-range)
```

Returns `true` if `language-tag` (first argument) matches `language-range` (second argument) per the basic filtering scheme defined in [RFC4647] section 3.3.1. `language-range` is a basic language range per Matching of Language Tags [RFC4647] section 2.1. A `language-range` of "*" matches any non-empty `language-tag` string.

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

_:a  dc:title         "That Seventies Show"@en .
_:a  dc:title         "Cette Série des Années Soixante-dix"@fr .
_:a  dc:title         "Cette Série des Années Septante"@fr-BE .
_:b  dc:title         "Il Buono, il Bruto, il Cattivo" .
```

This query uses `langMatches` and `lang` (described in section 11.2.3.8) to find the French titles for the show known in English as "That Seventies Show":

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?title
 WHERE { ?x dc:title  "That Seventies Show"@en ;
            dc:title  ?title .
         FILTER langMatches( lang(?title), "FR" ) }
```

Query result:

title
"Cette Série des Années Soixante-dix"@fr
"Cette Série des Années Septante"@fr-BE

The idiom `langMatches( lang( ?v ), "*" )` will not match literals without a language tag as `lang( ?v )` will return an empty string, so

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?title
 WHERE { ?x dc:title  ?title .
         FILTER langMatches( lang(?title), "*" ) }
```

will report all of the titles with a language tag:

title
"That Seventies Show"@en
"Cette Série des Années Soixante-dix"@fr
"Cette Série des Années Septante"@fr-BE

11.4.13 regex

```
 xsd:boolean   REGEX (simple literal text, simple literal pattern)
 xsd:boolean   REGEX (simple literal text, simple literal pattern, simple literal flags)
```

Invokes the XPath fn:matches function to match `text` against a regular expression `pattern`. The regular expression language is defined in XQuery 1.0 and XPath 2.0 Functions and Operators section 7.6.1 Regular Expression Syntax [FUNCOP].

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Alice".
_:b  foaf:name       "Bob" .
```

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
 WHERE { ?x foaf:name  ?name
         FILTER regex(?name, "^ali", "i") }
```

Query result:

name
"Alice"

11.5 Constructor Functions

SPARQL imports a subset of the XPath constructor functions defined in XQuery 1.0 and XPath 2.0 Functions and Operators [FUNCOP] in section 17.1 Casting from primitive types to primitive types. SPARQL constructors include all of the XPath constructors for the SPARQL operand data types plus the additional datatypes imposed by the RDF data model. Casting in SPARQL is performed by calling a constructor function for the target type on an operand of the source type.

XPath defines only the casts from one XML Schema datatype to another. The remaining casts are defined as follows:

- Casting an IRI to an `xsd:string` produces a typed literal with a lexical value of the codepoints comprising the IRI, and a datatype of `xsd:string`.
- Casting a simple literal to any XML Schema datatype is defined as the product of casting an `xsd:string` with the string value equal to the lexical value of the literal to the target datatype.

The table below summarizes the casting operations that are always allowed (Y), never allowed (N) and dependent on the lexical value (M). For example, a casting operation from an `xsd:string` (the first row) to an `xsd:float` (the second column) is dependent on the lexical value (M).

bool = xsd:boolean
dbl = xsd:double
flt = xsd:float
dec = xsd:decimal
int = xsd:integer
dT = xsd:dateTime
str = xsd:string
IRI = IRI
ltrl = `simple literal`

From \ To	str	flt	dbl	dec	int	dT	bool
str	Y	M	M	M	M	M	M
flt	Y	Y	Y	M	M	N	Y
dbl	Y	Y	Y	M	M	N	Y
dec	Y	Y	Y	Y	Y	N	Y

From \ To	str	flt	dbl	dec	int	dT	bool
int	Y	Y	Y	Y	Y	N	Y
dT	Y	N	N	N	N	Y	N
bool	Y	Y	Y	Y	Y	N	Y
IRI	Y	N	N	N	N	N	N
ltrl	Y	M	M	M	M	M	M

11.6 Extensible Value Testing

A PrimaryExpression grammar rule can be a call to an extension function named by an IRI. An extension function takes some number of RDF terms as arguments and returns an RDF term. The semantics of these functions are identified by the IRI that identifies the function.

SPARQL queries using extension functions are likely to have limited interoperability.

As an example, consider a function called `func:even`:

```
xsd:boolean   func:even (numeric value)
```

This function would be invoked in a FILTER as such:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX func: <http://example.org/functions#>
SELECT ?name ?id
WHERE { ?x foaf:name  ?name ;
          func:empId   ?id .
        FILTER (func:even(?id)) }
```

For a second example, consider a function `aGeo:distance` that calculates the distance between two points, which is used here to find the places near Grenoble:

```
xsd:double   aGeo:distance (numeric x1, numeric y1, numeric x2, numeric y2)
```

```
PREFIX aGeo: <http://example.org/geo#>

SELECT ?neighbor
WHERE { ?a aGeo:placeName "Grenoble" .
        ?a aGeo:location ?axLoc .
        ?a aGeo:location ?ayLoc .

        ?b aGeo:placeName ?neighbor .
        ?b aGeo:location ?bxLoc .
        ?b aGeo:location ?byLoc .

        FILTER ( aGeo:distance(?axLoc, ?ayLoc, ?bxLoc, ?byLoc) < 10 ) .
      }
```

An extension function might be used to test some application datatype not supported by the core SPARQL specification, it might be a transformation between datatype formats, for example into an XSD dateTime RDF term from another date format.

12 Definition of SPARQL

This section defines the correct behavior for evaluation of graph patterns and solution modifiers, given a query string and an RDF dataset. It does not imply a SPARQL implementation must use the process defined here.

The outcome of executing a SPARQL is defined by a series of steps, starting from the SPARQL query as a string, turning that string into an abstract syntax form, then turning the abstract syntax into a SPARQL abstract query comprising operators from the SPARQL algebra. This abstract query is then evaluated on an RDF dataset.

12.1 Initial Definitions

12.1.1 RDF Terms

SPARQL is defined in terms of IRIs [RFC3987]. IRIs are a subset of RDF URI References that omits spaces.

Definition: RDF Term

Let I be the set of all IRIs.
Let RDF-L be the set of all RDF Literals
Let RDF-B be the set of all blank nodes in RDF graphs

The set of **RDF Terms**, RDF-T, is I union RDF-L union RDF-B.

This definition of **RDF Term** collects together several basic notions from the RDF data model, but updated to refer to IRIs rather than RDF URI references.

12.1.2 RDF Dataset

Definition: RDF Dataset

An RDF dataset is a set:
{ G, (<u_1>, G_1), (<u_2>, G_2), . . . (<u_n>, G_n) }
where G and each G_i are graphs, and each <u_i> is an IRI. Each <u_i> is distinct.

G is called the default graph. (<u_i>, G_i) are called named graphs.

Definition: Active Graph

The **active graph** is the graph from the dataset used for basic graph pattern matching.

12.1.3 Query Variables

Definition: Query Variable

A **query variable** is a member of the set V where V is infinite and disjoint from RDF-T.

12.1.4 Triple Patterns

Definition: Triple Pattern

A **triple pattern** is member of the set:
(RDF-T union V) x (I union V) x (RDF-T union V)

This definition of Triple Pattern includes literal subjects. This has been noted by RDF-core.

> "[The RDF core Working Group] noted that it is aware of no reason why literals should not be subjects and a future WG with a less restrictive charter may extend the syntaxes to allow literals as the subjects of statements."

Because RDF graphs may not contain literal subjects, any SPARQL triple pattern with a literal as subject will fail to match on any RDF graph.

12.1.5 Basic Graph Patterns

Definition: Basic Graph Pattern

A **Basic Graph Pattern** is a set of Triple Patterns.

The empty graph pattern is a basic graph pattern which is the empty set.

12.1.6 Solution Mapping

A solution mapping is a mapping from a set of variables to a set of RDF terms. We use the term 'solution' where it is clear.

Definition: Solution Mapping

A **solution mapping**, μ, is a partial function μ : V -> T.

The domain of μ, dom(μ), is the subset of V where μ is defined.

Definition: Solution Sequence

A **solution sequence** is a list of solutions, possibly unordered.

12.1.7 Solution Sequence Modifiers

Definition: Solution Sequence Modifier

A **solution sequence modifier** is one of:

- Order By modifier: put the solutions in order
- Projection modifier: choose certain variables
- Distinct modifier: ensure solutions in the sequence are unique
- Reduced modifier: permit any non-unique solutions to be eliminated
- Offset modifier: control where the solutions start from in the overall sequence of solutions
- Limit modifier: restrict the number of solutions

12.2 SPARQL Query

This section defines the process of converting graph patterns and solution modifiers in a SPARQL query string into a SPARQL algebra expression.

After parsing a SPARQL query string, and applying the abbreviations for IRIs and triple patterns given in section 4, there is an abstract syntax tree composed of:

Patterns	Modifiers	Query Forms
RDF terms	DISTINCT	SELECT
triple patterns	REDUCED	CONSTRUCT
Basic graph patterns	PROJECT	DESCRIBE
Groups	ORDER BY	ASK
OPTIONAL	LIMIT	
UNION	OFFSET	
GRAPH		
FILTER		

The result of converting such an abstract syntax tree is a SPARQL query that uses the following symbols in the SPARQL algebra:

Graph Pattern	Solution Modifiers
BGP	ToList
Join	OrderBy
LeftJoin	Project
Filter	Distinct
Union	Reduced
Graph	Slice

Slice is the combination of OFFSET and LIMIT. *mod* is any one of the solution modifiers.

ToList is used where conversion from the results of graph pattern matching to sequences occurs.

Definition: SPARQL Query

A **SPARQL Abstract Query** is a tuple (E, DS, R) where:

- E is a SPARQL algebra expression
- DS is an RDF Dataset
- R is a query form

12.2.1 Converting Graph Patterns

This section describes the process for translating a SPARQL graph patterns into a SPARQL algebra expression. After translating syntactic abbreviations for IRIs and triple patterns, it recursively processes syntactic forms into algebra expressions:

The working group notes that the point at the simplification step is applied leads to ambiguous transformation of queries involving a doubly nested filter and pattern in an optional:

```
OPTIONAL { { ... FILTER ( ... ?x ... ) } }..
```

This is illustrated by two non-normative test cases:

- Simplification applied after all transformations or not at all.
- Simplification applied during transformation.

First, expand abbreviations for IRIs and triple patterns given in section 4.

The WhereClause consists of a GroupGraphPattern which is comprised of the following forms:

- TriplesBlock
- Filter
- OptionalGraphPattern
- GroupOrUnionGraphPattern
- GraphGraphPattern

Each is translated by the following procedure:

Transform(syntax form)

If the form is TriplesBlock

```
The result is BGP(list of triple patterns)
```

If the form is GroupOrUnionGraphPattern

```
Let A := undefined

For each element G in the GroupOrUnionGraphPattern
    If A is undefined
        A := Transform(G)
    Else
        A := Union(A, Transform(G))

The result is A
```

If the form is GraphGraphPattern

```
If the form is GRAPH IRI GroupGraphPattern
    The result is Graph(IRI, Transform(GroupGraphPattern))
If the form is GRAPH Var GroupGraphPattern
    The result is Graph(Var, Transform(GroupGraphPattern))
```

If the form is GroupGraphPattern

We introduce the following symbols:

- Join(Pattern, Pattern)
- LeftJoin(Pattern, Pattern, expression)
- Filter(expression, Pattern)

```
Let FS := the empty set
Let G := the empty pattern, Z, a basic graph pattern which is the empty set.

For each element E in the GroupGraphPattern
   If E is of the form FILTER(expr)
       FS := FS set-union {expr}
   If E is of the form OPTIONAL{P}
   Then
       Let A := Transform(P)
       If A is of the form Filter(F, A2)
           G := LeftJoin(G, A2, F)
       else
           G := LeftJoin(G, A, true)
   If E is any other form:
      Let A := Transform(E)
      G := Join(G, A)


If FS is not empty:
  Let X := Conjunction of expressions in FS
  G := Filter(X, G)

The result is G.
```

Simplification step:

Groups of one graph pattern (not a filter) become join(Z, A) and can be replaced by A. The empty graph pattern Z is the identity for join:

```
Replace join(Z, A) by A
Replace join(A, Z) by A
```

12.2.2 Examples of Mapped Graph Patterns

The second form of a rewrite example is the first with empty group joins removed by the simplification step.

Example: group with a basic graph pattern consisting of a single triple pattern:

```
{ ?s ?p ?o }
```

```
Join(Z, BGP(?s ?p ?o) )
```

```
BGP(?s ?p ?o)
```

Example: group with a basic graph pattern consisting of two triple patterns:

```
{ ?s :p1 ?v1 ; :p2 ?v2 }
```

```
BGP( ?s :p1 ?v1 .?s :p2 ?v2 )
```

Example: group consisting of a union of two basic graph patterns:

```
{ { ?s :p1 ?v1 } UNION {?s :p2 ?v2 } }
```

```
Union(Join(Z, BGP(?s :p1 ?v1)),
      Join(Z, BGP(?s :p2 ?v2)) )
```

```
Union( BGP(?s :p1 ?v1) , BGP(?s :p2 ?v2) )
```

Example: group consisting a union of a union and a basic graph pattern:

```
{ { ?s :p1 ?v1 } UNION {?s :p2 ?v2 } UNION {?s :p3 ?v3 } }
```

```
Union(
    Union( Join(Z, BGP(?s :p1 ?v1)),
           Join(Z, BGP(?s :p2 ?v2))) ,
    Join(Z, BGP(?s :p3 ?v3)) )
```

```
Union(
    Union( BGP(?s :p1 ?v1) ,
           BGP(?s :p2 ?v2),
    BGP(?s :p3 ?v3))
```

Example: group consisting of a basic graph pattern and an optional graph pattern:

```
{ ?s :p1 ?v1 OPTIONAL {?s :p2 ?v2 } }
```

```
LeftJoin(
    Join(Z, BGP(?s :p1 ?v1)),
    Join(Z, BGP(?s :p2 ?v2)) ),
    true)
```

```
LeftJoin(BGP(?s :p1 ?v1), BGP(?s :p2 ?v2), true)
```

Example: group consisting of a basic graph pattern and two optional graph patterns:

```
{ ?s :p1 ?v1 OPTIONAL {?s :p2 ?v2 } OPTIONAL { ?s :p3 ?v3 } }
```

```
LeftJoin(
    LeftJoin(
        BGP(?s :p1 ?v1),
        BGP(?s :p2 ?v2),
        true) ,
    BGP(?s :p3 ?v3),
    true)
```

Example: group consisting of a basic graph pattern and an optional graph pattern with a filter:

```
{ ?s :p1 ?v1 OPTIONAL {?s :p2 ?v2 FILTER(?v1<3) } }
```

```
LeftJoin(
     Join(Z, BGP(?s :p1 ?v1)),
     Join(Z, BGP(?s :p2 ?v2)),
     (?v1<3) )
```

```
LeftJoin(
    BGP(?s :p1 ?v1) ,
    BGP(?s :p2 ?v2) ,
   (?v1<3) )
```

Example: group consisting of a union graph pattern and an optional graph pattern:

```
{ {?s :p1 ?v1} UNION {?s :p2 ?v2} OPTIONAL {?s :p3 ?v3} }
```

```
LeftJoin(
  Union(BGP(?s :p1 ?v1),
        BGP(?s :p2 ?v2)) ,
  BGP(?s :p3 ?v3) ,
  true )
```

Example: group consisting of a basic graph pattern, a filter and an optional graph pattern:

```
{ ?s :p1 ?v1 FILTER (?v1 < 3 ) OPTIONAL {?s :p2 ?v2} } }
```

```
Filter( ?v1 < 3 ,
  LeftJoin( BGP(?s :p1 ?v1), BGP(?s :p2 ?v2), true) ,
  )
```

12.2.3 Converting Solution Modifiers

Step 1 : ToList

ToList turns a multiset into a sequence with the same elements and cardinality. There is no implied ordering to the sequence; duplicates need not be adjacent.

Let M := ToList(Pattern)

Step 2 : ORDER BY

If the query string has an ORDER BY clause

M := OrderBy(M, list of order comparators)

Step 3 : Projection

M := Project(M, vars)

where vars is the set of variables mentioned in the SELECT clause or all named variables in the query if SELECT * used.

Step 4 : DISTINCT

If the query contains DISTINCT,

M := Distinct(M)

Step 5 : REDUCED

If the query contains REDUCED,

M := Reduced(M)

Step 6 : OFFSET and LIMIT

If the query contains "OFFSET start" or "LIMIT length"

M := Slice(M, start, length)

start defaults to 0

length defaults to (size(M)-start).

The overall abstract query is M.

12.3 Basic Graph Patterns

When matching graph patterns, the possible solutions form a *multiset* [multiset], also known as a *bag*. A multiset is an unordered collection of elements in which each element may appear more than once. It is described by a set of elements and a cardinality function giving the number of occurrences of each element from the set in the multiset.

Write μ for solution mappings and

Write μ_0 for the mapping such that dom(μ_0) is the empty set.

Write Ω_0 for the multiset consisting of exactly the empty mapping μ_0, with cardinality 1. This is the join identity.

Write μ(?x->t) for the solution mapping variable x to RDF term t : { (x, t) }

Write Ω(?x->t) for the multiset consisting of exactly μ(?x->t), that is, { { (x, t) } } with cardinality 1.

Definition: Compatible Mappings
Two solution mappings μ_1 and μ_2 are compatible if, for every variable v in dom(μ_1) and in dom(μ_2), $\mu_1(v) = \mu_2(v)$.

If μ_1 and μ_2 are compatible then μ_1 *set-union* μ_2 is also a mapping. Write merge(μ_1, μ_2) for μ_1 *set-union* μ_2

Write card[Ω](μ) for the cardinality of solution mapping μ in a multiset of mappings Ω.

12.3.1 SPARQL Basic Graph Pattern Matching

Basic graph patterns form the basis of SPARQL pattern matching. A basic graph pattern is matched against the active graph for that part of the query. Basic graph patterns can be instantiated by replacing both variables and blank nodes by terms, giving two notions of instance. Blank nodes are replaced using an RDF instance mapping, σ, from blank nodes to RDF terms; variables are replaced by a solution mapping from query variables to RDF terms.

Definition: Pattern Instance Mapping
A **Pattern Instance Mapping**, P, is the combination of an RDF instance mapping, σ, and solution mapping, μ. P(x) = μ(σ(x))

Any pattern instance mapping defines a unique solution mapping and a unique RDF instance mapping obtained by restricting it to query variables and blank nodes respectively.

Definition: Basic Graph Pattern Matching
Let BGP be a basic graph pattern and let G be an RDF graph.

μ is a **solution** for BGP from G when there is a pattern instance mapping P such that P(BGP) is a subgraph of G and μ is the restriction of P to the query variables in BGP.

card[Ω](μ) = card[Ω](number of distinct RDF instance mappings, σ, such that P = μ(σ) is a pattern instance mapping and P(BGP) is a subgraph of G).

If a basic graph pattern is the empty set, then the solution is Ω_0.

12.3.2 Treatment of Blank Nodes

This definition allows the solution mapping to bind a variable in a basic graph pattern, BGP, to a blank node in G. Since SPARQL treats blank node identifiers in a SPARQL Query Results XML Format document as scoped to the document, they cannot be understood as identifying nodes in the active graph of the dataset. If DS is the dataset of a query, pattern solutions are therefore understood to be not from the active graph of DS itself, but from an RDF graph, called the *scoping graph,* which is graph-equivalent to the active graph of DS but shares no blank nodes with DS or with BGP. The same scoping graph is used for all solutions to a single query. The scoping graph is purely a theoretical construct; in practice, the effect is obtained simply by the document scope conventions for blank node identifiers.

Since RDF blank nodes allow infinitely many redundant solutions for many patterns, there can be infinitely many pattern solutions (obtained by replacing blank nodes by different blank nodes). It is necessary, therefore, to somehow delimit the solutions for a basic graph pattern. SPARQL uses the subgraph match criterion to determine the solutions of a basic graph pattern. There is one solution for each distinct pattern instance mapping from the basic graph pattern to a subset of the active graph.

This is optimized for ease of computation rather than redundancy elimination. It allows query results to contain

redundancies even when the active graph of the dataset is lean, and it allows logically equivalent datasets to yield different query results.

12.4 SPARQL Algebra

For each symbol in a SPARQL abstract query, we define an operator for evaluation. The SPARQL algebra operators of the same name are used to evaluate SPARQL abstract query nodes as described in the section "Evaluation Semantics".

Definition: Filter

Let Ω be a multiset of solution mappings and expr be an expression. We define:

Filter(expr, Ω) = { μ | μ in Ω and expr(μ) is an expression that has an effective boolean value of true }

card[Filter(expr, Ω)](μ) = card[Ω](μ)

Definition: Join

Let Ω_1 and Ω_2 be multisets of solution mappings. We define:

Join(Ω_1, Ω_2) = { merge(μ_1, μ_2) | μ_1 in $\Omega_1$and μ_2 in Ω_2, and μ_1 and μ_2 are compatible }

card[Join(Ω_1, Ω_2)](μ) =
  for each merge(μ_1, μ_2), μ_1 in $\Omega_1$and μ_2 in Ω_2 such that μ = merge(μ_1, μ_2),
    sum over (μ_1, μ_2), card[Ω_1](μ_1)*card[Ω_2](μ_2)

It is possible that a solution mapping μ in a Join can arise in different solution mappings, $\mu_1$and μ_2 in the multisets being joined. The cardinality of μ is the sum of the cardinalities from all possibilities.

Definition: Diff

Let Ω_1 and Ω_2 be multisets of solution mappings. We define:

Diff(Ω_1, Ω_2, expr) = { μ | μ in Ω_1 such that for all μ′ in Ω_2, either μ and μ′ are not compatible or μ and μ' are compatible and expr(merge(μ, μ')) has an effective boolean value of false }

card[Diff(Ω_1, Ω_2, expr)](μ) = card[Ω_1](μ)

Diff is used internally for the definition of LeftJoin.

Definition: LeftJoin

Let Ω_1 and Ω_2 be multisets of solution mappings and expr be an expression. We define:

LeftJoin(Ω_1, Ω_2, expr) = Filter(expr, Join(Ω_1, Ω_2)) *set-union* Diff(Ω_1, Ω_2, expr)

card[LeftJoin(Ω_1, Ω_2, expr)](μ) = card[Filter(expr, Join(Ω_1, Ω_2))](μ) + card[Diff(Ω_1, Ω_2, expr)](μ)

Written in full that is:

LeftJoin(Ω_1, Ω_2, expr) =
  { merge(μ_1, μ_2) | μ_1 in $\Omega_1$and μ_2 in Ω_2, and μ_1 and μ_2 are compatible and expr(merge(μ_1, μ_2)) is true }
set-union
  { μ_1 | μ_1 in $\Omega_1$and μ_2 in Ω_2, and μ_1 and μ_2 are not compatible }
set-union
  { μ_1 | μ_1 in $\Omega_1$and μ_2 in Ω_2, and μ_1 and μ_2 are compatible and expr(merge(μ_1, μ_2)) is false }

As these are distinct, the cardinality of LeftJoin is cardinality of these individual components of the definition.

Definition: Union

Let Ω_1 and Ω_2 be multisets of solution mappings. We define:

Union(Ω_1, Ω_2) = { μ | μ in Ω_1 or μ in Ω_2 }

card[Union(Ω1, Ω2)](μ) = card[Ω1](μ) + card[Ω2](μ)

Write [x | C] for a sequence of elements where C(x) is true.

Write card[L](x) to be the cardinality of x in L.

Definition: ToList
Let Ω be a multiset of solution mappings. We define:
ToList(Ω) = a sequence of mappings μ in Ω in any order, with card[Ω](μ) occurrences of μ
card[ToList(Ω)](μ) = card[Ω](μ)

Definition: OrderBy
Let Ψ be a sequence of solution mappings. We define:
OrderBy(Ψ, condition) = [μ | μ in Ψ and the sequence satisfies the ordering condition]
card[OrderBy(Ψ, condition)](μ) = card[Ψ](μ)

Definition: Project
Let Ψ be a sequence of solution mappings and PV a set of variables.
For mapping μ, write Proj(μ, PV) to be the restriction of μ to variables in PV.
Project(Ψ, PV) = [Proj(Ψ[μ], PV) | μ in Ψ]
card[Project(Ψ, PV)](μ) = card[Ψ](μ)
The order of Project(Ψ, PV) must preserve any ordering given by OrderBy.

Definition: Distinct
Let Ψ be a sequence of solution mappings. We define:
Distinct(Ψ) = [μ | μ in Ψ]
card[Distinct(Ψ)](μ) = 1
The order of Distinct(Ψ) must preserve any ordering given by OrderBy.

Definition: Reduced
Let Ψ be a sequence of solution mappings. We define:
Reduced(Ψ) = [μ | μ in Ψ]
card[Reduced(Ψ)](μ) is between 1 and card[Ψ](μ)
The order of Reduced(Ψ) must preserve any ordering given by OrderBy.

The Reduced solution sequence modifier does not guarantee a defined cardinality.

Definition: Slice
Let Ψ be a sequence of solution mappings. We define:
Slice(Ψ, start, length)[i] = Ψ[start+i] for i = 0 to (length-1)

12.5 Evaluation Semantics

We define eval(D(G), graph pattern) as the evaluation of a graph pattern with respect to a dataset D having active graph G. The active graph is initially the default graph.

```
D : a dataset
D(G) : D a dataset with active graph G (the one patterns match against)
D[i] : The graph with IRI i in dataset D
D[DFT] : the default graph of D
P, P1, P2 : graph patterns
L : a solution sequence
```

Definition: Evaluation of Filter(F, P)
`eval(D(G), Filter(F, P)) = Filter(F, eval(D(G),P))`

Definition: Evaluation of Join(P1, P2)
`eval(D(G), Join(P1, P2)) = Join(eval(D(G), P1), eval(D(G), P2))`

Definition: Evaluation of LeftJoin(P1, P2, F)
`eval(D(G), LeftJoin(P1, P2, F)) = LeftJoin(eval(D(G), P1), eval(D(G), P2), F)`

Definition: Evaluation of a Basic Graph Pattern
`eval(D(G), BGP) = multiset of solution mappings`
See section 12.3 Basic Graph Patterns

Definition: Evaluation of a Union Pattern
`eval(D(G), Union(P1,P2)) = Union(eval(D(G), P1), eval(D(G), P2))`

Definition: Evaluation of a Graph Pattern

```
if IRI is a graph name in D
eval(D(G), Graph(IRI,P)) = eval(D(D[IRI]), P)

if IRI is not a graph name in D
eval(D(G), Graph(IRI,P)) = the empty multiset

eval(D(G), Graph(var,P)) =
     Let R be the empty multiset
     foreach IRI i in D
        R := Union(R, Join( eval(D(D[i]), P) , Ω(?var->i) )
     the result is R
```

The evaluation of graph uses the SPARQL algebra union operator. The cardinality of a solution mapping is the sum of the cardinalities of that solution mapping in each join operation.

Definition: Evaluation of ToList

```
eval(D, ToList(P)) = ToList(eval(D(D[DFT]), P))
```

Definition: Evaluation of Distinct

```
eval(D, Distict(L)) = Distinct(eval(D, L))
```

Definition: Evaluation of Reduced

```
eval(D, Reduced(L)) = Reduced(eval(D, L))
```

Definition: Evaluation of Project

```
eval(D, Project(L, vars)) = Project(eval(D, L), vars)
```

Definition: Evaluation of OrderBy

```
eval(D, OrderBy(L, condition)) = OrderBy(eval(D, L), condition)
```

Definition: Evaluation of Slice

```
eval(D, Slice(L, start, length)) = Slice(eval(D, L), start, length)
```

12.6 Extending SPARQL Basic Graph Matching

The overall SPARQL design can be used for queries which assume a more elaborate form of entailment than simple entailment, by re-writing the matching conditions for basic graph patterns. Since it is an open research problem to state such conditions in a single general form which applies to all forms of entailment and optimally eliminates needless or inappropriate redundancy, this document only gives necessary conditions which any such solution should satisfy. These will need to be extended to full definitions for each particular case.

Basic graph patterns stand in the same relation to triple patterns that RDF graphs do to RDF triples, and much of the same terminology can be applied to them. In particular, two basic graph patterns are said to be *equivalent* if there is a bijection M between the terms of the triple patterns that maps blank nodes to blank nodes and maps variables, literals and IRIs to themselves, such that a triple (s, p, o) is in the first pattern if and only if the triple (M(s), M(p) M(o)) is in the second. This definition extends that for RDF graph equivalence to basic graph patterns by preserving variable names across equivalent patterns.

An *entailment regime* specifies

1. a subset of RDF graphs called *well-formed* for the regime
2. an *entailment* relation between subsets of well-formed graphs and well-formed graphs.

Examples of entailment regimes include simple entailment [RDF-MT], RDF entailment [RDF-MT], RDFS entailment [RDF-MT], D-entailment [RDF-MT] and OWL-DL entailment [OWL-Semantics]. Of these, only OWL-DL entailment restricts the set of well-formed graphs. If E is an entailment regime then we will refer to E-entailment, E-consistency, etc, following this naming convention.

Some entailment regimes can categorize some RDF graphs as inconsistent. For example, the RDF graph:

```
_:x rdf:type xsd:string .
_:x rdf:type xsd:decimal .
```

is D-inconsistent when D contains the XSD datatypes. The effect of a query on an inconsistent graph is not covered by this specification, but must be specified by the particular SPARQL extension.

A SPARQL extension to E-entailment must satisfy the following conditions.

1 -- The scoping graph, SG, corresponding to any consistent active graph AG is uniquely specified and is E-equivalent to AG.

2 -- For any basic graph pattern BGP and pattern solution mapping P, P(BGP) is well-formed for E

3 -- For any scoping graph SG and answer set $\{P_1 \ldots P_n\}$ for a basic graph pattern BGP, and where $\{BGP_1 \ldots BGP_n\}$ is a set of basic graph patterns all equivalent to BGP, none of which share any blank nodes with any other or with SG

SG E-entails (SG union $P_1(BGP_1)$ union ... union $P_n(BGP_n)$)

These conditions do not fully determine the set of possible answers, since RDF allows unlimited amounts of redundancy. In addition, therefore, the following must hold.

4 -- Each SPARQL extension must provide conditions on answer sets which guarantee that every BGP and AG has a finite set of answers which is unique up to RDF graph equivalence.

Notes

(a) SG will often be graph equivalent to AG, but restricting this to E-equivalence allows some forms of normalization, for example elimination of semantic redundancies, to be applied to the source documents before querying.

(b) The construction in condition 3 ensures that any blank nodes introduced by the solution mapping are used in a way which is internally consistent with the way that blank nodes occur in SG. This ensures that blank node identifiers occur in more than one answer in an answer set only when the blank nodes so identified are indeed identical in SG. If the extension does not allow answer bindings to blank nodes, then this condition can be simplified to the condition:

SG E-entails P(BGP) for each pattern solution P.

(c) These conditions do not impose the SPARQL requirement that SG share no blank nodes with AG or BGP. In particular, it allows SG to actually be AG. This allows query protocols in which blank node identifiers retain their meaning between the query and the source document, or across multiple queries. Such protocols are not supported by the current SPARQL protocol specification, however.

(d) Since conditions 1 to 3 are only necessary conditions on answers, condition 4 allows cases where the set of legal answers can be restricted in various ways. For example, the current state of the art in OWL-DL querying focusses on the case where answer bindings to blank nodes are prohibited. We note that these conditions even allow the pathological 'mute' case where every query has an empty answer set.

(e) None of these conditions refer explicitly to instance mappings on blank nodes in BGP. For some entailment regimes, the existential interpretation of blank nodes cannot be fully captured by the existence of a single instance mapping. These conditions allow such regimes to give blank nodes in query patterns a 'fully existential' reading.

It is straightforward to show that SPARQL satisfies these conditions for the case where E is simple entailment, given that the SPARQL condition on SG is that it is graph-equivalent to AG but shares no blank nodes with AG or BGP (which satisfies the first condition). The only condition which is nontrivial is (3).

Every answer P_i is the solution mapping restriction of a SPARQL instance M_i such that $M_i(BGP_i)$ is a subgraph of SG. Since BGP_i and SG have no blank nodes in common, the range of M_i contains no blank nodes from BGP_i; therefore, the solution mapping P_i and RDF instance mapping I_i components of M_i commute, so $M_i(BGP_i) = I_i(P_i(BGP_i))$. So

$M_1(BGP_1)$ union ... union $M_n(BGP_n)$
$= I_1(P_1(BGP_1))$ union ... union $I_n(P_n(BGP_n))$
$= [I_1 + \ldots + I_n](P_1(BGP_1)$ union ... union $P_n(BGP_n))$

since the domains of the I_i instance mappings are all mutually exclusive. Since they are also exclusive from SG,

SG union $[I_1 + \ldots + I_n](P_1(BGP_1)$ union ... union $P_n(BGP_n))$
$= [I_1 + \ldots + I_n](SG$ union $P_1(BGP_1)$ union ... union $P_n(BGP_n))$

i.e.

SG union $P_1(BGP_1)$ union ... union $P_n(BGP_n)$

has an instance which is a subgraph of SG, so is simply entailed by SG by the RDF interpolation lemma [RDF-MT].

A. SPARQL Grammar

A.1 SPARQL Query String

A SPARQL query string is a Unicode character string (c.f. section 6.1 String concepts of [CHARMOD]) in the language defined by the following grammar, starting with the Query production. For compatibility with future versions of Unicode, the characters in this string may include Unicode codepoints that are unassigned as of the date of this publication (see Identifier and Pattern Syntax [UNIID] section 4 Pattern Syntax). For productions with excluded

character classes (for example `[^<>'{}|^`]`), the characters are excluded from the range `#x0 - #x10FFFF`.

A.2 Codepoint Escape Sequences

A SPARQL Query String is processed for codepoint escape sequences before parsing by the grammar defined in EBNF below. The codepoint escape sequences for a SPARQL query string are:

Escape	Unicode code point
'\u' HEX HEX HEX HEX	A Unicode code point in the range U+0 to U+FFFF inclusive corresponding to the encoded hexadecimal value.
'\U' HEX HEX HEX HEX HEX HEX HEX HEX	A Unicode code point in the range U+0 to U+10FFFF inclusive corresponding to the encoded hexadecimal value.

where HEX is a hexadecimal character

```
HEX ::= [0-9] | [A-F] | [a-f]
```

Examples:

```
<ab\u00E9xy>        # Codepoint 00E9 is Latin small e with acute - é
\u03B1:a            # Codepoint x03B1 is Greek small alpha - α
a\u003Ab            # a:b -- codepoint x3A is colon
```

Codepoint escape sequences can appear anywhere in the query string. They are processed before parsing based on the grammar rules and so may be replaced by codepoints with significance in the grammar, such as ":" marking a prefixed name.

These escape sequences are not included in the grammar below. Only escape sequences for characters that would be legal at that point in the grammar may be given. For example, the variable "`?x\u0020y`" is not legal (`\u0020` is a space and is not permitted in a variable name).

A.3 White Space

White space (production `WS`) is used to separate two terminals which would otherwise be (mis-)recognized as one terminal. Rule names below in capitals indicate where white space is significant; these form a possible choice of terminals for constructing a SPARQL parser. White space is significant in strings.

For example:

```
?a<?b&&?c>?d
```

is the token sequence variable '`?a`', an IRI '`<?b&&?c>`', and variable '`?d`', not a expression involving the operator '`&&`' connexting two expression using '<' (less than) and '>' (greater than).

A.4 Comments

Comments in SPARQL queries take the form of '`#`', outside an IRI or string, and continue to the end of line (marked by characters `0x0D` or `0x0A`) or end of file if there is no end of line after the comment marker. Comments are treated as white space.

A.5 IRI References

Text matched by the `IRI_REF` production and *`PrefixedName`* (after prefix expansion) production, after escape processing, must be conform to the generic syntax of IRI references in section 2.2 of RFC 3987 "ABNF for IRI References and IRIs" [RFC3987]. For example, the `IRI_REF` `<abc#def>` may occur in a SPARQL query string, but the `IRI_REF` `<abc##def>` must not.

Base IRIs declared with the BASE keyword must be absolute IRIs. A prefix declared with the PREFIX keyword may not be re-declared in the same query. See section 2.1.1, Syntax of IRI Terms, for a description of BASE and PREFIX.

A.6 Blank Node Labels

The same blank node label may not be used in two separate basic graph patterns with a single query.

A.7 Escape sequences in strings

In addition to the codepoint escape sequences, the following escape sequences any `string` production (e.g. `STRING_LITERAL1`, `STRING_LITERAL2`, `STRING_LITERAL_LONG1`, `STRING_LITERAL_LONG2`):

Escape	Unicode code point
'\t'	U+0009 (tab)
'\n'	U+000A (line feed)
'\r'	U+000D (carriage return)
'\b'	U+0008 (backspace)
'\f'	U+000C (form feed)
'\"'	U+0022 (quotation mark, double quote mark)
"\'"	U+0027 (apostrophe-quote, single quote mark)
'\\'	U+005C (backslash)

Examples:

```
"abc\n"
"xy\rz"
'xy\tz'
```

A.8 Grammar

The EBNF notation used in the grammar is defined in Extensible Markup Language (XML) 1.1 [XML11] section 6 Notation.

Keywords are matched in a case-insensitive manner with the exception of the keyword '`a`' which, in line with Turtle and N3, is used in place of the IRI `rdf:type` (in full, `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`).

Keywords:

BASE	SELECT	ORDER BY	FROM	GRAPH	STR	isURI
PREFIX	CONSTRUCT	LIMIT	FROM NAMED	OPTIONAL	LANG	isIRI
	DESCRIBE	OFFSET	WHERE	UNION	LANGMATCHES	isLITERAL
	ASK	DISTINCT		FILTER	DATATYPE	REGEX
		REDUCED		a	BOUND	true
					sameTERM	false

Escape sequences are case sensitive.

When choosing a rule to match, the longest match is chosen.

[1]	*Query*	::=	Prologue (SelectQuery | ConstructQuery | DescribeQuery | AskQuery)
[2]	*Prologue*	::=	BaseDecl? PrefixDecl*
[3]	*BaseDecl*	::=	'BASE' IRI_REF
[4]	*PrefixDecl*	::=	'PREFIX' PNAME_NS IRI_REF
[5]	*SelectQuery*	::=	'SELECT' ('DISTINCT' | 'REDUCED')? (Var+ | '*') DatasetClause* WhereClause SolutionModifier
[6]	*ConstructQuery*	::=	'CONSTRUCT' ConstructTemplate DatasetClause* WhereClause SolutionModifier
[7]	*DescribeQuery*	::=	'DESCRIBE' (VarOrIRIref+ | '*') DatasetClause* WhereClause? SolutionModifier
[8]	*AskQuery*	::=	'ASK' DatasetClause* WhereClause
[9]	*DatasetClause*	::=	'FROM' (DefaultGraphClause | NamedGraphClause)
[10]	*DefaultGraphClause*	::=	SourceSelector
[11]	*NamedGraphClause*	::=	'NAMED' SourceSelector
[12]	*SourceSelector*	::=	IRIref
[13]	*WhereClause*	::=	'WHERE'? GroupGraphPattern
[14]	*SolutionModifier*	::=	OrderClause? LimitOffsetClauses?
[15]	*LimitOffsetClauses*	::=	(LimitClause OffsetClause? | OffsetClause LimitClause?)
[16]	*OrderClause*	::=	'ORDER' 'BY' OrderCondition+
[17]	*OrderCondition*	::=	(('ASC' | 'DESC') BrackettedExpression) | (Constraint | Var)
[18]	*LimitClause*	::=	'LIMIT' INTEGER
[19]	*OffsetClause*	::=	'OFFSET' INTEGER

[20]	*GroupGraphPattern*	::=	'{' TriplesBlock? ((GraphPatternNotTriples | Filter) '.'? TriplesBlock?)* '}'
[21]	*TriplesBlock*	::=	TriplesSameSubject ('.' TriplesBlock?)?
[22]	*GraphPatternNotTriples*	::=	OptionalGraphPattern | GroupOrUnionGraphPattern | GraphGraphPattern
[23]	*OptionalGraphPattern*	::=	'OPTIONAL' GroupGraphPattern
[24]	*GraphGraphPattern*	::=	'GRAPH' VarOrIRIref GroupGraphPattern
[25]	*GroupOrUnionGraphPattern*	::=	GroupGraphPattern ('UNION' GroupGraphPattern)*
[26]	*Filter*	::=	'FILTER' Constraint
[27]	*Constraint*	::=	BrackettedExpression | BuiltInCall | FunctionCall
[28]	*FunctionCall*	::=	IRIref ArgList
[29]	*ArgList*	::=	(NIL | '(' Expression (',' Expression)* ')')
[30]	*ConstructTemplate*	::=	'{' ConstructTriples? '}'
[31]	*ConstructTriples*	::=	TriplesSameSubject ('.' ConstructTriples?)?
[32]	*TriplesSameSubject*	::=	VarOrTerm PropertyListNotEmpty | TriplesNode PropertyList
[33]	*PropertyListNotEmpty*	::=	Verb ObjectList (';' (Verb ObjectList)?)*
[34]	*PropertyList*	::=	PropertyListNotEmpty?
[35]	*ObjectList*	::=	Object (',' Object)*
[36]	*Object*	::=	GraphNode
[37]	*Verb*	::=	VarOrIRIref | 'a'
[38]	*TriplesNode*	::=	Collection | BlankNodePropertyList
[39]	*BlankNodePropertyList*	::=	'[' PropertyListNotEmpty ']'
[40]	*Collection*	::=	'(' GraphNode+ ')'
[41]	*GraphNode*	::=	VarOrTerm | TriplesNode
[42]	*VarOrTerm*	::=	Var | GraphTerm
[43]	*VarOrIRIref*	::=	Var | IRIref
[44]	*Var*	::=	VAR1 | VAR2
[45]	*GraphTerm*	::=	IRIref | RDFLiteral | NumericLiteral | BooleanLiteral | BlankNode | NIL
[46]	*Expression*	::=	ConditionalOrExpression
[47]	*ConditionalOrExpression*	::=	ConditionalAndExpression ('||' ConditionalAndExpression)*
[48]	*ConditionalAndExpression*	::=	ValueLogical ('&&' ValueLogical)*
[49]	*ValueLogical*	::=	RelationalExpression
[50]	*RelationalExpression*	::=	NumericExpression ('=' NumericExpression | '!=' NumericExpression | '<' NumericExpression | '>' NumericExpression | '<=' NumericExpression | '>=' NumericExpression)?
[51]	*NumericExpression*	::=	AdditiveExpression
[52]	*AdditiveExpression*	::=	MultiplicativeExpression ('+' MultiplicativeExpression | '-' MultiplicativeExpression | NumericLiteralPositive | NumericLiteralNegative)*
[53]	*MultiplicativeExpression*	::=	UnaryExpression ('*' UnaryExpression | '/' UnaryExpression)*
[54]	*UnaryExpression*	::=	'!' PrimaryExpression | '+' PrimaryExpression | '-' PrimaryExpression | PrimaryExpression
[55]	*PrimaryExpression*	::=	BrackettedExpression | BuiltInCall | IRIrefOrFunction | RDFLiteral | NumericLiteral | BooleanLiteral | Var
[56]	*BrackettedExpression*	::=	'(' Expression ')'
[57]	*BuiltInCall*	::=	'STR' '(' Expression ')' | 'LANG' '(' Expression ')' | 'LANGMATCHES' '(' Expression ',' Expression ')' | 'DATATYPE' '(' Expression ')' | 'BOUND' '(' Var ')' | 'sameTerm' '(' Expression ',' Expression ')' | 'isIRI' '(' Expression ')' | 'isURI' '(' Expression ')' | 'isBLANK' '(' Expression ')' | 'isLITERAL' '(' Expression ')' | RegexExpression
[58]	*RegexExpression*	::=	'REGEX' '(' Expression ',' Expression (',' Expression)? ')'
[59]	*IRIrefOrFunction*	::=	IRIref ArgList?
[60]	*RDFLiteral*	::=	String (LANGTAG | ('^^' IRIref))?
[61]	*NumericLiteral*	::=	NumericLiteralUnsigned | NumericLiteralPositive | NumericLiteralNegative

[62]	*NumericLiteralUnsigned*	::=	INTEGER | DECIMAL | DOUBLE
[63]	*NumericLiteralPositive*	::=	INTEGER_POSITIVE | DECIMAL_POSITIVE | DOUBLE_POSITIVE
[64]	*NumericLiteralNegative*	::=	INTEGER_NEGATIVE | DECIMAL_NEGATIVE | DOUBLE_NEGATIVE
[65]	*BooleanLiteral*	::=	'true' | 'false'
[66]	*String*	::=	STRING_LITERAL1 | STRING_LITERAL2 | STRING_LITERAL_LONG1 | STRING_LITERAL_LONG2
[67]	*IRIref*	::=	IRI_REF | PrefixedName
[68]	*PrefixedName*	::=	PNAME_LN | PNAME_NS
[69]	*BlankNode*	::=	BLANK_NODE_LABEL | ANON

Productions for terminals:

[70]	*IRI_REF*	::=	'<' ([^<>"{}|^`\]-[#x00-#x20])* '>'
[71]	*PNAME_NS*	::=	PN_PREFIX? ':'
[72]	*PNAME_LN*	::=	PNAME_NS PN_LOCAL
[73]	*BLANK_NODE_LABEL*	::=	'_:' PN_LOCAL
[74]	*VAR1*	::=	'?' VARNAME
[75]	*VAR2*	::=	'$' VARNAME
[76]	*LANGTAG*	::=	'@' [a-zA-Z]+ ('-' [a-zA-Z0-9]+)*
[77]	*INTEGER*	::=	[0-9]+
[78]	*DECIMAL*	::=	[0-9]+ '.' [0-9]* | '.' [0-9]+
[79]	*DOUBLE*	::=	[0-9]+ '.' [0-9]* EXPONENT | '.' ([0-9])+ EXPONENT | ([0-9])+ EXPONENT
[80]	*INTEGER_POSITIVE*	::=	'+' INTEGER
[81]	*DECIMAL_POSITIVE*	::=	'+' DECIMAL
[82]	*DOUBLE_POSITIVE*	::=	'+' DOUBLE
[83]	*INTEGER_NEGATIVE*	::=	'-' INTEGER
[84]	*DECIMAL_NEGATIVE*	::=	'-' DECIMAL
[85]	*DOUBLE_NEGATIVE*	::=	'-' DOUBLE
[86]	*EXPONENT*	::=	[eE] [+-]? [0-9]+
[87]	*STRING_LITERAL1*	::=	"'" (([^#x27#x5C#xA#xD]) | ECHAR)* "'"
[88]	*STRING_LITERAL2*	::=	'"' (([^#x22#x5C#xA#xD]) | ECHAR)* '"'
[89]	*STRING_LITERAL_LONG1*	::=	"'''" (("'" | "''")? ([^'\] | ECHAR))* "'''"
[90]	*STRING_LITERAL_LONG2*	::=	'"""' (('"' | '""')? ([^"\] | ECHAR))* '"""'
[91]	*ECHAR*	::=	'\' [tbnrf\"']
[92]	*NIL*	::=	'(' WS* ')'
[93]	*WS*	::=	#x20 | #x9 | #xD | #xA
[94]	*ANON*	::=	'[' WS* ']'
[95]	*PN_CHARS_BASE*	::=	[A-Z] | [a-z] | [#x00C0-#x00D6] | [#x00D8-#x00F6] | [#x00F8-#x02FF] | [#x0370-#x037D] | [#x037F-#x1FFF] | [#x200C-#x200D] | [#x2070-#x218F] | [#x2C00-#x2FEF] | [#x3001-#xD7FF] | [#xF900-#xFDCF] | [#xFDF0-#xFFFD] | [#x10000-#xEFFFF]
[96]	*PN_CHARS_U*	::=	PN_CHARS_BASE | '_'
[97]	*VARNAME*	::=	(PN_CHARS_U | [0-9]) (PN_CHARS_U | [0-9] | #x00B7 | [#x0300-#x036F] | [#x203F-#x2040])*
[98]	*PN_CHARS*	::=	PN_CHARS_U | '-' | [0-9] | #x00B7 | [#x0300-#x036F] | [#x203F-#x2040]
[99]	*PN_PREFIX*	::=	PN_CHARS_BASE ((PN_CHARS|'.')* PN_CHARS)?
[100]	*PN_LOCAL*	::=	(PN_CHARS_U | [0-9]) ((PN_CHARS|'.')* PN_CHARS)? Note that SPARQL local names allow leading digits while XML local names do not.

Notes:

1. The SPARQL grammar is LL(1) when the rules with uppercased names are used as terminals.
2. In signed numbers, no white space is allowed between the sign and the number. The *AdditiveExpression* grammar rule allows for this by covering the the two cases of an expression followed by a signed number. These produce an addition or substraction of the unsigned number as appropriate.

Some grammar files for some commonly used tools are available here.

B. Conformance

See appendix A SPARQL Grammar regarding conformance of SPARQL Query strings, and section 10 Query Forms

for conformance of query results. See appendix E. Internet Media Type for conformance to the application/sparql-query media type.

This specification is intended for use in conjunction with the SPARQL Protocol [SPROT] and the SPARQL Query Results XML Format [RESULTS]. See those specifications for their conformance criteria.

Note that the SPARQL protocol describes an abstract interface as well as a network protocol, and the abstract interface may apply to APIs as well as network interfaces.

C. Security Considerations (Informative)

SPARQL queries using FROM, FROM NAMED, or GRAPH may cause the specified URI to be dereferenced. This may cause additional use of network, disk or CPU resources along with associated secondary issues such as denial of service. The security issues of Uniform Resource Identifier (URI): Generic Syntax [RFC3986] Section 7 should be considered. In addition, the contents of `file:` URIs can in some cases be accessed, processed and returned as results, providing unintended access to local resources.

The SPARQL language permits extensions, which will have their own security implications.

Multiple IRIs may have the same appearance. Characters in different scripts may look similar (a Cyrillic "o" may appear similar to a Latin "o"). A character followed by combining characters may have the same visual representation as another character (LATIN SMALL LETTER E followed by COMBINING ACUTE ACCENT has the same visual representation as LATIN SMALL LETTER E WITH ACUTE). Users of SPARQL must take care to construct queries with IRIs that match the IRIs in the data. Further information about matching of similar characters can be found in Unicode Security Considerations [UNISEC] and Internationalized Resource Identifiers (IRIs) [RFC3987] Section 8.

D. Internet Media Type, File Extension and Macintosh File Type

contact:
Eric Prud'hommeaux
See also:
How to Register a Media Type for a W3C Specification
Internet Media Type registration, consistency of use
TAG Finding 3 June 2002 (Revised 4 September 2002)

The Internet Media Type / MIME Type for the SPARQL Query Language is "application/sparql-query".

It is recommended that sparql query files have the extension ".rq" (all lowercase) on all platforms.

It is recommended that sparql query files stored on Macintosh HFS file systems be given a file type of "TEXT".

Type name:
application
Subtype name:
sparql-query
Required parameters:
None
Optional parameters:
None
Encoding considerations:
The syntax of the SPARQL Query Language is expressed over code points in Unicode [UNICODE]. The encoding is always UTF-8 [RFC3629].
Unicode code points may also be expressed using an \uXXXX (U+0 to U+FFFF) or \UXXXXXXXX syntax (for U+10000 onwards) where X is a hexadecimal digit [0-9A-F]
Security considerations:
See SPARQL Query appendix C, Security Considerations as well as RFC 3629 [RFC3629] section 7, Security Considerations.
Interoperability considerations:
There are no known interoperability issues.
Published specification:
This specification.
Applications which use this media type:
No known applications currently use this media type.
Additional information:
Magic number(s):
A SPARQL query may have the string 'PREFIX' (case independent) near the beginning of the document.
File extension(s):
".rq"

Base URI:
The SPARQL 'BASE <IRIref>' term can change the current base URI for relative IRIrefs in the query language that are used sequentially later in the document.
Macintosh file type code(s):
"TEXT"
Person & email address to contact for further information:
public-rdf-dawg-comments@w3.org
Intended usage:
COMMON
Restrictions on usage:
None
Author/Change controller:
The SPARQL specification is a work product of the World Wide Web Consortium's RDF Data Access Working Group. The W3C has change control over these specifications.

E. References

Normative References

[CHARMOD]
Character Model for the World Wide Web 1.0: Fundamentals, R. Ishida, F. Yergeau, M. J. Düst, M. Wolf, T. Texin, Editors, W3C Recommendation, 15 February 2005, http://www.w3.org/TR/2005/REC-charmod-20050215/ . Latest version available at http://www.w3.org/TR/charmod/ .
[CONCEPTS]
Resource Description Framework (RDF): Concepts and Abstract Syntax, G. Klyne, J. J. Carroll, Editors, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ . Latest version available at http://www.w3.org/TR/rdf-concepts/ .
[FUNCOP]
XQuery 1.0 and XPath 2.0 Functions and Operators, J. Melton, A. Malhotra, N. Walsh, Editors, W3C Recommendation, 23 January 2007, http://www.w3.org/TR/2007/REC-xpath-functions-20070123/ . Latest version available at http://www.w3.org/TR/xpath-functions/ .
[RDF-MT]
RDF Semantics, P. Hayes, Editor, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-mt-20040210/ . Latest version available at http://www.w3.org/TR/rdf-mt/ .
[RFC3629]
RFC 3629 *UTF-8, a transformation format of ISO 10646*, F. Yergeau November 2003
[RFC4647]
RFC 4647 *Matching of Language Tags*, A. Phillips, M. Davis September 2006
[RFC3986]
RFC 3986 *Uniform Resource Identifier (URI): Generic Syntax*, T. Berners-Lee, R. Fielding, L. Masinter January 2005
[RFC3987]
RFC 3987, "Internationalized Resource Identifiers (IRIs)", M. Dürst , M. Suignard
[UNICODE]
The Unicode Standard, Version 4. ISBN 0-321-18578-1, as updated from time to time by the publication of new versions. The latest version of Unicode and additional information on versions of the standard and of the Unicode Character Database is available at http://www.unicode.org/unicode/standard/versions/.
[XML11]
Extensible Markup Language (XML) 1.1, J. Cowan, J. Paoli, E. Maler, C. M. Sperberg-McQueen, F. Yergeau, T. Bray, Editors, W3C Recommendation, 4 February 2004, http://www.w3.org/TR/2004/REC-xml11-20040204/ . Latest version available at http://www.w3.org/TR/xml11/ .
[XPATH20]
XML Path Language (XPath) 2.0, A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, J. Siméon, Editors, W3C Recommendation, 23 January 2007, http://www.w3.org/TR/2007/REC-xpath20-20070123/ . Latest version available at http://www.w3.org/TR/xpath20/ .
[XQUERY]
XQuery 1.0: An XML Query Language, S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, Editors, W3C Recommendation, 23 January 2007, http://www.w3.org/TR/2007/REC-xquery-20070123/. Latest version available at http://www.w3.org/TR/xquery/ .
[XSDT]
XML Schema Part 2: Datatypes Second Edition, P. V. Biron, A. Malhotra, Editors, W3C Recommendation, 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/ . Latest version available at http://www.w3.org/TR/xmlschema-2/ .
[BCP47]
Best Common Practice 47, P. V. Biron, A. Malhotra, Editors, W3C Recommendation, 28 October 2004, http://www.rfc-editor.org/rfc/bcp/bcp47.txt .

Informative References

[CBD]
CBD - Concise Bounded Description, Patrick Stickler, Nokia, W3C Member Submission, 3 June 2005.

[DC]
Expressing Simple Dublin Core in RDF/XML Dublin Core Dublin Core Metadata Initiative Recommendation 2002-07-31.

[Multiset]
Multiset, Wikipedia, The Free Encyclopedia. Article as given on October 25, 2007 at http://en.wikipedia.org/w/index.php?title=Multiset&oldid=163605900. The latest version of this article is at http://en.wikipedia.org/wiki/Multiset.

[OWL-Semantics]
OWL Web Ontology Language Semantics and Abstract Syntax, Peter F. Patel-Schneider, Patrick Hayes, Ian Horrocks, Editors, W3C Recommendation http://www.w3.org/TR/2004/REC-owl-semantics-20040210/. Latest version at http://www.w3.org/TR/owl-semantics/.

[RDFS]
RDF Vocabulary Description Language 1.0: RDF Schema, Dan Brickley, R.V. Guha, Editors, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-schema-20040210/ . Latest version at http://www.w3.org/TR/rdf-schema/ .

[RESULTS]
SPARQL Query Results XML Format, D. Beckett, Editor, W3C Recommendation, 15 January 2008, http://www.w3.org/TR/2008/REC-rdf-sparql-XMLres-20080115/ . Latest version available at http://www.w3.org/TR/rdf-sparql-XMLres/ .

[SPROT]
SPARQL Protocol for RDF, K. Clark, Editor, W3C Recommendation, 15 January 2008, http://www.w3.org/TR/2008/REC-rdf-sparql-protocol-20080115/ . Latest version available at http://www.w3.org/TR/rdf-sparql-protocol/ .

[TURTLE]
Turtle - Terse RDF Triple Language, Dave Beckett.

[UCNR]
RDF Data Access Use Cases and Requirements, K. Clark, Editor, W3C Working Draft, 25 March 2005, http://www.w3.org/TR/2005/WD-rdf-dawg-uc-20050325/ . Latest version available at http://www.w3.org/TR/rdf-dawg-uc/ .

[UNISEC]
Unicode Security Considerations, Mark Davis, Michel Suignard

[VCARD]
Representing vCard Objects in RDF/XML, Renato Iannella, W3C Note, 22 February 2001, http://www.w3.org/TR/2001/NOTE-vcard-rdf-20010222/ . Latest version is available at `http://www.w3.org/TR/vcard-rdf` .

[WEBARCH]
Architecture of the World Wide Web, Volume One, I. Jacobs, N. Walsh, Editors, W3C Recommendation, 15 December 2004, http://www.w3.org/TR/2004/REC-webarch-20041215/ . Latest version is available at `http://www.w3.org/TR/webarch/` .

[UNIID]
Identifier and Pattern Syntax 4.1.0, Mark Davis, Unicode Standard Annex #31, 25 March 2005, http://www.unicode.org/reports/tr31/tr31-5.html . Latest version available at http://www.unicode.org/reports/tr31/ .

[SPARQL-sem-05]
A relational algebra for SPARQL, Richard Cyganiak, 2005

[SPARQL-sem-06]
Semantics of SPARQL, Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez, 2006

F. Acknowledgements (Informative)

The SPARQL RDF Query Language is a product of the whole of the W3C RDF Data Access Working Group, and our thanks for discussions, comments and reviews go to all present and past members.

In addition, we have had comments and discussions with many people through the working group comments list. All comments go to making a better document. Andy would also like to particularly thank Jorge Peérez, Geoff Chappell, Bob MacGregor, Yosi Scharf and Richard Newman for exploring specific issues related to SPARQL. Eric would like to acknowledge the invaluable help of Björn Höhrmann.

Change Log

This is a high-level summary of changes made to this document since publication of the 14 June 2007 Candidate Recommendation:

- In §9 Solution Sequences and Modifiers, the term `solution set` was changed to `solution sequence`.
- The media type `application/sparql-query` was approved so the text about the status of that request was removed.