SPARQL 1.1 Query Language

W3C Recommendation 21 March 2013

This version:
http://www.w3.org/TR/2013/REC-sparql11-query-20130321/
Latest version:
http://www.w3.org/TR/sparql11-query/
Previous version:
http://www.w3.org/TR/2012/PR-sparql11-query-20121108/
Editors:
Steve Harris, Garlik, a part of Experian
Andy Seaborne, The Apache Software Foundation
Previous Editor:
Eric Prud'hommeaux, W3C

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 aggregation, subqueries, negation, creating values by expressions, extensible value testing, and constraining queries by source RDF graph. The results of SPARQL queries can be result sets or RDF graphs.

Status of This Document

May Be Superseded

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/.

Set of Documents

This document is one of eleven SPARQL 1.1 Recommendations produced by the SPARQL Working Group:

1. SPARQL 1.1 Overview
2. SPARQL 1.1 Query Language (this document)
3. SPARQL 1.1 Update
4. SPARQL1.1 Service Description
5. SPARQL 1.1 Federated Query
6. SPARQL 1.1 Query Results JSON Format
7. SPARQL 1.1 Query Results CSV and TSV Formats
8. SPARQL Query Results XML Format (Second Edition)
9. SPARQL 1.1 Entailment Regimes
10. SPARQL 1.1 Protocol
11. SPARQL 1.1 Graph Store HTTP Protocol

No Substantive Changes

There have been no substantive changes to this document since the previous version. Minor editorial changes, if any, are detailed in the change log and visible in the color-coded diff.

Please Send Comments

Please send any comments to public-rdf-dawg-comments@w3.org (public archive). Although work on this document by the SPARQL Working Group is complete, comments may be addressed in the errata or in future revisions. Open discussion is welcome at public-sparql-dev@w3.org (public archive).

Endorsed By W3C

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.

Patents

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

---

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] and SPARQL New Features and Rationale [UCNR2].

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 patterns to test for the absense of information.

Section 9 adds property paths to graph pattern matching, giving a compact representation of queries and also the ability to match arbitrary length paths in the graph.

Section 10 describes the forms of assignment possible in SPARQL.

Sections 11 introduces the mechanism to group and aggregate results, which can be incorporated as subqueries as described in Section 12.

Section 13 introduces the ability to constrain portions of a query to particular source graphs. Section 13 also presents SPARQL's mechanism for defining the source graphs for a query.

Section 14 refers to the separate document SPARQL 1.1 Federated Query.

Section 15 defines the constructs that affect the solutions of a query by ordering, slicing, projecting, limiting, and removing duplicates from a sequence of solutions.

Section 16 defines the four types of SPARQL queries that produce results in different forms.

Section 17 defines SPARQL's extensible value testing and expression framework. It presents the functions and operators that can be used to constrain the values that appear in a query's results and also calculate new values to be returned by a query.

Section 18 is a formal definition of the evaluation of SPARQL graph patterns and solution modifiers.

Section 19 contains the normative definition of the syntax for the SPARQL query and SPARQL update languages, 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#
sfn:	http://www.w3.org/ns/sparql#

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

In addition, we define the following terms:

- RDF Term, which includes IRIs, blank nodes and literals
- Simple Literal, which covers literals without language tag or datatype IRI

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 tag "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 (see "SPARQL Query Results XML Format" and "SPARQL 1.1 Query Results JSON 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 Creating Values with Expressions

SPARQL 1.1 allows to create values from complex expressions. The queries below show how to the CONCAT function can be used to concatenate first names and last names from foaf data, then assign the value using an expression in the SELECT clause and also assign the value by using the BIND form.

Data:

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

_:a  foaf:givenName   "John" .
_:a  foaf:surname  "Doe" .
```

Query:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ( CONCAT(?G, " ", ?S) AS ?name )
WHERE  { ?P foaf:givenName ?G ; foaf:surname ?S }
```

Query:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE  {
   ?P foaf:givenName ?G ;
      foaf:surname ?S
   BIND(CONCAT(?G, " ", ?S) AS ?name)
}
```

name

"John Doe"

2.6 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 foaf: <http://xmlns.com/foaf/0.1/> .

_: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 FILTERs restrict solutions to those for which the filter expression evaluates to TRUE.

This section provides an informal introduction to SPARQL FILTERs; their semantics are defined in section 'Expressions and Testing Values' where there is a comprehensive function library. 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 Value of Strings

SPARQL FILTER functions like regex can test RDF literals. regex matches only string literals. 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 FILTERs 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 Operand Data Types). Section Operator Mapping describes the operators and section Function Definitions the functions that can be that can be applied to RDF terms.

4 SPARQL Syntax

This section covers the syntax used by SPARQL for RDF terms and triple patterns. The full grammar is given in section 19.

4.1 RDF Term Syntax

4.1.1 Syntax for IRIs

The iri 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 IRIREF 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.

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.

4.1.1.1 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. SPARQL local names also allow the non-alphanumeric characters allowed in IRIs via backslash character escapes (e.g. ns:id\=123). SPARQL local names have more syntactic restrictions than CURIEs.

4.1.1.2 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`

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

A query variable is marked by the use of 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.

4.1.4 Syntax for Blank Nodes

Blank nodes in graph patterns act as 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> .
```

4.2 Syntax for Triple Patterns

Triple Patterns are written as 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 }
```

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 .
```

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

```
[17]  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, with optional filters, comprises a single basic graph pattern. Any other 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 evaluates basic graph patterns using subgraph matching, which is defined for simple entailment. SPARQL can be extended to other forms of entailment given certain conditions as described below. The document SPARQL 1.1 Entailment Regimes describes several specific entailment regimes.

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 . }
       }
```

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 } }
```

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 }
       }
```

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

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

8 Negation

The SPARQL query language incorporates two styles of negation, one based on filtering results depending on whether a graph pattern does or does not match in the context of the query solution being filtered, and one based on removing solutions related to another pattern.

8.1 Filtering Using Graph Patterns

Filtering of query solutions is done within a `FILTER` expression using `NOT EXISTS` and `EXISTS`. Note that the filter scope rules apply to the whole group in which the filter appears.

8.1.1 Testing For the Absence of a Pattern

The `NOT EXISTS` filter expression tests whether a graph pattern does not match the dataset, given the values of variables in the group graph pattern in which the filter occurs. It does not generate any additional bindings.

Data:

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

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

Query:

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

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

Query Result:

person
<http://example/bob>

8.1.2 Testing For the Presence of a Pattern

The filter expression `EXISTS` is also provided. It tests whether the pattern can be found in the data; it does not generate any additional bindings.

Query:

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

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

Query Result:

person
<http://example/alice>

8.2 Removing Possible Solutions

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

Data:

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

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

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

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

Query:

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

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

Results:

s
<http://example/carol>
<http://example/alice>

8.3 Relationship and differences between NOT EXISTS and MINUS

`NOT EXISTS` and `MINUS` represent two ways of thinking about negation, one based on testing whether a pattern exists in the data, given the bindings already determined by the query pattern, and one based on removing matches based on the evaluation of two patterns. In some cases they can produce different answers.

8.3.1 Example: Sharing of variables

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

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

evaluates to a result set with no solutions because `{ ?x ?y ?z }` matches given any `?s ?p ?o`, so `NOT EXISTS { ?x ?y ?z }` eliminates any solutions.

s	p	o

whereas with `MINUS`, there is no shared variable between the first part (`?s ?p ?o`) and the second (`?x ?y ?z`) so no bindings are eliminated.

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

Results:

s	p	o
<http://example/a>	<http://example/b>	<http://example/c>

8.3.2 Example: Fixed pattern

Another case is where there is a concrete pattern (no variables) in the example:

```
PREFIX : <http://example/>
SELECT *
{
  ?s ?p ?o
  FILTER NOT EXISTS { :a :b :c }
}
```

evaluates to a result set with no query solutions:

Results:

s	p	o

whereas

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

evaluates to result set with one query solution:

Results:

s	p	o
<http://example/a>	<http://example/b>	<http://example/c>

because there is no match of bindings and so no solutions are eliminated.

8.3.3 Example: Inner FILTERs

Differences also arise because in a filter, variables from the group are in scope. In this example, the `FILTER` inside the `NOT EXISTS` has access to the value of ?n for the solution being considered.

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

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

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

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

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

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

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

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

9 Property Paths

A property path is a possible route through a graph between two graph nodes. A trivial case is a property path of length exactly 1, which is a triple pattern. The ends of the path may be RDF terms or variables. Variables can not be used as part of the path itself, only the ends.

Property paths allow for more concise expressions for some SPARQL basic graph patterns and they also add the ability to match connectivity of two resources by an arbitrary length path.

9.1 Property Path Syntax

In the description below, *iri* is either an IRI written in full or abbreviated by a prefixed name, or the keyword `a`. *elt* is a path element, which may itself be composed of path constructs.

Syntax Form	Property Path Expression Name	Matches
iri	PredicatePath	An IRI. A path of length one.
`^`*elt*	InversePath	Inverse path (object to subject).
elt1 `/` *elt2*	SequencePath	A sequence path of *elt1* followed by *elt2*.
elt1 `|` *elt2*	AlternativePath	A alternative path of *elt1* or *elt2* (all possibilities are tried).
elt`*`	ZeroOrMorePath	A path that connects the subject and object of the path by zero or more matches of *elt*.
elt`+`	OneOrMorePath	A path that connects the subject and object of the path by one or more matches of *elt*.
elt`?`	ZeroOrOnePath	A path that connects the subject and object of the path by zero or one matches of *elt*.
`!`*iri* or `!(`iri_1`|` ...`|`iri_n`)`	NegatedPropertySet	Negated property set. An IRI which is not one of iri_i. `!`*iri* is short for `!(`*iri*`)`.
`!^`*iri* or `!(^`iri_1`|` ...`|^`iri_n`)`	NegatedPropertySet	Negated property set where the excluded matches are based on reversed path. That is, not one of iri_1...iri_n as reverse paths. `!^`*iri* is short for `!(^`*iri*`)`.
`!(`iri_1`|` ...`|`iri_j`|^`iri_{j+1}`|` ...`|^`iri_n`)`	NegatedPropertySet	A combination of forward and reverse properties in a negated property set.
`(`*elt*`)`		A group path *elt*, brackets control precedence.

The order of IRIs, and reverse IRIs, in a negated property set is not significant and they can occur in a mixed order.

The precedence of the syntax forms is, from highest to lowest:

- IRI, prefixed names
- Negated property sets
- Groups
- Unary operators `*`, `?` and `+`
- Unary ^ inverse links
- Binary operator `/`
- Binary operator `|`

Precedence is left-to-right within groups.

9.2 Examples

Alternatives: Match one or both possibilities

```
{ :book1 dc:title|rdfs:label ?displayString }
```

which could have writen:

```
{ :book1 <http://purl.org/dc/elements/1.1/title> | <http://www.w3.org/2000/01/rdf-schema#label> ?displayString }
```

Sequence: Find the name of any people that Alice knows.

```
{
  ?x foaf:mbox <mailto:alice@example> .
  ?x foaf:knows/foaf:name ?name .
}
```

Sequence: Find the names of people 2 "`foaf:knows`" links away.

```
{
  ?x foaf:mbox <mailto:alice@example> .
  ?x foaf:knows/foaf:knows/foaf:name ?name .
}
```

This is the same as the SPARQL query:

```
SELECT ?x ?name
{
   ?x  foaf:mbox <mailto:alice@example> .
   ?x  foaf:knows [ foaf:knows [ foaf:name ?name ]].
}
```

or, with explicit variables:

```
SELECT ?x ?name
{
  ?x  foaf:mbox <mailto:alice@example> .
  ?x  foaf:knows ?a1 .
  ?a1 foaf:knows ?a2 .
  ?a2 foaf:name ?name .
}
```

Filtering duplicates: Because someone Alice knows may well know Alice, the example above may include Alice herself. This could be avoided with:

```
{ ?x foaf:mbox <mailto:alice@example> .
  ?x foaf:knows/foaf:knows ?y .
  FILTER ( ?x != ?y )
  ?y foaf:name ?name
}
```

Inverse Property Paths: These two are the same query: the second is just reversing the property direction which swaps the roles of subject and object.

```
{ ?x foaf:mbox <mailto:alice@example> }
```

```
{ <mailto:alice@example> ^foaf:mbox ?x }
```

Inverse Path Sequence: Find all the people who know someone `?x` knows.

```
{
  ?x foaf:knows/^foaf:knows ?y .
  FILTER(?x != ?y)
}
```

which is equivalent to (`?gen1` is a system generated variable):

```
{
  ?x foaf:knows ?gen1 .
  ?y foaf:knows ?gen1 .
  FILTER(?x != ?y)
}
```

Arbitrary length match: Find the names of all the people that can be reached from Alice by `foaf:knows`:

```
{
  ?x foaf:mbox <mailto:alice@example> .
  ?x foaf:knows+/foaf:name ?name .
}
```

Alternatives in an arbitrary length path:

```
{ ?ancestor (ex:motherOf|ex:fatherOf)+ <#me> }
```

Arbitrary length path match: Some forms of limited inference are possible as well. For example, for RDFS, all types and supertypes of a resource:

```
{ <http://example/thing> rdf:type/rdfs:subClassOf* ?type }
```

All resources and all their inferred types:

```
{ ?x rdf:type/rdfs:subClassOf* ?type }
```

Subproperty:

```
{ ?x ?p ?v . ?p rdfs:subPropertyOf* :property }
```

Negated Property Paths: Find nodes connected but not by rdf:type (either way round):

```
{ ?x !(rdf:type|^rdf:type) ?y }
```

Elements in an RDF collection:

```
{ :list rdf:rest*/rdf:first ?element }
```

Note: This path expression does not guarantee the order of the results.

9.3 Property Paths and Equivalent Patterns

SPARQL property paths treat the RDF triples as a directed, possibly cyclic, graph with named edges. Some property paths are equivalent to a translation into triple patterns and SPARQL UNION graph patterns. Evaluation of a property path expression can lead to duplicates because any variables introduced in the equivalent pattern are not part of the results and are not already used elsewhere. They are hidden by implicit projection of the results to just the variables given in the query.

For example, on the data:

```
@prefix :       <http://example/> .

:order  :item :z1 .
:order  :item :z2 .

:z1 :name "Small" .
:z1 :price 5 .

:z2 :name "Large" .
:z2 :price 5 .
```

Query:

```
PREFIX :   <http://example/>
SELECT *
{  ?s :item/:price ?x . }
```

Results:

s	x
<http://example/order>	5
<http://example/order>	5

whereas if the query were written out to include the intermediate variable (`?_a`), no rows in the results are duplicates:

```
PREFIX :   <http://example/>
SELECT *
{  ?s :item ?_a .
   ?_a :price ?x . }
```

Results:

s	_a	x
<http://example/order>	<http://example/z1>	5
<http://example/order>	<http://example/z2>	5

The equivalance to graphs patterns is particularly significant when query also involves an aggregation operation. The total cost of the order can be found with

```
PREFIX :   <http://example/>
SELECT (sum(?x) AS ?total)
{
  :order :item/:price ?x
}
```

total
10

9.4 Arbitrary Length Path Matching

Connectivity between the subject and object by a property path of arbitrary length can be found using the "zero or more" property path operator, `*`, and the "one or more" property path operator, `+`. There is also a "zero or one" connectivity property path operator, `?`.

Each of these operators uses the property path expression to try to find a connection between subject and object, using the path step a number of times, as restricted by the operator.

For example, finding all the the possible types of a resource, including supertypes of resources, can be achieved with:

```
PREFIX  rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
PREFIX  rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?x ?type
{
  ?x rdf:type/rdfs:subClassOf* ?type
}
```

Similarly, finding all the people `:x` connects to via the `foaf:knows` relationship,

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX :     <http://example/>
SELECT ?person
{
  :x foaf:knows+ ?person
}
```

Such connectivity matching does not introduce duplicates (it does not incorporate any count of the number of ways the connection can be made) even if the repeated path itself would otherwise result in duplicates.

The graph matched may include cycles. Connectivity matching is defined so that matching cycles does not lead to undefined or infinite results.

10 Assignment

The value of an expression can be added to a solution mapping by binding a new variable to the value of the expression, which is an RDF term. The variable can then be used in the query and also can be returned in results.

Three syntax forms allow this: the BIND keyword, expressions in the SELECT clause and expressions in the GROUP BY clause. The assignment form is (*expression* AS ?var).

If the evaluation of the expression produces an error, the variable remains unbound for that solution but the query evaluation continues.

Data can also be directly included in a query using VALUES for inline data.

10.1 BIND: Assigning to Variables

The BIND form allows a value to be assigned to a variable from a basic graph pattern or property path expression. Use of BIND ends the preceding basic graph pattern. The variable introduced by the BIND clause must not have been used in the group graph pattern up to the point of use in BIND.

Example:

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 .
:book1  ns:discount  0.2 .

:book2  dc:title     "The Semantic Web" .
:book2  ns:price     23 .
:book2  ns:discount  0.25 .
```

Query:

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

SELECT  ?title ?price
{  ?x ns:price ?p .
   ?x ns:discount ?discount
   BIND (?p*(1-?discount) AS ?price)
   FILTER(?price < 20)
   ?x dc:title ?title .
}
```

Equivalent query (BIND ends the basic graph pattern; the FILTER applies to the whole group graph pattern):

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

SELECT  ?title ?price
{  { ?x ns:price ?p .
     ?x ns:discount ?discount
     BIND (?p*(1-?discount) AS ?price)
   }
   {?x dc:title ?title . }
   FILTER(?price < 20)
}
```

Results:

title	price
"The Semantic Web"	17.25

10.2 VALUES: Providing inline data

Data can be directly written in a graph pattern or added to a query using VALUES. VALUES provides inline data as a solution sequence which are combined with the results of query evaluation by a join operation. It can be used by an application to provide specific requirements on query results and also by SPARQL query engine implementations that provide federated query through the SERVICE keyword to send a more constrained query to a remote query service.

10.2.1 VALUES syntax

VALUES allows multiple variables to be specified in the data block; there is a special syntax for the common case of specifying just one variable and some values.

In the following example, there is a table of two variables, ?x and ?y. The second row has no value for ?y.

```
VALUES (?x ?y) {
  (:uri1 1)
  (:uri2 UNDEF)
}
```

Optionally, when there is a single variable and some values:

```
VALUES ?z { "abc" "def" }
```

which is the same as using the general form:

```
VALUES (?z) { ("abc") ("def") }
```

10.2.2 VALUES Examples

A VALUES block of data can appear in a query pattern or at the end of a SELECT query, including a subquery.

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 .
```

Query:

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

SELECT ?book ?title ?price
{
   VALUES ?book { :book1 :book3 }
   ?book dc:title ?title ;
         ns:price ?price .
}
```

Result:

book	title	price
<http://example.org/book/book1>	"SPARQL Tutorial"	42

If a variable has no value for a particular solution in the VALUES clause, the keyword UNDEF is used instead of an RDF term.

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

SELECT ?book ?title ?price
{
   ?book dc:title ?title ;
         ns:price ?price .
   VALUES (?book ?title)
   { (UNDEF "SPARQL Tutorial")
     (:book2 UNDEF)
   }
}
```

book	title	price
<http://example.org/book/book1>	"SPARQL Tutorial"	42
<http://example.org/book/book2>	"The Semantic Web"	23

In this example, the VALUES might have been specified to execute over the results of the SELECT query:

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

SELECT ?book ?title ?price
{
   ?book dc:title ?title ;
         ns:price ?price .
}
VALUES (?book ?title)
{ (UNDEF "SPARQL Tutorial")
  (:book2 UNDEF)
}
```

This is a different query but, in the example situation, has the same results.

11 Aggregates

Aggregates apply expressions over groups of solutions. By default a solution set consists of a single group, containing all solutions.

Grouping may be specified using the GROUP BY syntax.

Aggregates defined in version 1.1 of SPARQL are COUNT, SUM, MIN, MAX, AVG, GROUP_CONCAT, and SAMPLE.

Aggregates are used where the querier wishes to see a result which is computed over a group of solutions, rather than a single solution. For example the maximum value that a particular variable takes, rather than each value individually.

11.1 Aggregate Example

Data:

```
@prefix : <http://books.example/> .

:org1 :affiliates :auth1, :auth2 .
:auth1 :writesBook :book1, :book2 .
:book1 :price 9 .
:book2 :price 5 .
:auth2 :writesBook :book3 .
:book3 :price 7 .
:org2 :affiliates :auth3 .
:auth3 :writesBook :book4 .
:book4 :price 7 .
```

Query:

```
PREFIX : <http://books.example/>
SELECT (SUM(?lprice) AS ?totalPrice)
WHERE {
  ?org :affiliates ?auth .
  ?auth :writesBook ?book .
  ?book :price ?lprice .
}
GROUP BY ?org
HAVING (SUM(?lprice) > 10)
```

Results:

totalPrice
21

This example demonstrates two features of aggregates: `GROUP BY`, which groups query solutions according to one or more expressions (in this case `?org`), and `HAVING`, which is analogous to a `FILTER` expression, but operates over groups, rather than individual solutions.

The example is produced by grouping solutions according to the `GROUP BY` expression (i.e. all solutions where `?org` takes a particular value appear within the same group), and evaluating the Set Function `SUM` over that group. The groups are then filtered by the `HAVING` expression, which removes all groups where `SUM(?lprice)` is not greater than 10.

In aggregate queries and sub-queries, variables that appear in the query pattern, but are not in the `GROUP BY` clause, can only be projected or used in select expressions if they are aggregated. The `SAMPLE` aggregate may be used for this purpose. For details see the section on Projection Restrictions.

It should be noted that as per functions, aggregate expressions are required to be aliased (again, similar to the `BIND` clause, using the keyword `AS`) in order to project them from queries or subqueries. In the example above this is done using the variable `?totalPrice`. It is an error for aggregates to project variables with a name already used in other aggregate projections, or in the `WHERE` clause.

11.2 GROUP BY

In order to calculate aggregate values for a solution, the solution is first divided into one or more groups, and the aggregate value is calculated for each group.

If aggregates are used in the query level in `SELECT`, `HAVING` or `ORDER BY` but the `GROUP BY` term is not used, then this is taken to be a single implicit group, to which all solutions belong.

Within `GROUP BY` clauses the binding keyword, `AS`, may be used, such as `GROUP BY (?x + ?y AS ?z)`. This is equivalent to `{ ... BIND (?x + ?y AS ?z) } GROUP BY ?z`.

For example, given a solution sequence S, ({?x→2, ?y→3}, {?x→2, ?y→5}, {?x→6, ?y→7}), we might wish to group the solutions according to the value of ?x, and calculate the average of the values of ?y for each group.

This could be written as:

```
SELECT (AVG(?y) AS ?avg)
WHERE {
  ?a :x ?x ;
     :y ?y .
}
GROUP BY ?x
```

11.3 HAVING

`HAVING` operates over grouped solution sets, in the same way that `FILTER` operates over un-grouped ones.

`HAVING` expressions have the same evaluation rules as projections from grouped queries, as described in the following section.

An example of the use of `HAVING` is given below.

```
PREFIX : <http://data.example/>
SELECT (AVG(?size) AS ?asize)
WHERE {
  ?x :size ?size
}
GROUP BY ?x
HAVING(AVG(?size) > 10)
```

This will return average sizes, grouped by the subject, but only where the mean size is greater than 10.

11.4 Aggregate Projection Restrictions

In a query level which uses aggregates, only expressions consisting of aggregates and constants may be projected, with one exception. When `GROUP BY` is given with one or more simple expressions consisting of just a variable, those variables may be projected from the level.

For example, the following query is legal as ?x is given as a `GROUP BY` term.

```
PREFIX : <http://example.com/data/#>
SELECT ?x (MIN(?y) * 2 AS ?min)
WHERE {
  ?x :p ?y .
  ?x :q ?z .
} GROUP BY ?x (STR(?z))
```

Note that it would not be legal to project `STR(?z)` as this is not a simple variable expression. However, with `GROUP BY (STR(?z) AS ?strZ)` it would be possible to project `?strZ`.

Other expressions, not using `GROUP BY` variables, or aggregates may have non-deterministic values projected from their groups using the `SAMPLE` aggregate.

11.5 Aggregate Example (with errors)

This section shows an example query using aggregation, which demonstrates how errors are handled in results, in the presence of aggregates.

Data:

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

:x :p 1, 2, 3, 4 .
:y :p 1, _:b2, 3, 4 .
:z :p 1.0, 2.0, 3.0, 4 .
```

Query:

```
PREFIX : <http://example.com/data/#>
SELECT ?g (AVG(?p) AS ?avg) ((MIN(?p) + MAX(?p)) / 2 AS ?c)
WHERE {
  ?g :p ?p .
}
GROUP BY ?g
```

Result:

g	avg	c
<http://example.com/data/#x>	2.5	2.5
<http://example.com/data/#y>		
<http://example.com/data/#z>	2.5	2.5

Note that the bindings for the :y group is not included in the results as the evaluation of Avg({1, _:b2, 3, 4}), and (_:b2 + 4) / 2 is an error, removing the bindings from the solution.

12 Subqueries

Subqueries are a way to embed SPARQL queries within other queries, normally to achieve results which cannot otherwise be achieved, such as limiting the number of results from some sub-expression within the query.

Due to the bottom-up nature of SPARQL query evaluation, the subqueries are evaluated logically first, and the results are projected up to the outer query.

Note that only variables projected out of the subquery will be visible, or in scope, to the outer query.

Example

Data:

```
@prefix : <http://people.example/> .

:alice :name "Alice", "Alice Foo", "A. Foo" .
:alice :knows :bob, :carol .
:bob :name "Bob", "Bob Bar", "B. Bar" .
:carol :name "Carol", "Carol Baz", "C. Baz" .
```

Return a name (the one with the lowest sort order) for all the people that know Alice and have a name.

Query:

```
PREFIX : <http://people.example/>
PREFIX : <http://people.example/>
SELECT ?y ?minName
WHERE {
  :alice :knows ?y .
  {
    SELECT ?y (MIN(?name) AS ?minName)
    WHERE {
      ?y :name ?name .
    } GROUP BY ?y
  }
}
```

Results:

y	minName
:bob	"B. Bar"
:carol	"C. Baz"

This result is achieved by first evaluating the inner query:

```
SELECT ?y (MIN(?name) AS ?minName)
WHERE {
  ?y :name ?name .
} GROUP BY ?y
```

This produces the following solution sequence:

y	minName
:alice	"A. Foo"
:bob	"B. Bar"
:carol	"C. Baz"

Which is joined with the results of the outer query:

y
:bob
:carol

13 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 13.3 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.

13.1 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.

13.2 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.

13.2.1 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 (located 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 the RDF merge of the graphs obtained from representations of the resources identified by the given IRIs.

13.2.2 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].

13.2.3 Combining FROM and FROM NAMED

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

```
# Default graph (located 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.

13.3 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 graph patterns within that part of the query. Outside the use of GRAPH, matching is done using the default graph.

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 .
```

13.3.1 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"

13.3.2 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"

13.3.3 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.

13.3.4 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

14 Basic Federated Query

This document incorporates the syntax for SPARQL federation extensions.

This feature is defined in the document SPARQL 1.1 Federated Query.

15 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-distinct 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.

15.1 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/>

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

```
PREFIX     :    <http://example.org/ns#>
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 17.3.1 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 "1"^^xsd:integer (a simple literal and a literal with a supported datatype)
- "1"^^my:integer and "2"^^my:integer (two unsupported datatypes)
- "1"^^xsd:integer and "2"^^my:integer (a supported datatype and an unsupported datatype)

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.

RDF Term	Reason
"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.

15.2 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"

15.3 Duplicate Solutions

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

Data:

```
@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> .
```

Query:

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

Results:

name
"Alice"
"Alice"
"Alice"

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

15.3.1 DISTINCT

The DISTINCT solution modifier eliminates duplicate solutions. Only one solution solution that binds the same variables to the same RDF terms is returned from the query.

```
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.

15.3.2 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 a 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"

15.4 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
```

15.5 LIMIT

The LIMIT clause puts an upper bound on the number of solutions returned. If the number of actual solutions, after OFFSET is applied, 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.

16 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.

Formats such as SPARQL 1.1 Query Results JSON Format, SPARQL Query Results XML Format or SPARQL 1.1 Query Results CSV and TSV Formats can be used to serialize the result set from a SELECT query or the boolean result of an ASK query.

16.1 SELECT

The SELECT form of results returns variables and their bindings directly. It combines the operations of projecting the required variables with introducing new variable bindings into a query solution.

16.1.1 Projection

Specific variables and their bindings are returned when a list of variable names is given in the SELECT clause. The syntax SELECT * is an abbreviation that selects all of the variables that are in-scope at that point in the query. It excludes variables only used in FILTER, in the right-hand side of MINUS, and takes account of subqueries.

Use of SELECT * is only permitted when the query does not have a GROUP BY clause.

```
@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 JSON, XML, CSV or TSV.

SPARQL 1.1 Query Results JSON Format:

```
{
  "head": {
    "vars": [ "nameX" , "nameY" , "nickY" ]
  } ,
  "results": {
    "bindings": [
      {
        "nameX": { "type": "literal" , "value": "Alice" } ,
        "nameY": { "type": "literal" , "value": "Bob" }
      } ,
      {
        "nameX": { "type": "literal" , "value": "Alice" } ,
        "nameY": { "type": "literal" , "value": "Clare" } ,
        "nickY": { "type": "literal" , "value": "CT" }
      }
    ]
  }
}
```

SPARQL Query Results XML Format:

```
<?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>
```

16.1.2 SELECT Expressions

As well as choosing which variables from the pattern matching are included in the results, the SELECT clause can also introduce new variables. The rules of assignment in SELECT expression are the same as for assignment in BIND. The expression combines variable bindings already in the query solution, or defined earlier in the SELECT clause, to produce a binding in the query solution.

The scoping for `(expr AS v)` applies immediately. In `SELECT` expressions, the variable may be used in an expression later in the same `SELECT` clause and may not be be assigned again in the same `SELECT` clause.

Example:

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 .
:book1  ns:discount 0.2 .

:book2  dc:title  "The Semantic Web" .
:book2  ns:price  23 .
:book2  ns:discount 0.25 .
```

Query:

```
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
PREFIX  ns:  <http://example.org/ns#>
SELECT  ?title (?p*(1-?discount) AS ?price)
{ ?x ns:price ?p .
  ?x dc:title ?title .
  ?x ns:discount ?discount
}
```

Results:

title	price
"The Semantic Web"	17.25
"SPARQL Tutorial"	33.6

New variables can also be used in expressions if they are introduced earlier, syntactically, in the same SELECT clause:

```
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
PREFIX  ns:  <http://example.org/ns#>
SELECT  ?title (?p AS ?fullPrice) (?fullPrice*(1-?discount) AS ?customerPrice)
{ ?x ns:price ?p .
   ?x dc:title ?title .
   ?x ns:discount ?discount
}
```

Results:

title	fullPrice	customerPrice
"The Semantic Web"	23	17.25
"SPARQL Tutorial"	42	33.6

16.2 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" .
```

16.2.1 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.

16.2.2 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#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

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` identifies an extension function to turn the date format into an `xsd:dateTime` RDF term.

16.2.3 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" .
```

16.2.4 CONSTRUCT WHERE

A short form for the CONSTRUCT query form is provided for the case where the template and the pattern are the same and the pattern is just a basic graph pattern (no `FILTER`s and no complex graph patterns are allowed in the short form). The keyword `WHERE` is required in

the short form.

The following two queries are the same; the first is a short form of the second.

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

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

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

16.3 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" }
```

```
true
```

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>
  <boolean>true</boolean>
</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> }
```

```
false
```

16.4 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.

16.4.1 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/>
```

16.4.2 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 functional 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}
```

16.4.3 Descriptions of Resources

The RDF returned is determined by the information publisher. It may be information the service deems relevant to the resources being described. 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   "bee135d3af1e418104bc42904596fe148e90f033" ;
        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.

17 Expressions and Testing Values

SPARQL `FILTER`s restrict the solutions of a graph pattern match according to a given constraint. Specifically, `FILTER`s 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 17.2.2 *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 effect 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 17.3 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 17.5. 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") ) ...
```

Typographical convention in this section: XPath operators are labeled with the prefix `op:`. XPath operators have no namespace; `op:` is a labeling convention.

17.1 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 datatypes.

17.2 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 17.2.2.
- Apart from BOUND, COALESCE, NOT EXISTS and EXISTS, 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

17.2.1 Invocation

SPARQL defines a syntax for invoking functions on a list of arguments. Unless otherwise noted, 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.

There are also "functional forms" which have different evaluation rules to functions as specified by each such form.

17.2.2 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`, and it has a valid lexical form, 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, and it has a valid lexical form, 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".

17.3 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 17.4. 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 candidate 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 definitions 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 Binary Operators

Operator	Type(A)	Type(B)	Function	Result type
Logical Connectives				
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				
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

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

17.3.1 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 `FILTER`s will produce *at least* the same intermediate bindings after applying a `FILTER` as an unextended implementation.

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

17.4 Function Definitions

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

17.4.1 Functional Forms

17.4.1.1 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 ?givenName
 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.

17.4.1.2 IF

```
rdfTerm  IF (expression1, expression2, expression3)
```

The `IF` function form evaluates the first argument, interprets it as a effective boolean value, then returns the value of `expression2` if the EBV is true, otherwise it returns the value of `expression3`. Only one of `expression2` and `expression3` is evaluated. If evaluating the first argument raises an error, then an error is raised for the evaluation of the `IF` expression.

Examples: Suppose ?x = 2, ?z = 0 and ?y is not bound in some query solution:

IF(?x = 2, "yes", "no")	returns "yes"
IF(bound(?y), "yes", "no")	returns "no"
IF(?x=2, "yes", 1/?z)	returns "yes", the expression 1/?z is not evaluated
IF(?x=1, "yes", 1/?z)	raises an error
IF("2" > 1, "yes", "no")	raises an error

17.4.1.3 COALESCE

```
rdfTerm  COALESCE(expression, ....)
```

The `COALESCE` function form returns the RDF term value of the first expression that evaluates without error. In SPARQL, evaluating an unbound variable raises an error.

If none of the arguments evaluates to an RDF term, an error is raised. If no expressions are evaluated without error, an error is raised.

Examples: Suppose ?x = 2 and ?y is not bound in some query solution:

COALESCE(?x, 1/0)	returns 2, the value of x
COALESCE(1/0, ?x)	returns 2
COALESCE(5, ?x)	returns 5
COALESCE(?y, 3)	returns 3
COALESCE(?y)	raises an error because y is not bound.

17.4.1.4 NOT EXISTS and EXISTS

There is a filter operator `EXISTS` that takes a graph pattern. `EXISTS` returns `true`/`false` depending on whether the pattern matches the dataset given the bindings in the current group graph pattern, the dataset and the active graph at this point in the query evaluation. No additional binding of variables occurs. The `NOT EXISTS` form translates into `fn:not(EXISTS{...})`.

```
 xsd:boolean  NOT EXISTS { pattern }
```

Returns `false` if `pattern` matches. Returns true otherwise.

`NOT EXISTS { pattern }` is equivalent to `fn:not(EXISTS { pattern })`.

```
xsd:boolean  EXISTS { pattern }
```

Returns `true` if `pattern` matches. Returns false otherwise.

Variables in the `pattern` that are bound in the current solution mapping take the value that they have from the solution mapping. Variables in the pattern `pattern` that are not bound in the current solution mapping take part in pattern matching.

To facilitate this, we introduce a function Exists that evaluates a SPARQL Algebra expression and returns true or false, depending on whether there are any solutions to the pattern, given the solution mapping being tested by the filter operation.

17.4.1.5 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 17.2, Filter Evaluation, for the `||` operator's treatment of errors.

17.4.1.6 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 17.2, Filter Evaluation, for the `&&` operator's treatment of errors.

17.4.1.7 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 at a specific date and time (New Year's Day 2005, measures in timezone +00:00), 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`.

17.4.1.8 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 datatypes:

```
@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.

17.4.1.9 IN

```
boolean  rdfTerm IN (expression, ...)
```

The `IN` operator tests whether the RDF term on the left-hand side is found in the values of list of expressions on the right-hand side. The test is done with "=" operator, which tests for the same value, as determined by the operator mapping.

A list of zero terms on the right-hand side is legal.

Errors in comparisons cause the `IN` expression to raise an error if the RDF term being tested is not found elsewhere in the list of terms.

The `IN` operator is equivalent to the SPARQL expression:

```
(lhs = expression1) || (lhs = expression2) || ...
```

Examples:

`2 IN (1, 2, 3)`	`true`
`2 IN ()`	`false`
`2 IN (<http://example/iri>, "str", 2.0)`	`true`
`2 IN (1/0, 2)`	`true`
`2 IN (2, 1/0)`	`true`
`2 IN (3, 1/0)`	`raises an error`

17.4.1.10 NOT IN

```
boolean  rdfTerm NOT IN (expression, ...)
```

The `NOT IN` operator tests whether the RDF term on the left-hand side is not found in the values of list of expressions on the right-hand side. The test is done with "!=" operator, which tests for not the same value, as determined by the operator mapping.

A list of zero terms on the right-hand side is legal.

Errors in comparisons cause the `NOT IN` expression to raise an error if the RDF term being tested is not found to be in the list elsewhere in the list of terms.

The `NOT IN` operator is equivalent to the SPARQL expression:

```
(lhs != expression1) && (lhs != expression2) && ...
```

`NOT IN (...)` is equivalent to `!(IN (...))`.

Examples:

`2 NOT IN (1, 2, 3)`	`false`
`2 NOT IN ()`	`true`
`2 NOT IN (<http://example/iri>, "str", 2.0)`	`false`
`2 NOT IN (1/0, 2)`	`false`
`2 NOT IN (2, 1/0)`	`false`
`2 NOT IN (3, 1/0)`	`raises an error`

17.4.2 Functions on RDF Terms

17.4.2.1 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>

17.4.2.2 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 `dc:creator` predicates, but only one (`_:c`) was a blank node.

17.4.2.3 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 17.4.2.1 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"

17.4.2.4 isNumeric

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

Returns `true` if `term` is a numeric value. Returns `false` otherwise. `term` is numeric if it has an appropriate datatype (see the section Operand Data Types) and has a valid lexical form, making it a valid argument to functions and operators taking numeric arguments.

Examples:

isNumeric(12)	true
isNumeric("12")	false
isNumeric("12"^^xsd:nonNegativeInteger)	true
isNumeric("1200"^^xsd:byte)	false
isNumeric(<http://example/>)	false

17.4.2.5 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>

17.4.2.6 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>

17.4.2.7 datatype

```
iri  DATATYPE (literal literal)
```

Returns the datatype IRI of a `literal`.

- If the literal is a typed literal, return the datatype IRI.
- If the literal is a simple literal, return `xsd:string`
- If the literal is literal with a language tag, return `rdf:langString`

```
@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

In SPARQL 1.0, the `DATATYPE` function was not defined for literals with a language tag. Therefore, an unextended implementation would raise an error when `DATATYPE` was called with a literal with a language tag. Operator extensibility allows implementations to return a result rather than raise an error. SPARQL 1.1 defines the result of `DATATYPE` applied to a literal with a language tag to be `rdf:langString`.

> The SPARQL Working Group is using `rdf:langString` based on the latest Working Drafts of the RDF Working Group. This usage should be considered experimental (and non-normative) until/unless `rdf:langString` becomes part of an updated RDF Recommendation.

17.4.2.8 IRI

```
iri  IRI(simple literal)
iri  IRI(xsd:string)
iri  IRI(iri)
iri  URI(simple literal)
iri  URI(xsd:string)
iri  URI(iri)
```

The `IRI` function constructs an IRI by resolving the string argument (see RFC 3986 and RFC 3987 or any later RFC that superceeds RFC 3986 or RFC 3987). The IRI is resolved against the base IRI of the query and must result in an absolute IRI.

The `URI` function is a synonym for `IRI`.

If the function is passed an IRI, it returns the IRI unchanged.

Passing any RDF term other than a simple literal, xsd:string or an IRI is an error.

An implementation MAY normalize the IRI.

Examples:

IRI("http://example/")	<http://example/>
IRI(<http://example/>)	<http://example/>

17.4.2.9 BNODE

```
blank node  BNODE()
```

```
blank node  BNODE(simple literal)
```

```
blank node  BNODE(xsd:string)
```

The `BNODE` function constructs a blank node that is distinct from all blank nodes in the dataset being queried and distinct from all blank nodes created by calls to this constructor for other query solutions. If the no argument form is used, every call results in a distinct blank node. If the form with a simple literal is used, every call results in distinct blank nodes for different simple literals, and the same blank node for calls with the same simple literal within expressions for one solution mapping.

This functionality is compatible with the treatment of blank nodes in SPARQL CONSTRUCT templates.

17.4.2.10 STRDT

```
literal  STRDT(simple literal lexicalForm, IRI datatypeIRI)
```

The `STRDT` function constructs a literal with lexical form and type as specified by the arguments.

STRDT("123", xsd:integer)	"123"^^<http://www.w3.org/2001/XMLSchema#integer>

STRDT("iiii", <http://example/romanNumeral>)	"iiii"^^<http://example/romanNumeral>

17.4.2.11 STRLANG

```
literal  STRLANG(simple literal lexicalForm, simple literal langTag)
```

The `STRLANG` function constructs a literal with lexical form and language tag as specified by the arguments.

STRLANG("chat", "en")	"chat"@en

17.4.2.12 UUID

```
iri  UUID()
```

Return a fresh IRI from the UUID URN scheme. Each call of `UUID()` returns a different UUID. It must not be the "nil" UUID (all zeroes). The variant and version of the UUID is implementation dependent.

UUID()	<urn:uuid:b9302fb5-642e-4d3b-af19-29a8f6d894c9>

17.4.2.13 STRUUID

```
simple literal  STRUUID()
```

Return a string that is the scheme specific part of UUID. That is, as a simple literal, the result of generating a UUID, converting to a simple literal and removing the initial `urn:uuid:`.

STRUUID()	"73cd4307-8a99-4691-a608-b5bda64fb6c1"

17.4.3 Functions on Strings

17.4.3.1 Strings in SPARQL Functions

17.4.3.1.1 String arguments

Certain functions (e.g. REGEX, STRLEN, CONTAINS) take a `string literal` as an argument and accept a simple literal, a plain literal with language tag, or a literal with datatype xsd:string. They then act on the lexcial form of the literal.

The term `string literal` is used in the function descriptions for this. Use of any other RDF term will cause a call to the function to raise an error.

17.4.3.1.2 Argument Compatibility Rules

The functions STRSTARTS, STRENDS, CONTAINS, STRBEFORE and STRAFTER take two arguments. These arguments must be compatible otherwise invocation of one of these functions raises an error.

Compatibility of two arguments is defined as:

- The arguments are simple literals or literals typed as xsd:string
- The arguments are plain literals with identical language tags
- The first argument is a plain literal with language tag and the second argument is a simple literal or literal typed as xsd:string

Argument1	Argument2	Compatible?
"abc"	"b"	yes
"abc"	"b"^^xsd:string	yes
"abc"^^xsd:string	"b"	yes
"abc"^^xsd:string	"b"^^xsd:string	yes
"abc"@en	"b"	yes
"abc"@en	"b"^^xsd:string	yes
"abc"@en	"b"@en	yes
"abc"@fr	"b"@ja	no
"abc"	"b"@ja	no
"abc"	"b"@en	no
"abc"^^xsd:string	"b"@en	no

17.4.3.1.3 String Literal Return Type

Functions that return a string literal do so with the string literal of the same kind as the first argument (simple literal, plain literal with same language tag, xsd:string). This includes SUBSTR, STRBEFORE and STRAFTER.

The function CONCAT returns a string literal based on the details of all its arguments.

17.4.3.2 STRLEN

```
xsd:integer  STRLEN(string literal str)
```

The `strlen` function corresponds to the XPath fn:string-length function and returns an `xsd:integer` equal to the length in characters of the lexical form of the literal.

strlen("chat")	4
strlen("chat"@en)	4
strlen("chat"^^xsd:string)	4

17.4.3.3 SUBSTR

```
string literal  SUBSTR(string literal source, xsd:integer startingLoc)
```

```
string literal  SUBSTR(string literal source, xsd:integer startingLoc, xsd:integer length)
```

The `substr` function corresponds to the XPath fn:substring function and returns a literal of the same kind (simple literal, literal with language tag, `xsd:string` typed literal) as the `source` input parameter but with a lexical form formed from the substring of the lexcial form of the source.

The arguments `startingLoc` and `length` may be derived types of xsd:integer.

The index of the first character in a strings is 1.

substr("foobar", 4)	"bar"
substr("foobar"@en, 4)	"bar"@en
substr("foobar"^^xsd:string, 4)	"bar"^^xsd:string
substr("foobar", 4, 1)	"b"
substr("foobar"@en, 4, 1)	"b"@en
substr("foobar"^^xsd:string, 4, 1)	"b"^^xsd:string

17.4.3.4 UCASE

```
string literal  UCASE(string literal str)
```

The `UCASE` function corresponds to the XPath fn:upper-case function. It returns a string literal whose lexical form is the upper case of the lexcial form of the argument.

ucase("foo")	"FOO"
ucase("foo"@en)	"FOO"@en
ucase("foo"^^xsd:string)	"FOO"^^xsd:string

17.4.3.5 LCASE

```
string literal  LCASE(string literal str)
```

The `LCASE` function corresponds to the XPath fn:lower-case function. It returns a string literal whose lexical form is the lower case of the lexcial form of the argument.

lcase("BAR")	"bar"
lcase("BAR"@en)	"bar"@en
lcase("BAR"^^xsd:string)	"bar"^^xsd:string

17.4.3.6 STRSTARTS

```
xsd:boolean  STRSTARTS(string literal arg1, string literal arg2)
```

The `STRSTARTS` function corresponds to the XPath fn:starts-with function. The arguments must be argument compatible otherwise an error is raised.

For such input pairs, the function returns true if the lexical form of `arg1` starts with the lexical form of `arg2`, otherwise it returns false.

strStarts("foobar", "foo")	true
strStarts("foobar"@en, "foo"@en)	true
strStarts("foobar"^^xsd:string, "foo"^^xsd:string)	true

strStarts("foobar"^^xsd:string, "foo")	true
strStarts("foobar", "foo"^^xsd:string)	true
strStarts("foobar"@en, "foo")	true
strStarts("foobar"@en, "foo"^^xsd:string)	true

17.4.3.7 STRENDS

```
xsd:boolean  STRENDS(string literal arg1, string literal arg2)
```

The STRENDS function corresponds to the XPath fn:ends-with function. The arguments must be argument compatible otherwise an error is raised.

For such input pairs, the function returns true if the lexical form of arg1 ends with the lexical form of arg2, otherwise it returns false.

strEnds("foobar", "bar")	true
strEnds("foobar"@en, "bar"@en)	true
strEnds("foobar"^^xsd:string, "bar"^^xsd:string)	true
strEnds("foobar"^^xsd:string, "bar")	true
strEnds("foobar", "bar"^^xsd:string)	true
strEnds("foobar"@en, "bar")	true
strEnds("foobar"@en, "bar"^^xsd:string)	true

17.4.3.8 CONTAINS

```
xsd:boolean  CONTAINS(string literal arg1, string literal arg2)
```

The CONTAINS function corresponds to the XPath fn:contains. The arguments must be argument compatible otherwise an error is raised.

contains("foobar", "bar")	true
contains("foobar"@en, "foo"@en)	true
contains("foobar"^^xsd:string, "bar"^^xsd:string)	true
contains("foobar"^^xsd:string, "foo")	true
contains("foobar", "bar"^^xsd:string)	true
contains("foobar"@en, "foo")	true
contains("foobar"@en, "bar"^^xsd:string)	true

17.4.3.9 STRBEFORE

```
literal  STRBEFORE(string literal arg1, string literal arg2)
```

The STRBEFORE function corresponds to the XPath fn:substring-before function. The arguments must be argument compatible otherwise an error is raised.

For compatible arguments, if the lexical part of the second argument occurs as a substring of the lexical part of the first argument, the function returns a literal of the same kind as the first argument arg1 (simple literal, plain literal same language tag, xsd:string). The lexical form of the result is the substring of the lexical form of arg1 that precedes the first occurrence of the lexical form of arg2. If the lexical form of arg2 is the empty string, this is considered to be a match and the lexical form of the result is the empty string.

If there is no such occurrence, an empty simple literal is returned.

strbefore("abc","b")	"a"
strbefore("abc"@en,"bc")	"a"@en
strbefore("abc"@en,"b"@cy)	error
strbefore("abc"^^xsd:string,"")	""^^xsd:string
strbefore("abc","xyz")	""
strbefore("abc"@en, "z"@en)	""
strbefore("abc"@en, "z")	""
strbefore("abc"@en, ""@en)	""@en
strbefore("abc"@en, "")	""@en

17.4.3.10 STRAFTER

```
literal  STRAFTER(string literal arg1, string literal arg2)
```

The STRAFTER function corresponds to the XPath fn:substring-after function. The arguments must be argument compatible otherwise an error is raised.

For compatible arguments, if the lexical part of the second argument occurs as a substring of the lexical part of the first argument, the function returns a literal of the same kind as the first argument arg1 (simple literal, plain literal same language tag, xsd:string). The lexical form of the result is the substring of the lexcial form of arg1 that follows the first occurrence of the lexical form of arg2. If the lexical form of arg2 is the empty string, this is considered to be a match and the lexical form of the result is the lexical form of arg1.

If there is no such occurrence, an empty simple literal is returned.

strafter("abc","b")	"c"
strafter("abc"@en,"ab")	"c"@en
strafter("abc"@en,"b"@cy)	error
strafter("abc"^^xsd:string,"")	"abc"^^xsd:string
strafter("abc","xyz")	""
strafter("abc"@en, "z"@en)	""
strafter("abc"@en, "z")	""
strafter("abc"@en, ""@en)	"abc"@en
strafter("abc"@en, "")	"abc"@en

17.4.3.11 ENCODE_FOR_URI

```
simple literal  ENCODE_FOR_URI(string literal ltrl)
```

The ENCODE_FOR_URI function corresponds to the XPath fn:encode-for-uri function. It returns a simple literal with the lexical form obtained from the lexical form of its input after translating reserved characters according to the fn:encode-for-uri function.

encode_for_uri("Los Angeles")	"Los%20Angeles"
encode_for_uri("Los Angeles"@en)	"Los%20Angeles"
encode_for_uri("Los Angeles"^^xsd:string)	"Los%20Angeles"

17.4.3.12 CONCAT

```
string literal  CONCAT(string literal ltrl1 ... string literal ltrln)
```

The CONCAT function corresponds to the XPath fn:concat function. The function accepts string literals as arguments.

The lexical form of the returned literal is obtained by concatenating the lexical forms of its inputs. If all input literals are typed literals of type xsd:string, then the returned literal is also of type xsd:string, if all input literals are plain literals with identical language tag, then the returned literal is a plain literal with the same language tag, in all other cases, the returned literal is a simple literal.

concat("foo", "bar")	"foobar"
concat("foo"@en, "bar"@en)	"foobar"@en
concat("foo"^^xsd:string, "bar"^^xsd:string)	"foobar"^^xsd:string
concat("foo", "bar"^^xsd:string)	"foobar"
concat("foo"@en, "bar")	"foobar"
concat("foo"@en, "bar"^^xsd:string)	"foobar"

17.4.3.13 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 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

17.4.3.14 REGEX

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

17.4.3.15 REPLACE

```
string literal  REPLACE (string literal arg, simple literal pattern, simple literal replacement )
string literal  REPLACE (string literal arg, simple literal pattern, simple literal replacement,  simple literal flags)
```

The `REPLACE` function corresponds to the XPath fn:replace function. It replaces each non-overlapping occurrence of the regular expression `pattern` with the replacement string. Regular expression matching may involve modifier flags. See REGEX.

replace("abcd", "b", "Z")	"aZcd"
replace("abab", "B", "Z","i")	"aZaZ"
replace("abab", "B.", "Z","i")	"aZb"

17.4.4 Functions on Numerics

17.4.4.1 abs

```
numeric  ABS (numeric term)
```

Returns the absolute value of `arg`. An error is raised if `arg` is not a numeric value.

This function is the same as fn:numeric-abs for terms with a datatype from XDM.

abs(1)	1
abs(-1.5)	1.5

17.4.4.2 round

```
numeric  ROUND (numeric term)
```

Returns the number with no fractional part that is closest to the argument. If there are two such numbers, then the one that is closest to positive infinity is returned. An error is raised if `arg` is not a numeric value.

This function is the same as fn:numeric-round for terms with a datatype from XDM.

round(2.4999)	2.0
round(2.5)	3.0
round(-2.5)	-2.0

17.4.4.3 ceil

```
numeric  CEIL (numeric term)
```

Returns the smallest (closest to negative infinity) number with no fractional part that is not less than the value of `arg`. An error is raised if `arg` is not a numeric value.

This function is the same as fn:numeric-ceil for terms with a datatype from XDM.

ceil(10.5)	11.0
ceil(-10.5)	-10.0

17.4.4.4 floor

```
numeric  FLOOR (numeric term)
```

Returns the largest (closest to positive infinity) number with no fractional part that is not greater than the value of `arg`. An error is raised if `arg` is not a numeric value.

This function is the same as fn:numeric-floor for terms with a datatype from XDM.

floor(10.5)	10.0
floor(-10.5)	-11.0

17.4.4.5 RAND

```
xsd:double  RAND ( )
```

Returns a pseudo-random number between 0 (inclusive) and 1.0e0 (exclusive). Different numbers can be produced every time this function is invoked. Numbers should be produced with approximately equal probability.

rand()	"0.31221030831984886"^^xsd:double

17.4.5 Functions on Dates and Times

17.4.5.1 now

```
xsd:dateTime  NOW ()
```

Returns an XSD dateTime value for the current query execution. All calls to this function in any one query execution must return the same value. The exact moment returned is not specified.

now()	"2011-01-10T14:45:13.815-05:00"^^xsd:dateTime

17.4.5.2 year

```
xsd:integer  YEAR (xsd:dateTime arg)
```

Returns the year part of `arg` as an integer.

This function corresponds to fn:year-from-dateTime.

year("2011-01-10T14:45:13.815-05:00"^^xsd:dateTime)	2011

17.4.5.3 month

```
xsd:integer  MONTH (xsd:dateTime arg)
```

Returns the month part of `arg` as an integer.

This function corresponds to fn:month-from-dateTime.

month("2011-01-10T14:45:13.815-05:00"^^xsd:dateTime)	1

17.4.5.4 day

```
xsd:integer  DAY (xsd:dateTime arg)
```

Returns the day part of `arg` as an integer.

This function corresponds to fn:day-from-dateTime.

day("2011-01-10T14:45:13.815-05:00"^^xsd:dateTime)	10

17.4.5.5 hours

```
xsd:integer  HOURS (xsd:dateTime arg)
```

Returns the hours part of `arg` as an integer. The value is as given in the lexical form of the XSD dateTime.

This function corresponds to fn:hours-from-dateTime.

hours("2011-01-10T14:45:13.815-05:00"^^xsd:dateTime)	14

17.4.5.6 minutes

```
xsd:integer  MINUTES (xsd:dateTime arg)
```

Returns the minutes part of the lexical form of `arg`. The value is as given in the lexical form of the XSD dateTime.

This function corresponds to fn:minutes-from-dateTime.

minutes("2011-01-10T14:45:13.815-05:00"^^xsd:dateTime)	45

17.4.5.7 seconds

```
xsd:decimal  SECONDS (xsd:dateTime arg)
```

Returns the seconds part of the lexical form of `arg`.

This function corresponds to fn:seconds-from-dateTime.

seconds("2011-01-10T14:45:13.815-05:00"^^xsd:dateTime)	13.815

17.4.5.8 timezone

```
xsd:dayTimeDuration  TIMEZONE (xsd:dateTime arg)
```

Returns the timezone part of `arg` as an xsd:dayTimeDuration. Raises an error if there is no timezone.

This function corresponds to fn:timezone-from-dateTime except for the treatment of literals with no timezone.

timezone("2011-01-10T14:45:13.815-05:00"^^xsd:dateTime)	"-PT5H"^^xsd:dayTimeDuration
timezone("2011-01-10T14:45:13.815Z"^^xsd:dateTime)	"PT0S"^^xsd:dayTimeDuration
timezone("2011-01-10T14:45:13.815"^^xsd:dateTime)	error

17.4.5.9 tz

```
simple literal  TZ (xsd:dateTime arg)
```

Returns the timezone part of `arg` as a simple literal. Returns the empty string if there is no timezone.

tz("2011-01-10T14:45:13.815-05:00"^^xsd:dateTime)	"-05:00"
tz("2011-01-10T14:45:13.815Z"^^xsd:dateTime)	"Z"
tz("2011-01-10T14:45:13.815"^^xsd:dateTime)	""

17.4.6 Hash Functions

17.4.6.1 MD5

```
simple literal  MD5 (simple literal arg)
```

```
simple literal  MD5 (xsd:string arg)
```

Returns the MD5 checksum, as a hex digit string, calculated on the UTF-8 representation of the simple literal or lexical form of the `xsd:string`. Hex digits SHOULD be in lower case.

MD5("abc")	"900150983cd24fb0d6963f7d28e17f72"
MD5("abc"^^xsd:string)	"900150983cd24fb0d6963f7d28e17f72"

17.4.6.2 SHA1

```
simple literal  SHA1 (simple literal arg)
```

```
simple literal  SHA1 (xsd:string arg)
```

Returns the SHA1 checksum, as a hex digit string, calculated on the UTF-8 representation of the simple literal or lexical form of the `xsd:string`. Hex digits SHOULD be in lower case.

SHA1("abc")	"a9993e364706816aba3e25717850c26c9cd0d89d"
SHA1("abc"^^xsd:string)	"a9993e364706816aba3e25717850c26c9cd0d89d"

17.4.6.3 SHA256

```
simple literal  SHA256 (simple literal arg)
```

```
simple literal  SHA256 (xsd:string arg)
```

Returns the SHA256 checksum, as a hex digit string, calculated on the UTF-8 representation of the simple literal or lexical form of the `xsd:string`. Hex digits SHOULD be in lower case.

SHA256("abc")	"ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad"
SHA256("abc"^^xsd:string)	"ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad"

17.4.6.4 SHA384

```
simple literal  SHA384 (simple literal arg)
```

```
simple literal  SHA384 (xsd:string arg)
```

Returns the SHA384 checksum, as a hex digit string, calculated on the UTF-8 representation of the simple literal or lexical form of the `xsd:string`. Hex digits SHOULD be in lower case.

SHA384("abc")	"cb00753f45a35e8bb5a03d699ac65007272c32ab0eded1631a8b605a43ff5bed8086072ba1e7cc2358baeca134c825a7"
SHA384("abc"^^xsd:string)	"cb00753f45a35e8bb5a03d699ac65007272c32ab0eded1631a8b605a43ff5bed8086072ba1e7cc2358baeca134c825a7"

17.4.6.5 SHA512

```
simple literal  SHA512 (simple literal arg)
```

```
simple literal  SHA512 (xsd:string arg)
```

Returns the SHA512 checksum, as a hex digit string, calculated on the UTF-8 representation of the simple literal or lexical form of the `xsd:string`. Hex digits SHOULD be in lower case.

SHA512("abc")	"ddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f"
SHA512("abc"^^xsd:string)	"ddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f"

17.5 XPath 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 datatypes 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
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

17.6 Extensible Value Testing

It should be noted that any function or operator that is specified to return an error under some conditions is a valid extension point. That is, an implementation may return a non-error value in these error cases, and still be conformant with this recommendation.

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:locationX ?axLoc .
        ?a aGeo:locationY ?ayLoc .

        ?b aGeo:placeName ?neighbor .
        ?b aGeo:locationX ?bxLoc .
        ?b aGeo:locationY ?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.

18 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 query 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.

18.1 Initial Definitions

18.1.1 RDF Terms

SPARQL is defined in terms of IRIs [RFC3987]. IRIs are a subset of RDF URI References that omits the use of 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 ∪ RDF-L ∪ 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.

18.1.2 Simple Literal

Definition: Simple Literal

The set of **Simple Literals** is the set of all RDF Literals with no language tag or datatype IRI.

18.1.3 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.

Definition: RDF Dataset Merge

Let DS1 = { G1, (<$u1_1$>, $G1_1$), (<$u1_2$>, $G1_2$), . . . (<$u1_n$>, $G1_n$) },
and DS2 = { G2, (<$u2_1$>, $G2_1$), (<$u2_2$>, $G2_2$), . . . (<$u2_m$>, $G2_m$) }

then we define the RDF Dataset Merge of DS1 and DS2 to be:
DS={ G, (<u_1>, G_1), (<u_2>, G_2), . . . (<u_k>, G_k) }
where:

Write N1 for { <$u1_j$> j = 1 to n }
Write N2 for { <$u2_j$> j = 1 to m }

- G is the merge of G1 and G2
- (<u_i>, G_i) where <u_i> is in N1 but not in N2
- (<u_i>, G_i) where <u_i> is in N2 but not in N1
- (<u_i>, G_i) where <u_i> is equal to <u_j> in N1 and equal to <u_k> in N2 and G_i is the merge of $G1_j$ and $G2_k$

18.1.4 Query Variables

Definition: Query Variable

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

18.1.5 Triple Patterns

Definition: Triple Pattern

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

18.1.6 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.

18.1.7 Property Path Patterns

Definition: Property Path

A Property Path is a sequence of triples, t_i in sequence ST, with n = length(ST)-1, such that, for i=0 to n, the object of t_i is the same term as the subject of t_{i+1}.

We call the subject of t_0 the start of the path.

We call the object of t_n the end of the path.

A Property Path is a path in graph G if each t_i is a triple of G.

A property path does not span multiple graphs in a dataset.

Definition: Property Path Expression

A property path expression is an expression using the property path forms described above.

Definition: Property Path Pattern

Let PP be the set of all property path expressions. A property path pattern is a member of the set:
(RDF-T ∪ V) x PP x (RDF-T ∪ V)

A Property Path Pattern is a generalization of a Triple Pattern to include a property path expression in the property position.

18.1.8 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 -> RDF-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.

Write expr(μ) for the value of the expression expr, using the terms for variables given by μ. Evaluation may result in an error.

18.1.9 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-distinct 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

18.1.10 SPARQL Query

Definition: SPARQL Query

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

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

Definition: Query Level

A query level is a graph pattern, a set of group and aggregation, and a set of solution modifiers.

A query is a tree of "query levels", where each subquery forms one query level in the tree.

18.2 Translation to the SPARQL Algebra

This section defines the process of converting graph patterns and solution modifiers in a SPARQL query string into a SPARQL algebra expression. The process described converts one level of query nesting, as formed by subqueries using the nested `SELECT` syntax and is applied recursively on subqueries. Each level consists of graph pattern matching and filtering, followed by the application of solution modifiers.

The SPARQL query string is parsed and the abbreviations for IRIs and triple patterns given in section 4 are applied. At this point the abstract syntax tree is composed of:

Patterns	Modifiers	Query Forms	Other
RDF terms	DISTINCT	SELECT	VALUES
Property path expression	REDUCED	CONSTRUCT	SERVICE
Property path patterns	Projection	DESCRIBE	
Groups	ORDER BY	ASK	
OPTIONAL	LIMIT		
UNION	OFFSET		
GRAPH	Select expressions		
BIND			
GROUP BY			
HAVING			
MINUS			
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	Property Path
BGP	ToList	PredicatePath
Join	OrderBy	InversePath
LeftJoin	Project	SequencePath
Filter	Distinct	AlernativePath
Union	Reduced	ZeroOrMorePath
Graph	Slice	OneOrMorePath
Extend	ToMultiSet	ZeroOrOnePath
Minus		NegatedPropertySet
Group		
Aggregation		
AggregateJoin		

Slice is the combination of OFFSET and LIMIT.

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

ToMultiSet is used where conversion from a solution sequence to a multiset occurs.

18.2.1 Variable Scope

We define a variable to be *in-scope* if there is a way for a variable to be in the domain of a solution mapping at that point in the execution of the SPARQL algebra for the query. The definition below provides a way of determing this from the abstract syntax of a query.

Note that a subquery with a projection can hide variables; use of a variable in `FILTER`, or in `MINUS` does not cause a variable to be in-scope outside of those forms.

Let **P**, **P1**, **P2** be graph patterns and **E**, **E1**,...**En** be expressions. A variable `v` is in-scope if:

Syntax Form	In-scope variables
Basic Graph Pattern (BGP)	`v` occurs in the BGP
Path	`v` occurs in the path
Group `{ P1 P2 ... }`	`v` is in-scope if it is in-scope in one or more of P1, P2, ...
`GRAPH term { P }`	`v` is `term` or `v` is in-scope in P
`{ P1 } UNION { P2 }`	`v` is in-scope in P1 or in-scope in P2
`OPTIONAL {P}`	`v` is in-scope in P
`SERVICE term {P}`	`v` is `term` or `v` is in-scope in P
`BIND (expr AS v)`	`v` is in-scope
`SELECT .. v .. { P }`	`v` is in-scope
`SELECT ... (expr AS v)`	`v` is in-scope
`GROUP BY (expr AS v)`	`v` is in-scope
`SELECT * { P }`	`v` is in-scope in `P`
`VALUES v { values }`	`v` is in-scope
`VALUES varlist { values }`	`v` is in-scope if `v` is in `varlist`

The variable `v` must not be in-scope at the point of the `(expr AS v)` form. The scoping for `(expr AS v)` applies immediately in `SELECT` expressions.

In `BIND (expr AS v)` requires that the variable `v` is not in-scope from the preceeding elements in the group graph pattern in which it is used.

In `SELECT`, the variable `v` must not be in-scope in the graph pattern of the `SELECT` clause, nor used in another select expression earlier in the clause.

18.2.2 Converting Graph Patterns

This section describes the process for translating a SPARQL graph pattern into a SPARQL algebra expression. This process is applied to the group graph pattern (the unit between `{...}` delimiters) forming the `WHERE` clause of a query, and recursively to each syntactic element within the group graph pattern. The result of the translation is a SPARQL algebra expression.

In summary, the steps are applied as follows:

- Expand syntax forms for IRIs, literals and triple patterns.
- Translate property path expressions
- Convert some property path patterns to triples
- Collect the `FILTER`s in the group
- Translate Basic Graph Patterns
- Translate the remaining graph patterns in the group
- Add in Filters
- Simplify the algebra expression

We write

translate(graph pattern)

for the algorithm described here to translate graph patterns.

The working group notes that in SPARQL 1.0, the point at which 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.

Applying the simpification step after all the translation of graph patterns is the preferred reading.

18.2.2.1 Expand Syntax Forms

Expand abbreviations for IRIs and triple patterns given in section 4.

18.2.2.2 Collect `FILTER` Elements

`FILTER` expressions apply to the whole group graph pattern in which they appear. The algebra operators to perform filtering are added to the group after translation of each group element. We collect the filters together here and remove them from group, then apply them to the whole translated group graph pattern.

In this step, we also translate graph patterns within `FILTER` expressions `EXISTS` and `NOT EXISTS`.

```
Let FS := empty set

For each form FILTER(expr) in the group graph pattern:
    In expr, replace NOT EXISTS{P} with fn:not(exists(translate(P)))
    In expr, replace EXISTS{P} with exists(translate(P))
    FS := FS ∪ {expr}
    End
```

The set of filter expressions `FS` is used later.

18.2.2.3 Translate Property Path Expressions

The following table gives the translation of property paths expressions from SPARQL syntax to terms in the SPARQL algebra. This applies to all elements of a property path expression recursively.

The next step after this one translates certain forms to triple patterns, and these are converted later to basic graph patterns by adjacency (without intervening group pattern delimiters `{` and `}`) or other syntax forms. Overall, SPARQL syntax property paths of just an IRI become triple patterns and these are aggregated into basic graph patterns.

Notes:

- The order of forms IRI and ^IRI in negated property sets is not relevant.

We introduce the following symbols:

- link
- inv
- alt
- seq
- ZeroOrMorePath
- OneOrMorePath
- ZeroOrOnePath
- NPS (for NegatedPropertySet)

Syntax Form (path)	Algebra (path)
`iri`	`link(iri)`

^path	inv(path)
!(:iri$_1$|...|:iri$_n$)	NPS({:iri$_1$ ... :iri$_n$})
!(^:iri$_1$|...|^:iri$_n$)	inv(NPS({:iri$_1$ ... :iri$_n$}))
!(:iri$_1$|...|:iri$_i$|^:iri$_{i+1}$|...|^:iri$_m$)	alt(NPS({:iri$_1$ ...:iri$_i$}), inv(NPS({:iri$_{i+1}$, ..., :iri$_m$})))
path1 / path2	seq(path1, path2)
path1 | path2	alt(path1, path2)
path*	ZeroOrMorePath(path)
path+	OneOrMorePath(path)
path?	ZeroOrOnePath(path)

18.2.2.4 Translate Property Path Patterns

The previous step translated property path expressions. This step translates property path patterns, which are a subject end point, property path expression and object end point, into triple patterns or wraps in a general algebra operation for path evaluation.

Notes:

- X and Y are RDF terms or variables.
- ?V is a fresh variable.
- P and Q are path expressions.
- These are only applied to property path patterns, not within property path expressions.
- Translations earlier in the table are applied in preference to the last translation.
- The final translation simply wraps any remaining property path expression to use a common form `Path(...)`.

Algebra (path)	Translation
X link(iri) Y	X iri Y
X inv(iri) Y	Y iri X
X seq(P, Q) Y	X P ?V . ?V Q P
X P Y	Path(X, P, Y)

Examples of the whole path translation process (`?_v` is a fresh variable):

```
?s :p/:q ?o
```

```
?s :p ?_V .
?_V :q ?o
```

```
?s :p* ?o
```

```
Path(?s, ZeroOrMorePath(link(:p)), ?o)
```

```
:list rdf:rest*/rdf:first ?member
```

```
Path(:list, ZeroOrMorePath(link(rdf:rest)), ?_V) .
?_V rdf:first ?member
```

18.2.2.5 Translate Basic Graph Patterns

After translating property paths, any adjacent triple patterns are collected together to form a basic graph pattern `BGP(triples)`.

18.2.2.6 Translate Graph Patterns

Next, we translate each remaining graph pattern form, recursively applying the translation process.

If the form is GroupOrUnionGraphPattern

```
Let A := undefined

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

The result is A
```

If the form is GraphGraphPattern

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

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

If the form is GroupGraphPattern:

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

For each element E in the GroupGraphPattern

    If E is of the form OPTIONAL{P}
        Let A := Translate(P)
        If A is of the form Filter(F, A2)
            G := LeftJoin(G, A2, F)
        Else
            G := LeftJoin(G, A, true)
            End
        End

    If E is of the form MINUS{P}
        G := Minus(G, Translate(P))
        End

    If E is of the form BIND(expr AS var)
        G := Extend(G, var, expr)
        End

    If E is any other form
        Let A := Translate(E)
        G := Join(G, A)
        End

   End

The result is G.
```

If the form is InlineData

```
The result is a multiset of solution mappings 'data'.
```

data is formed by forming a solution mapping from the variable in the corresponding position in list of variables (or single variable), omitting a binding if the `BindingValue` is the word `UNDEF`.

If the form is SubSelect

```
The result is ToMultiset(Translate(SubSelect))
```

18.2.2.7 Filters of Group

After the group has been translated, the filter expressions are added so they wil apply to the whole of the rest of the group:

```
If FS is not empty
    Let G := output of preceding step
    Let X := Conjunction of expressions in FS
    G := Filter(X, G)
    End
```

18.2.2.8 Simplification step

Some groups of one graph pattern become `join(Z, A)`, where Z is the empty basic graph pattern (which is the empty set). These 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
```

18.2.3 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 of 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) ,
  )
```

Example: Pattern involving BIND:

```
{ ?s :p ?v . BIND (2*?v AS ?v2) ?s :p1 ?v2 }
```

```
Join(
   Extend( BGP(?s :p ?v), ?v2, 2*?v) ,
   BGP(?s :p1 ?v2) )
```

Example: Pattern involving BIND:

```
{ ?s :p ?v . {} BIND (2*?v AS ?v2) }
```

```
Join(
   BGP(?s :p ?v), ?v2, 2*?v) ,
   Extend({}, ?v2, 2*?v)
)
```

Example: Pattern involving MINUS:

```
{ ?s :p ?v . MINUS {?s :p1 ?v2 } }
```

```
Minus(
   BGP(?s :p ?v)
   BGP(?s :p1 ?v2))
```

Example: Pattern involving a subquery:

```
{ ?s :p ?o . {SELECT DISTINCT ?o {?o ?p ?z} } }
```

```
Join(
   BGP(?s :p ?o) ,
   ToMultiSet(
     Distinct(Project(BGP(?o ?p ?z), {?o})) )
   )
```

18.2.4 Converting Groups, Aggregates, HAVING, final VALUES clause and SELECT Expressions

In this step, we process clauses on the query level in the following order:

- Grouping
- Aggregates
- HAVING
- VALUES
- Select expressions

18.2.4.1 Grouping and Aggregation

Step: GROUP BY

If the GROUP BY keyword is used, or there is implicit grouping due to the use of aggregates in the projection, then grouping is performed by the Group function. It divides the solution set into groups of one or more solutions, with the same overall cardinality. In case of implicit grouping, a fixed constant (1) is used to group all solutions into a single group.

Step: Aggregates

The aggregation step is applied as a transformation on the query level, replacing aggregate expressions in the query level with Aggregation() algebraic expressions.

The transformation for query levels that use any aggregates is given below:

```
Let A := the empty sequence
Let Q := the query level being evaluated
Let P := the algebra translation of the GroupGraphPattern of the query level
Let E := [], a list of pairs of the form (variable, expression)

If Q contains GROUP BY exprlist
   Let G := Group(exprlist, P)
Else If Q contains an aggregate in SELECT, HAVING, ORDER BY
   Let G := Group((1), P)
Else
   skip the rest of the aggregate step
   End

Global i := 1   # Initially 1 for each query processed

For each (X AS Var) in SELECT, each HAVING(X), and each ORDER BY X in Q
  For each unaggregated variable V in X
      Replace V with Sample(V)
      End
  For each aggregate R(args ; scalarvals) now in X
      # note scalarvals may be omitted, then it's equivalent to the empty set
      A_i := Aggregation(args, R, scalarvals, G)
      Replace R(...) with agg_i in Q
      i := i + 1
      End
  End

For each variable V appearing outside of an aggregate
   A_i := Aggregation(V, Sample, {}, G)
   E := E append (V, agg_i)
   i := i + 1
   End

A := A_i, ..., A_i-1
P := AggregateJoin(A)
```

Note: agg_i is a temporary variable. E is then used in 18.2.4.4 for the processing of select expressions.

Example:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT (SUM(?val) AS ?sum) (COUNT(?a) AS ?count)
WHERE {
  ?a rdf:value ?val .
} GROUP BY ?a
```

The SUM expression becomes agg_1, and the COUNT expression becomes agg_2.

```
Let G := Group((?a), BGP(?a rdf:value ?val))
A_1 = Aggregation((?val), Sum, {}, G)
A_2 = Aggregation((?a), Count, {}, G)
A := (A_1, A_2)
Let P := AggregateJoin(A)
```

18.2.4.2 HAVING

The HAVING expression is evaluated using the same rules as FILTER(). Note that, due to the logic position in which the HAVING clause is evaluated, expressions projected by the SELECT clause are not visible to the HAVING clause.

```
Let Q := the query level being evaluated
Let P := the algebra translation of the query level so far

For each HAVING(E) in Q
    P := Filter(E, P)
    End
```

18.2.4.3 VALUES

If the query has a trailing VALUES clause:

```
Let P := the algebra translation of the query level so far
P := Join(P, ToMultiSet(data))
  where data is a solution sequence formed from the VALUES clause
```

The translatation of the data is the same as for inline data.

18.2.4.4 SELECT Expressions

Step: Select expressions

We have two forms of the abstract syntax to consider:

```
SELECT selItem ... { pattern }
SELECT * { pattern }
```

```
Let X := algebra from earlier steps
Let VS := list of all variables visible in the pattern,
          so restricted by sub-SELECT projected variables and GROUP BY variables.
          Not visible: only in filter, exists/not exists, masked by a subselect,
                       non-projected GROUP variables, only in the right hand side of MINUS

Let PV := {}, a set of variable names
Note, E is a list of pairs of the form (variable, expression), defined in section 18.2.4

If "SELECT *"
    PV := VS

If  "SELECT selItem ...:"
    For each selItem:
        If selItem is a variable
            PV := PV ∪ { variable }
        End
        If selItem is (expr AS variable)
            variable must not appear in VS nor in PV; if it does then generate a syntax error and stop
            PV := PV ∪ { variable }
            E := E append (variable, expr)
        End
    End

For each pair (var, expr) in E
    X := Extend(X, var, expr)
    End

Result is X
The set PV is used later for projection.
```

The syntax error arises for use of a variable as the named target of AS (e.g. ... AS ?x) when the variable is used inside the WHERE clause of the SELECT or if already used as the traget of AS in this SELECT expression.

18.2.5 Converting Solution Modifiers

Solutions modifiers apply to the processing of a SPARQL query after pattern matching. The solution modifiers are applied to a query in the following order:

- Order by
- Projection
- Distinct
- Reduced
- Offset
- Limit

Step: 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)

18.2.5.1 ORDER BY

If the query string has an ORDER BY clause

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

18.2.5.2 Projection

The set of projection variables, PV, was calculated in the processing of SELECT expressions.

M := Project(M, PV)

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

18.2.5.3 DISTINCT

If the query contains DISTINCT,

M := Distinct(M)

18.2.5.4 REDUCED

If the query contains REDUCED,

M := Reduced(M)

18.2.5.5 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).

18.2.5.6 Final Algebra Expression

The overall abstract query is M.

18.3 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.

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) for the solution mapping variable x to RDF term t : { (x, t) }

Write Ω(x) 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), $\mu_1(v) = \mu_2(v)$.

Here, $\mu_1(v) = \mu_2(v)$ means that $\mu_1(v)$ and $\mu_2(v)$ are the same RDF term.

If μ_1 and μ_2 are compatible then $\mu_1 \cup \mu_2$ is also a mapping. Write merge(μ_1, μ_2) for $\mu_1 \cup \mu_2$

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

18.3.1 SPARQL Basic Graph 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))

For a BGP 'x', P(x) denotes the result of replacing blank nodes b in x for which σ is defined with σ(b) and all variables v in x for which μ is defined with μ(v).

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.

18.3.2 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 results format document (SPARQL Query Results XML Format, SPARQL 1.1 Query Results JSON Format and SPARQL 1.1 Query Results CSV and TSV Formats) 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.

18.4 Property Path Patterns

This section defines the evaluation of property path patterns. A property path pattern is a subject endpoint (an RDF term or a variable), a property path express and an object endpoint. The translation of property path expressions converts some forms to other SPARQL expressions, such as converting property paths of length one to triple patterns, which in turn are combined into basic graph patterns. This leaves property path operators ZeroOrOnePath, ZeroOrMorePath, OneOrMorePath and NegatedPropertySets and also path expressions contained within these operators.

All remaining property path expressions are present in the algebra in the form `Path(X, path, Y)` for endpoints X and Y. For example: syntax`(:p/:q)*` is a ZeroOrMorePath expression involving a sequence property path becoming the algebra expression `ZeroOrMorePath(seq(link(:p), link(:q)))`.

Notation

Write

```
eval(Path(X, PP, Y))
```

for the evaluation of the property path patterns. This produces a multiset of solution mappings μ, each solution mapping having a binding for variables used (each of X and Y can be a variable). Some operators only produce a set of solution mappings.

Write

$Var(x_1, x_2, ..., x_n) = \{ x_i \mid i \text{ in } 1...n \text{ and } x_i \text{ is a variable} \}$

for the variables in $x_1, x_2, ..., x_n$.

Write

`x:term`	when `x` is an RDF term
`x:var`	when `x` is a variable
`x:path`	when `x` is a path expression

All evaluation is carried out by matching the active graph at that point in the overall query evaluation. We omit explicitly including the active graph in each definition for clarity.

Definition: Evaluation of Predicate Property Path

Let Path(X, link(iri), Y) be an predicate inverse property path pattern, using some IRI iri.

```
eval(Path(X, link(iri), Y)) = evaluation of basic graph pattern {X iri Y}
```

If both X and Y are variables, this is the same as:

```
eval(Path(X:var, link(iri), Y:var)) =
    { (X, xn) (Y, yn) | xn and yn are RDF terms and triple (xn iri yn) is in the active graph }
```

If X is a variable and Y an RDF term:

```
eval(Path(X:var, link(iri), Y:term)) =
    { (X, xn) | xn is an RDF term and triple (xn iri Y) is in the active graph }
```

If X is an RDF term and Y is a variable:

```
eval(Path(X:term, link(iri), Y:var)) =
    { (Y, yn) | yn is an RDF term and triple (X iri yn) is in the active graph }
```

If both X and Y are RDF terms:

```
eval(Path(X:term, link(iri), Y:term)) =
    { μ0 } if triple (X iri Y) is in the active graph
    = { { } }
    = Ω0

eval(Path(X:term, link(iri), Y:term)) =
    { } if triple (X iri Y) is not in the active graph
```

Informally, evaluating a Predicate Property Path is the same as executing a subquery `SELECT * { X P Y }` at that point in the query evaluation.

Definition: Evaluation of Inverse Property Path

Let P be a property path expression, then:

```
eval(Path(X, inv(P), Y)) = eval(Path(Y, P, X))
```

Definition: Evaluation of Sequence Property Path

Let P and Q be property path expressions. Let V be a fresh variable.

```
A = Join( eval(Path(X, P, V)), eval(Path(V, Q, Y)) )

eval(Path(X, seq(P,Q), Y)) = Project(A, Var(X,Y))
```

Informally, this is the same as:

```
SELECT * { X P _:a . _:a Q Y }
```

using the fact that a blank node `_:a` acts like a variable (under simple entailment) except it does not appear in the results from `SELECT *`.

Definition: Evaluation of Alternative Property Path

Let P and Q be property path expressions.

```
eval(Path(X, alt(P,Q), Y)) = Union(eval(Path(X, P, Y)), eval(Path(X, Q, Y)))
```

Informally, this is the same as:

```
SELECT * { { X P Y } UNION { X Q Y } }
```

Definition: Node set of a graph

The node set of a graph G, nodes(G), is:

nodes(G) = { n | n is an RDF term that is used as a subject or object of a triple of G}

Definition: Evaluation of ZeroOrOnePath

```
eval(Path(X:term, ZeroOrOnePath(P), Y:var)) = { (Y, yn) | yn = X or {(Y, yn)} in eval(Path(X,P,Y)) }

eval(Path(X:var, ZeroOrOnePath(P), Y:term)) = { (X, xn) | xn = Y or {(X, xn)} in eval(Path(X,P,Y)) }

eval(Path(X:term, ZeroOrOnePath(P), Y:term)) =
    { {} } if X = Y or eval(Path(X,P,Y)) is not empty
    { } othewise

eval(Path(X:var, ZeroOrOnePath(P), Y:var)) =
    { (X, xn) (Y, yn) | either (yn in nodes(G) and xn = yn) or {(X,xn), (Y,yn)} in eval(Path(X,P,Y)) }
```

We define an auxillary function, ALP, used in the definitions of ZeroOrMorePath and OneOrMorePath. Note that the algorithm given here serves to specify the feature. An implementation is free to implement evaluation by any method that produces the same results for the query overall. The ZeroOrMorePath and OneOrMorePath forms return matches based on distinct nodes connected by the path.

The matching algorithm is based on following all paths, and detecting when a graph node (subject or object), has been already visited on the path.

Informally, this algorithm attempts to extend the multiset of results by one application of `path` at each step, noting which nodes it has visited for this particular path. If a node has been visited for the path under consideration, it is not a candidate for another step.

Definition: Function ALP

```
Let eval(x:term, path) be the evaluation of 'path', starting at RDF term x,
                       and returning a multiset of RDF terms reached
                       by repeated matches of path.

ALP(x:term, path) =
    Let V = empty multiset
    ALP(x:term, path, V)
    return is V

# V is the set of nodes visited

ALP(x:term, path, V:set of RDF terms) =
    if ( x in V ) return
    add x to V
    X = eval(x,path)
    For n:term in X
        ALP(n, path, V)
        End
```

Definition: Evaluation of ZeroOrMorePath

```
eval(Path(X:term, ZeroOrMorePath(path), vy:var)) =
    { { (vy, n) } | n in ALP(X, path) }

eval(Path(vx:var, ZeroOrMorePath(path), vy:var)) =
    { { (vx, t), (vy, n) } |  t in nodes(G), (vy, n) in eval(Path(t, ZeroOrMorePath(path), vy)) }

eval(Path(vx:var, ZeroOrMorePath(path), y:term)) =
    eval(Path(y:term, ZeroOrMorePath(inv(path)), vx:var))

eval(Path(x:term, ZeroOrMorePath(path), y:term)) =
    { { } } if { (vy:var,y) } in eval(Path(x, ZeroOrMorePath(path) vy)
    { } otherwise
```

Definition: Evaluation of OneOrMorePath

eval(Path(X, OneOrMorePath(path), Y))

```
# For OneOrMorePath, we take one step of the path then start
# recording nodes for results.

eval(Path(x:term, OneOrMorePath(path), vy:var)) =
    Let X = eval(x, path)
    Let V = the empty multiset
    For n in X
        ALP(n, path, V)
        End
    result is V

eval(Path(vx:var, OneOrMorePath(path), vy:var)) =
   { { (vx, t), (vy, n) } |  t in nodes(G), (vy, n) in eval(Path(t, OneOrMorePath(path), vy)) }

eval(Path(vx:var, OneOrMorePath(path), y:term)) =
   eval(Path(y:term, OneOrMorePath(inv(path)), vx))

eval(Path(x:term, OneOrMorePath(path), y:term)) =
    { { } } if { (vy:var, y) } in eval(Path(x, OneOrMorePath(path), vy))
    { } otherwise
```

Definition: Evaluation of NegatedPropertySet

```
Write μ' as the extension of a solution mapping:
μ'(μ,x) = μ(x)   if x is a variable
μ'(μ,t) = t   if t is a RDF term


Let x and y be variables or RDF terms, and S a set of IRIs:

eval(Path(x, NPS(S), y)) = { μ | ∃ triple(μ'(μ,x), p, μ'(μ,y)) in G, such that the IRI of p ∉ S }
```

18.5 SPARQL Algebra

For each remaining 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". Evaluation of basic graph patterns and property path patterns has been described above.

Definition: Filter

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

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

card[Filter(expr, Ω, D(G))](μ) = card[Ω](μ)

> Note that evaluating an `exists(pattern)` expression uses the dataset and active graph, D(G). See the evaluation of filter.

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 and expr be an expression. We define:

Diff(Ω_1, Ω_2, expr) = { μ | μ in Ω_1 such that ∀ 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)) ∪ 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 Ω_1 and μ_2 in Ω_2, μ_1 and μ_2 are compatible and expr(merge(μ_1, μ_2)) is true }
∪
{ μ_1 | μ_1 in Ω_1, ∀ μ_2 in Ω_2, μ_1 and μ_2 are not compatible, or Ω_2 is empty }
∪
{ μ_1 | μ_1 in Ω_1, ∃ μ_2 in Ω_2, 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(Ω_1, Ω_2)](μ) = card[Ω_1](μ) + card[Ω_2](μ)

Definition: Minus

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

Minus(Ω_1, Ω_2) = { μ | μ in Ω_1 . ∀ μ' in Ω_2, either μ and μ' are not compatible or dom(μ) and dom(μ') are disjoint }

card[Minus(Ω_1, Ω_2)](μ) = card[Ω_1](μ)

The additional restriction on dom(μ) and dom(μ') is added because otherwise if there is a solution mapping in Ω_2 that has no variables in common with the solution mappings of Ω_1, then Minus(Ω_1, Ω_2) would be empty, regardless of the rest of Ω_2. The empty solution mapping is compatible with every other solution mapping so `P MINUS {}` would otherwise be empty for any pattern `P`.

Definition: Extend

Let μ be a solution mapping, Ω a multiset of solution mappings, *var* a variable and *expr* be an expression, then we define:

Extend(μ, var, expr) = μ ∪ { (var,value) | var not in dom(μ) and value = expr(μ) }

Extend(μ, var, expr) = μ if var not in dom(μ) and expr(μ) is an error

Extend is undefined when var in dom(μ).

Extend(Ω, var, expr) = { Extend(μ, var, expr) | μ in Ω }

Write [x | C] for a sequence of elements where C is a condition on x.

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)

Definition: ToMultiSet

Let Ψ be a solution sequence. We define:

ToMultiSet(Ψ) = { μ | μ in Ψ }

card[ToMultiSet(Ψ)](μ) = card[Ψ](μ)

ListEval is a function which is used to evaluate a list of expressions against a solution and return a list of the resulting values.

Definition: ToMultiset

ToMultiset turns a sequence into a multiset with the same elements and cardinality as the sequence. The order of the sequence has no effect on the resulting multiset, and duplicates are preserved.

Definition: Exists

exists(pattern) is a function that returns true if the pattern evaluates to a non-empty solution sequence, given the current solution mapping and active graph at the time of evaluation; otherwise it returns false.

18.5.1 Aggregate Algebra

Group is a function which groups a solution sequence into multiple solutions, based on some attribute of the solutions.

Definition: Group

Group evaluates a list of expressions against a solution sequence, producing a set of partial functions from keys to solution sequences.

Group(exprlist, Ω) = { ListEval(exprlist, μ) → { μ' | μ' in Ω, ListEval(exprlist, μ) = ListEval(exprlist, μ') } | μ in Ω }

Definition: ListEval

ListEval((expr$_1$, ..., expr$_n$), μ) returns a list (e_1, ..., e_n), where e_i = expr$_i$(μ) or error.

ListEval retains errors resulting from the evaluation of the list elements.

Note that, although the result of a ListEval can be an error, and errors may be used to group, solutions containing error values are removed at projection time.

ListEval((unbound), μ) = (error), as the evaluation of an unbound expression is an error.

Aggregation, a function which calculates a scalar value as an output of the aggregate expression. It is used in the SELECT clause, the HAVING evaluation process, and in ORDER BY (where required). Aggregation calculates aggregated values over groups of solutions, using set functions.

Definition: Aggregation

Let *exprlist* be a list of expressions or *, *func* a set function, *scalarvals* a set of partial functions (possibly empty) passed from the aggregate in the query, and let { key$_1$→Ω$_1$, ..., key$_m$→Ω$_m$ } be a multiset of partial functions from keys to solution sequences as produced by the grouping step.

Aggregation applies the set function func to the given multiset and produces a single value for each key and partition of solutions for that key.

Aggregation(exprlist, func, scalarvals, { key$_1$→Ω$_1$, ..., key$_m$→Ω$_m$ })
  = { (key, F(Ω)) | key → Ω in { key$_1$→Ω$_1$, ..., key$_m$→Ω$_m$ } }

where
 M(Ω) = { ListEval(exprlist, μ) | μ in Ω }
 F(Ω) = func(M(Ω), scalarvals), for non-DISTINCT
 F(Ω) = func(Distinct(M(Ω)), scalarvals), for DISTINCT

Special Case: when `COUNT` is used with the expression `*` the value of F will be the cardinality of the group solution sequence, `card[Ω]`, or `card[Distinct(Ω)]` if the `DISTINCT` keyword is present.

scalarvals are used to pass values to the underlying set function, bypassing the mechanics of the grouping. For example, the aggregate expression `GROUP_CONCAT(?x ; separator="|")` has a scalarvals argument of { "separator" → "|" }.

All aggregates may have the `DISTINCT` keyword as the first token in their argument list. If this keyword is present then first argument to func is Distinct(M).

Example

Given a solution multiset (Ω) with the following values:

solution	?x	?y	?z
μ$_1$	1	2	3
μ$_2$	1	3	4
μ$_3$	2	5	6

And the query expression SELECT (ex:agg(?y, ?z) AS ?agg) WHERE { ?x ?y ?z } GROUP BY ?x.

We produce G = Group((?x), Ω) = { ((1), { μ$_1$, μ$_2$ }), ((2), { μ$_3$ }) }

And so Aggregation((?y, ?z), ex:agg, {}, G) =
{ ((1), eg:agg({(2, 3), (3, 4)}, {})), ((2), eg:agg({(5, 6)}, {})) }.

Definition: AggregateJoin

Let S_1, ..., S_n be a list of sets, where each set S_i contains key to (aggregated) value maps as produced by Aggregate.

Let K = { key | key in dom(S_j) for some 1 <= j <= n } be the set of keys, then
AggregateJoin(S_1, ..., S_n) = { agg$_1$→val$_1$, ..., agg$_n$→val$_n$ | key in K and key→val$_i$ in S_i for each 1 <= i <= n }

Flatten is a function which is used to collapse multisets of lists into a multiset, so for example { (1, 2), (3, 4) } becomes { 1, 2, 3, 4 }.

Definition: Flatten

The Flatten(M) function takes a multiset of lists, M {(L_1, L_2, ...), ...}, and returns the multiset { x | L in M and x in L }.

18.5.1.1 Set Functions

The set functions which underlie SPARQL aggregates all have a common signature: SetFunc(M), or SetFunc(M, scalarvals) where M is a multiset of lists, and scalarvals is one or more scalar values that are passed to the set function indirectly via the (... ; key=value) syntax for aggregates in the SPARQL grammar. The only use of this that is supported by the built-in aggregates in SPARQL Query 1.1 is `GROUP_CONCAT`, as in `GROUP_CONCAT(?x ; separator=", ")`.

Note that the name "Set Function" is somewhat historical — the arguments to set functions are in fact multisets. The name is retained due to the commonality with SQL Set Functions, which also operate over multisets.

The set functions defined in this document are Count, Sum, Min, Max, Avg, GroupConcat, and Sample — corresponding to the aggregates `COUNT`, `SUM`, `MIN`, `MAX`, `AVG`, `GROUP_CONCAT`, and `SAMPLE`. Definitions may be found in the following sections. Systems may choose to expand this set using local extensions, using the same notation as for functions and casts. Note that, unless the ; separator is used this requires the parser to know whether some IRI refers to a function, cast, or aggregate before it can determine if there are any errors in a query where aggregates are used.

18.5.1.2 Count

Count is a SPARQL set function which counts the number of times a given expression has a bound, and non-error value within the aggregate group.

Definition: Count

xsd:integer Count(multiset M)

N = Flatten(M)

remove error elements from N

Count(M) = card[N]

18.5.1.3 Sum

Sum is a SPARQL set function that will return the numeric value obtained by summing the values within the aggregate group. Type promotion happens as per the op:numeric-add function, applied transitively, (see definition below) so the value of SUM(?x), in an aggregate group where ?x has values 1 (integer), 2.0e0 (float), and 3.0 (decimal) will be 6.0 (float).

Definition: Sum

numeric Sum(multiset M)

The Sum set function is used by the `SUM` aggregate in the syntax.

Sum(M) = Sum(ToList(Flatten(M))).

Sum(S) = op:numeric-add(S_1, Sum($S_{2..n}$)) when card[S] > 1
Sum(S) = op:numeric-add(S_1, 0) when card[S] = 1
Sum(S) = "0"^^xsd:integer when card[S] = 0

In this way, Sum({1, 2, 3}) = op:numeric-add(1, op:numeric-add(2, op:numeric-add(3, 0))).

18.5.1.4 Avg

The Avg set function calculates the average value for an expression over a group. It is defined in terms of Sum and Count.

Definition: Avg

numeric Avg(multiset M)

Avg(M) = "0"^^xsd:integer, where Count(M) = 0

Avg(M) = Sum(M) / Count(M), where Count(M) > 0

For example, Avg({1, 2, 3}) = Sum({1, 2, 3})/Count({1, 2, 3}) = 6/3 = 2.

18.5.1.5 Min

Min is a SPARQL set functions that returns the minimum value from a group respectively.

It makes use of the SPARQL ORDER BY ordering definition, to allow ordering over arbitrarily typed expressions.

Definition: Min

term Min(multiset M)

Min(M) = Min(ToList(Flatten(M)))

Min({}) = error.

The flattened multiset of values passed as an argument is converted to a sequence S, this sequence is ordered as per the `ORDER BY ASC` clause.

Min(S) = S_0

18.5.1.6 Max

Max is a SPARQL set function that return the maximum value from a group respectively.

It makes use of the SPARQL ORDER BY ordering definition, to allow ordering over arbitrarily typed expressions.

Definition: Max

term Max(multiset M)

Max(M) = Max(ToList(Flatten(M)))

Max({}) = error.

The multiset of values passed as an argument is converted to a sequence S, this sequence is ordered as per the `ORDER BY DESC` clause.

Max(S) = S_0

18.5.1.7 GroupConcat

GroupConcat is a set function which performs a string concatenation across the values of an expression with a group. The order of the strings is not specified. The separator character used in the concatenation may be given with the scalar argument SEPARATOR.

Definition: GroupConcat

literal GroupConcat(multiset M)

If the "separator" scalar argument is absent from GROUP_CONCAT then it is taken to be the "space" character, unicode codepoint U+0020.

The multiset of values, M passed as an argument is converted to a sequence S.

GroupConcat(M, scalarvals) = GroupConcat(Flatten(M), scalarvals("separator"))

GroupConcat(S, sep) = "", where $|S| = 0$

GroupConcat(S, sep) = CONCAT("", S_0), where $|S| = 1$

GroupConcat(S, sep) = CONCAT(S_0, sep, GroupConcat($S_{1..n-1}$, sep)), where $|S| > 1$

For example, GroupConcat({"a", "b", "c"}, {"separator" → "."}) = "a.b.c".

18.5.1.8 Sample

Sample is a set function which returns an arbitrary value from the multiset passed to it.

Definition: Sample

RDFTerm Sample(multiset M)

Sample(M) = v, where v in Flatten(M)

Sample({}) = error

For example, given Sample({"a", "b", "c"}), "a", "b", and "c" are all valid return values. Note that Sample() is not required to be deterministic for a given input, the only restriction is that the output value must be present in the input multiset.

18.6 Evaluation Semantics

We define eval(D(G), algebra expression) as the evaluation of an algebra expression 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
P, P1, P2 : graph patterns
L : a solution sequence
F : an expression
```

Definition: Evaluation of a Basic Graph Pattern

`eval(D(G), BGP) = multiset of solution mappings`

See section Basic Graph Patterns

Definition: Evaluation of a Property Path Pattern

`eval(D(G), Path(X, path, Y)) = multiset of solution mappings`

See section Property Path Expresions

Definition: Evaluation of Filter

```
eval(D(G), Filter(F, P)) = Filter(F, eval(D(G),P), D(G))
```

'substitute' is a filter function in support of the evaluation of `EXISTS` and `NOT EXISTS` forms which were translated to `exists`.

Definition: Substitute

Let μ be a solution mapping.

substitute(*pattern*, μ) = the pattern formed by replacing every occurrence of a variable v in *pattern* by μ(v) for each v in dom(μ)

Definition: Evaluation of Exists

Let μ be the current solution mapping for a filter and P a graph pattern:

The value exists(P), given D(G) is true if and only if eval(D(G), substitute(P, μ)) is a non-empty sequence.

Definition: Evaluation of Join

```
eval(D(G), Join(P1, P2)) = Join(eval(D(G), P1), eval(D(G), P2))
```

Definition: Evaluation of LeftJoin

```
eval(D(G), LeftJoin(P1, P2, F)) = LeftJoin(eval(D(G), P1), eval(D(G), P2), F)
```

Definition: Evaluation of Union

```
eval(D(G), Union(P1,P2)) = Union(eval(D(G), P1), eval(D(G), P2))
```

Definition: Evaluation of Graph

```
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 Group

eval(D(G), Group(exprlist, P)) = Group(exprlist, eval(D(G), P))

Definition: Evaluation of Aggregation

eval(D(G), Aggregation(exprlist, func, scalarvals, P)) = Aggregation(exprlist, func, scalarvals, eval(D(G), P))

Definition: Evaluation of AggregateJoin

eval(D(G), AggregateJoin(A_1, ..., A_n)) = AggregateJoin(eval(D(G), A_1), ..., eval(D(G), A_n))

Note that if eval(D(G), A_i) is an error, it is ignored.

Definition: Evaluation of Extend

```
eval(D(G), Extend(P, var, expr)) = Extend(eval(D(G), P), var, expr)
```

Definition: Evaluation of ToList

```
eval(D(G), ToList(P)) = ToList(eval(D(G), P))
```

Definition: Evaluation of Distinct

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

Definition: Evaluation of Reduced

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

Definition: Evaluation of Project

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

Definition: Evaluation of OrderBy

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

Definition: Evaluation of ToMultiSet

```
eval(D(G), ToMultiSet(L)) = ToMultiSet(eval(D), M))
```

Definition: Evaluation of Slice

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

18.7 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.

Detailed definitions for querying various entailment regimes can be found in SPARQL 1.1 Entailment Regimes.

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.

An entailment regime E must provide conditions on basic graph pattern evaluation such that for any basic graph pattern BGP, any RDF graph G, and any evaluation that satisfies the conditions, the resulting multiset of solutions is uniquely determined up to RDF graph equivalence. We denote the multiset of solutions from evaluating BGP over G using E with Eval-E(G, BGP).
An entailment regime must further satisfy the following conditions:

1. For any E-consistent active graph AG, the entailment regime E uniquely specifies a scoping graph SG that is E-equivalent to AG.
2. A set of well-formed graphs for E is specified such that, for any basic graph pattern BGP, scoping graph SG, and solution mapping μ in Eval-E(SG, BGP), the graph μ(BGP) is well-formed for E.
3. For any basic graph pattern BGP and scoping graph SG, if μ_1, ..., μ_n in Eval-E(SG, BGP) and BGP_1, ..., BGP_n are basic graph patterns all equivalent to BGP but not sharing any blank nodes with each other or with SG, then

   SG E-entails (SG union $\mu_1(BGP_1)$ union ... union $\mu_n(BGP_n)$)

   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. Entailment regimes should provide conditions to prevent trivial infinite solution multisets as appropriate to the regime.

18.7.1 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 bindings to blank nodes, then this condition can be simplified to the condition:

SG E-entails μ(BGP) for each solution mapping μ.

(c) These conditions do not impose the SPARQL requirement that SG shares 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.

(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).

For every solution mapping μ_i, there is, by definition of basic graph pattern matching, an RDF instance mapping σ_i such that $P_i(BGP_i)$ is a subgraph of SG where P_i is the pattern instance mapping composed of μ_i and σ_i. Since BGP_i and SG have no blank nodes in common, the ranges of σ_i and μ_i contain no blank nodes from BGP_i; therefore, the solution mapping μ_i and the RDF instance mapping σ_i of P_i commute, so $P_i(BGP_i) = \sigma_i(\mu_i(BGP_i))$. So

$P_1(BGP_1)$ union ... union $P_n(BGP_n)$
$= \sigma_1(\mu_1(BGP_1))$ union ... union $\sigma_n(\mu_n(BGP_n))$
$= [\ \sigma_1 + ... + \sigma_n](\ \mu_1(BGP_1)$ union ... union $\mu_n(BGP_n)\ )$

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

SG union $[\ \sigma_1 + ... + \sigma_n](\ \mu_1(BGP_1)$ union ... union $\mu_n(BGP_n)\ )$
$= [\ \sigma_1 + ... + \sigma_n]$(SG union $\mu_1(BGP_1)$ union ... union $\mu_n(BGP_n)\ )$

i.e.

SG union $\mu_1(BGP_1)$ union ... union $\mu_n(BGP_n)$

has an instance which is a subgraph of SG, so is simply entailed by SG by the RDF interpolation lemma [RDF-MT].

19 SPARQL Grammar

The SPARQL grammar covers both SPARQL Query and SPARQL Update.

19.1 SPARQL Request String

A SPARQL Request String is a SPARQL Query String or SPARQL Update String and it is a Unicode character string (c.f. section 6.1 String concepts of [CHARMOD]) in the language defined by the following grammar.

A SPARQL Query String start at the QueryUnit production.

A SPARQL Update String start at the UpdateUnit 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`.

19.2 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).

19.3 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. Otherwise, white space is ignored between tokens.

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 '&&' connecting two expression using '<' (less than) and '>' (greater than).

19.4 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.

19.5 IRI References

Text matched by the IRIREF production and PrefixedName (after prefix expansion) production, after escape processing, must 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 IRIREF <abc#def> may occur in a SPARQL query string, but the IRIREF <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 4.1.1, Syntax of IRI Terms, for a description of BASE and PREFIX.

19.6 Blank Nodes and Blank Node Labels

Blank nodes can not be used in:

- DELETE WHERE
- DELETE DATA
- a DeleteClause

in a SPARQL Update request.

Blank node labels are scoped to the SPARQL Request String in which they occur. Different uses of the same blank node label in a request string refer to the same blank node. Fresh blank nodes are generated for each request; blank nodes can not be referenced by label across requests.

The same blank node label can not be used in:

- two basic graph patterns in a SPARQL Query
- two WHERE clauses within a single SPARQL Update request
- two INSERT DATA operations within a single SPARQL Update request

Note that the same blank node label can occur in different QuadPattern clauses in a SPARQL Update request.

19.7 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'
```

19.8 Grammar

The EBNF notation used in the grammar is defined in Extensible Markup Language (XML) 1.1 [XML11] section 6 Notation.

Notes:

1. 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).
2. Escape sequences are case sensitive.
3. When tokenizing the input and choosing grammar rules, the longest match is chosen.
4. The SPARQL grammar is LL(1) when the rules with uppercased names are used as terminals.
5. There are two entry points into the grammar: QueryUnit for SPARQL queries, and UpdateUnit for SPARQL Update requests.

6. In signed numbers, no white space is allowed between the sign and the number. The AdditiveExpression grammar rule allows for this by covering the two cases of an expression followed by a signed number. These produce an addition or subtraction of the unsigned number as appropriate.
7. The tokens INSERT DATA, DELETE DATA, DELETE WHERE allow any amount of white space between the words. The single space version is used in the grammar for clarity.
8. The QuadData and QuadPattern rules both use rule Quads. The rule QuadData, used in INSERT DATA and DELETE DATA, must not allow variables in the quad patterns.
9. Blank node syntax is not allowed in DELETE WHERE, the DeleteClause for DELETE, nor in DELETE DATA.
10. Rules for limiting the use of blank node labels are given in section 19.6.
11. The number of variables in the variable list of VALUES block must be the same as the number of each list of associated values in the DataBlock.
12. Variables introduced by AS in a SELECT clause must not already be in-scope.
13. The variable assigned in a BIND clause must not be already in-use within the immediately preceding TriplesBlock within a GroupGraphPattern.
14. Aggregate functions can be one of the built-in keywords for aggregates or a custom aggregate, which is syntactically a function call. Aggregate functions may only be used in SELECT, HAVING and ORDER BY clauses.
15. Only custom aggregate functions use the DISTINCT keyword in a function call.

[1]	QueryUnit	::=	Query
[2]	Query	::=	Prologue (SelectQuery | ConstructQuery | DescribeQuery | AskQuery) ValuesClause
[3]	UpdateUnit	::=	Update
[4]	Prologue	::=	(BaseDecl | PrefixDecl)*
[5]	BaseDecl	::=	'BASE' IRIREF
[6]	PrefixDecl	::=	'PREFIX' PNAME_NS IRIREF
[7]	SelectQuery	::=	SelectClause DatasetClause* WhereClause SolutionModifier
[8]	SubSelect	::=	SelectClause WhereClause SolutionModifier ValuesClause
[9]	SelectClause	::=	'SELECT' ('DISTINCT' | 'REDUCED')? ((Var | ('(' Expression 'AS' Var ')'))+ | '*')
[10]	ConstructQuery	::=	'CONSTRUCT' (ConstructTemplate DatasetClause* WhereClause SolutionModifier | DatasetClause* 'WHERE' '{' TriplesTemplate? '}' SolutionModifier)
[11]	DescribeQuery	::=	'DESCRIBE' (VarOrIri+ | '*') DatasetClause* WhereClause? SolutionModifier
[12]	AskQuery	::=	'ASK' DatasetClause* WhereClause SolutionModifier
[13]	DatasetClause	::=	'FROM' (DefaultGraphClause | NamedGraphClause)
[14]	DefaultGraphClause	::=	SourceSelector
[15]	NamedGraphClause	::=	'NAMED' SourceSelector
[16]	SourceSelector	::=	iri
[17]	WhereClause	::=	'WHERE'? GroupGraphPattern
[18]	SolutionModifier	::=	GroupClause? HavingClause? OrderClause? LimitOffsetClauses?
[19]	GroupClause	::=	'GROUP' 'BY' GroupCondition+
[20]	GroupCondition	::=	BuiltInCall | FunctionCall | '(' Expression ('AS' Var)? ')' | Var
[21]	HavingClause	::=	'HAVING' HavingCondition+
[22]	HavingCondition	::=	Constraint
[23]	OrderClause	::=	'ORDER' 'BY' OrderCondition+
[24]	OrderCondition	::=	(('ASC' | 'DESC') BrackettedExpression) | (Constraint | Var)
[25]	LimitOffsetClauses	::=	LimitClause OffsetClause? | OffsetClause LimitClause?
[26]	LimitClause	::=	'LIMIT' INTEGER
[27]	OffsetClause	::=	'OFFSET' INTEGER
[28]	ValuesClause	::=	('VALUES' DataBlock)?
[29]	Update	::=	Prologue (Update1 (';' Update)?)?
[30]	Update1	::=	Load | Clear | Drop | Add | Move | Copy | Create | InsertData | DeleteData | DeleteWhere | Modify
[31]	Load	::=	'LOAD' 'SILENT'? iri ('INTO' GraphRef)?
[32]	Clear	::=	'CLEAR' 'SILENT'? GraphRefAll
[33]	Drop	::=	'DROP' 'SILENT'? GraphRefAll
[34]	Create	::=	'CREATE' 'SILENT'? GraphRef
[35]	Add	::=	'ADD' 'SILENT'? GraphOrDefault 'TO' GraphOrDefault
[36]	Move	::=	'MOVE' 'SILENT'? GraphOrDefault 'TO' GraphOrDefault
[37]	Copy	::=	'COPY' 'SILENT'? GraphOrDefault 'TO' GraphOrDefault
[38]	InsertData	::=	'INSERT DATA' QuadData
[39]	DeleteData	::=	'DELETE DATA' QuadData
[40]	DeleteWhere	::=	'DELETE WHERE' QuadPattern
[41]	Modify	::=	('WITH' iri)? (DeleteClause InsertClause? | InsertClause) UsingClause* 'WHERE' GroupGraphPattern
[42]	DeleteClause	::=	'DELETE' QuadPattern
[43]	InsertClause	::=	'INSERT' QuadPattern
[44]	UsingClause	::=	'USING' (iri | 'NAMED' iri)
[45]	GraphOrDefault	::=	'DEFAULT' | 'GRAPH'? iri
[46]	GraphRef	::=	'GRAPH' iri
[47]	GraphRefAll	::=	GraphRef | 'DEFAULT' | 'NAMED' | 'ALL'
[48]	QuadPattern	::=	'{' Quads '}'
[49]	QuadData	::=	'{' Quads '}'
[50]	Quads	::=	TriplesTemplate? (QuadsNotTriples '.'? TriplesTemplate?)*
[51]	QuadsNotTriples	::=	'GRAPH' VarOrIri '{' TriplesTemplate? '}'
[52]	TriplesTemplate	::=	TriplesSameSubject ('.' TriplesTemplate?)?

[53]	GroupGraphPattern	::=	'{' (SubSelect | GroupGraphPatternSub) '}'
[54]	GroupGraphPatternSub	::=	TriplesBlock? (GraphPatternNotTriples '.'? TriplesBlock?)*
[55]	TriplesBlock	::=	TriplesSameSubjectPath ('.' TriplesBlock?)?
[56]	GraphPatternNotTriples	::=	GroupOrUnionGraphPattern | OptionalGraphPattern | MinusGraphPattern | GraphGraphPattern | ServiceGraphPattern | Filter | Bind | InlineData
[57]	OptionalGraphPattern	::=	'OPTIONAL' GroupGraphPattern
[58]	GraphGraphPattern	::=	'GRAPH' VarOrIri GroupGraphPattern
[59]	ServiceGraphPattern	::=	'SERVICE' 'SILENT'? VarOrIri GroupGraphPattern
[60]	Bind	::=	'BIND' '(' Expression 'AS' Var ')'
[61]	InlineData	::=	'VALUES' DataBlock
[62]	DataBlock	::=	InlineDataOneVar | InlineDataFull
[63]	InlineDataOneVar	::=	Var '{' DataBlockValue* '}'
[64]	InlineDataFull	::=	(NIL | '(' Var* ')') '{' ('(' DataBlockValue* ')' | NIL)* '}'
[65]	DataBlockValue	::=	iri | RDFLiteral | NumericLiteral | BooleanLiteral | 'UNDEF'
[66]	MinusGraphPattern	::=	'MINUS' GroupGraphPattern
[67]	GroupOrUnionGraphPattern	::=	GroupGraphPattern ('UNION' GroupGraphPattern)*
[68]	Filter	::=	'FILTER' Constraint
[69]	Constraint	::=	BrackettedExpression | BuiltInCall | FunctionCall
[70]	FunctionCall	::=	iri ArgList
[71]	ArgList	::=	NIL | '(' 'DISTINCT'? Expression (',' Expression)* ')'
[72]	ExpressionList	::=	NIL | '(' Expression (',' Expression)* ')'
[73]	ConstructTemplate	::=	'{' ConstructTriples? '}'
[74]	ConstructTriples	::=	TriplesSameSubject ('.' ConstructTriples?)?
[75]	TriplesSameSubject	::=	VarOrTerm PropertyListNotEmpty | TriplesNode PropertyList
[76]	PropertyList	::=	PropertyListNotEmpty?
[77]	PropertyListNotEmpty	::=	Verb ObjectList (';' (Verb ObjectList)?)*
[78]	Verb	::=	VarOrIri | 'a'
[79]	ObjectList	::=	Object (',' Object)*
[80]	Object	::=	GraphNode
[81]	TriplesSameSubjectPath	::=	VarOrTerm PropertyListPathNotEmpty | TriplesNodePath PropertyListPath
[82]	PropertyListPath	::=	PropertyListPathNotEmpty?
[83]	PropertyListPathNotEmpty	::=	(VerbPath | VerbSimple) ObjectListPath (';' ((VerbPath | VerbSimple) ObjectList)?)*
[84]	VerbPath	::=	Path
[85]	VerbSimple	::=	Var
[86]	ObjectListPath	::=	ObjectPath (',' ObjectPath)*
[87]	ObjectPath	::=	GraphNodePath
[88]	Path	::=	PathAlternative
[89]	PathAlternative	::=	PathSequence ('|' PathSequence)*
[90]	PathSequence	::=	PathEltOrInverse ('/' PathEltOrInverse)*
[91]	PathElt	::=	PathPrimary PathMod?
[92]	PathEltOrInverse	::=	PathElt | '^' PathElt
[93]	PathMod	::=	'?' | '*' | '+'
[94]	PathPrimary	::=	iri | 'a' | '!' PathNegatedPropertySet | '(' Path ')'
[95]	PathNegatedPropertySet	::=	PathOneInPropertySet | '(' (PathOneInPropertySet ('|' PathOneInPropertySet)*)? ')'
[96]	PathOneInPropertySet	::=	iri | 'a' | '^' (iri | 'a')
[97]	Integer	::=	INTEGER
[98]	TriplesNode	::=	Collection | BlankNodePropertyList
[99]	BlankNodePropertyList	::=	'[' PropertyListNotEmpty ']'
[100]	TriplesNodePath	::=	CollectionPath | BlankNodePropertyListPath
[101]	BlankNodePropertyListPath	::=	'[' PropertyListPathNotEmpty ']'
[102]	Collection	::=	'(' GraphNode+ ')'
[103]	CollectionPath	::=	'(' GraphNodePath+ ')'
[104]	GraphNode	::=	VarOrTerm | TriplesNode
[105]	GraphNodePath	::=	VarOrTerm | TriplesNodePath
[106]	VarOrTerm	::=	Var | GraphTerm
[107]	VarOrIri	::=	Var | iri
[108]	Var	::=	VAR1 | VAR2
[109]	GraphTerm	::=	iri | RDFLiteral | NumericLiteral | BooleanLiteral | BlankNode | NIL
[110]	Expression	::=	ConditionalOrExpression
[111]	ConditionalOrExpression	::=	ConditionalAndExpression ('||' ConditionalAndExpression)*
[112]	ConditionalAndExpression	::=	ValueLogical ('&&' ValueLogical)*
[113]	ValueLogical	::=	RelationalExpression
[114]	RelationalExpression	::=	NumericExpression ('=' NumericExpression | '!=' NumericExpression | '<' NumericExpression | '>' NumericExpression | '<=' NumericExpression | '>=' NumericExpression | 'IN' ExpressionList | 'NOT' 'IN' ExpressionList)?
[115]	NumericExpression	::=	AdditiveExpression
[116]	AdditiveExpression	::=	MultiplicativeExpression ('+' MultiplicativeExpression | '-' MultiplicativeExpression | (NumericLiteralPositive | NumericLiteralNegative) (('*' UnaryExpression) | ('/' UnaryExpression))*)*
[117]	MultiplicativeExpression	::=	UnaryExpression ('*' UnaryExpression | '/' UnaryExpression)*
[118]	UnaryExpression	::=	'!' PrimaryExpression | '+' PrimaryExpression

			| '-' PrimaryExpression | PrimaryExpression
[119]	PrimaryExpression	::=	BrackettedExpression | BuiltInCall | iriOrFunction | RDFLiteral | NumericLiteral | BooleanLiteral | Var
[120]	BrackettedExpression	::=	'(' Expression ')'
[121]	BuiltInCall	::=	Aggregate | 'STR' '(' Expression ')' | 'LANG' '(' Expression ')' | 'LANGMATCHES' '(' Expression ',' Expression ')' | 'DATATYPE' '(' Expression ')' | 'BOUND' '(' Var ')' | 'IRI' '(' Expression ')' | 'URI' '(' Expression ')' | 'BNODE' ('(' Expression ')' | NIL) | 'RAND' NIL | 'ABS' '(' Expression ')' | 'CEIL' '(' Expression ')' | 'FLOOR' '(' Expression ')' | 'ROUND' '(' Expression ')' | 'CONCAT' ExpressionList | SubstringExpression | 'STRLEN' '(' Expression ')' | StrReplaceExpression | 'UCASE' '(' Expression ')' | 'LCASE' '(' Expression ')' | 'ENCODE_FOR_URI' '(' Expression ')' | 'CONTAINS' '(' Expression ',' Expression ')' | 'STRSTARTS' '(' Expression ',' Expression ')' | 'STRENDS' '(' Expression ',' Expression ')' | 'STRBEFORE' '(' Expression ',' Expression ')' | 'STRAFTER' '(' Expression ',' Expression ')' | 'YEAR' '(' Expression ')' | 'MONTH' '(' Expression ')' | 'DAY' '(' Expression ')' | 'HOURS' '(' Expression ')' | 'MINUTES' '(' Expression ')' | 'SECONDS' '(' Expression ')' | 'TIMEZONE' '(' Expression ')' | 'TZ' '(' Expression ')' | 'NOW' NIL | 'UUID' NIL | 'STRUUID' NIL | 'MD5' '(' Expression ')' | 'SHA1' '(' Expression ')' | 'SHA256' '(' Expression ')' | 'SHA384' '(' Expression ')' | 'SHA512' '(' Expression ')' | 'COALESCE' ExpressionList | 'IF' '(' Expression ',' Expression ',' Expression ')' | 'STRLANG' '(' Expression ',' Expression ')' | 'STRDT' '(' Expression ',' Expression ')' | 'sameTerm' '(' Expression ',' Expression ')' | 'isIRI' '(' Expression ')' | 'isURI' '(' Expression ')' | 'isBLANK' '(' Expression ')' | 'isLITERAL' '(' Expression ')' | 'isNUMERIC' '(' Expression ')' | RegexExpression | ExistsFunc | NotExistsFunc
[122]	RegexExpression	::=	'REGEX' '(' Expression ',' Expression (',' Expression)? ')'
[123]	SubstringExpression	::=	'SUBSTR' '(' Expression ',' Expression (',' Expression)? ')'
[124]	StrReplaceExpression	::=	'REPLACE' '(' Expression ',' Expression ',' Expression (',' Expression)? ')'
[125]	ExistsFunc	::=	'EXISTS' GroupGraphPattern
[126]	NotExistsFunc	::=	'NOT' 'EXISTS' GroupGraphPattern
[127]	Aggregate	::=	'COUNT' '(' 'DISTINCT'? ('*' | Expression) ')' | 'SUM' '(' 'DISTINCT'? Expression ')' | 'MIN' '(' 'DISTINCT'? Expression ')' | 'MAX' '(' 'DISTINCT'? Expression ')' | 'AVG' '(' 'DISTINCT'? Expression ')' | 'SAMPLE' '(' 'DISTINCT'? Expression ')' | 'GROUP_CONCAT' '(' 'DISTINCT'? Expression (';' 'SEPARATOR' '=' String)? ')'
[128]	iriOrFunction	::=	iri ArgList?
[129]	RDFLiteral	::=	String (LANGTAG | ('^^' iri))?
[130]	NumericLiteral	::=	NumericLiteralUnsigned | NumericLiteralPositive | NumericLiteralNegative
[131]	NumericLiteralUnsigned	::=	INTEGER | DECIMAL | DOUBLE
[132]	NumericLiteralPositive	::=	INTEGER_POSITIVE | DECIMAL_POSITIVE | DOUBLE_POSITIVE
[133]	NumericLiteralNegative	::=	INTEGER_NEGATIVE | DECIMAL_NEGATIVE | DOUBLE_NEGATIVE
[134]	BooleanLiteral	::=	'true' | 'false'
[135]	String	::=	STRING_LITERAL1 | STRING_LITERAL2 | STRING_LITERAL_LONG1 | STRING_LITERAL_LONG2
[136]	iri	::=	IRIREF | PrefixedName
[137]	PrefixedName	::=	PNAME_LN | PNAME_NS
[138]	BlankNode	::=	BLANK_NODE_LABEL | ANON

Productions for terminals:

[139]	IRIREF	::=	'<' ([^<>"{}|^`\]-[#x00-#x20])* '>'
[140]	PNAME_NS	::=	PN_PREFIX? ':'
[141]	PNAME_LN	::=	PNAME_NS PN_LOCAL
[142]	BLANK_NODE_LABEL	::=	'_:' (PN_CHARS_U | [0-9]) ((PN_CHARS|'.')* PN_CHARS)?
[143]	VAR1	::=	'?' VARNAME
[144]	VAR2	::=	'$' VARNAME
[145]	LANGTAG	::=	'@' [a-zA-Z]+ ('-' [a-zA-Z0-9]+)*
[146]	INTEGER	::=	[0-9]+
[147]	DECIMAL	::=	[0-9]* '.' [0-9]+
[148]	DOUBLE	::=	[0-9]+ '.' [0-9]* EXPONENT | '.' ([0-9])+ EXPONENT | ([0-9])+ EXPONENT

[149]	INTEGER_POSITIVE	::=	'+' INTEGER
[150]	DECIMAL_POSITIVE	::=	'+' DECIMAL
[151]	DOUBLE_POSITIVE	::=	'+' DOUBLE
[152]	INTEGER_NEGATIVE	::=	'-' INTEGER
[153]	DECIMAL_NEGATIVE	::=	'-' DECIMAL
[154]	DOUBLE_NEGATIVE	::=	'-' DOUBLE
[155]	EXPONENT	::=	[eE] [+-]? [0-9]+
[156]	STRING_LITERAL1	::=	"'" (([^#x27#x5C#xA#xD]) | ECHAR)* "'"
[157]	STRING_LITERAL2	::=	'"' (([^#x22#x5C#xA#xD]) | ECHAR)* '"'
[158]	STRING_LITERAL_LONG1	::=	"'''" (("'" | "''")? ([^'\] | ECHAR))* "'''"
[159]	STRING_LITERAL_LONG2	::=	'"""' (('"' | '""')? ([^"\] | ECHAR))* '"""'
[160]	ECHAR	::=	'\' [tbnrf\"']
[161]	NIL	::=	'(' WS* ')'
[162]	WS	::=	#x20 | #x9 | #xD | #xA
[163]	ANON	::=	'[' WS* ']'
[164]	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]
[165]	PN_CHARS_U	::=	PN_CHARS_BASE | '_'
[166]	VARNAME	::=	(PN_CHARS_U | [0-9]) (PN_CHARS_U | [0-9] | #x00B7 | [#x0300-#x036F] | [#x203F-#x2040])*
[167]	PN_CHARS	::=	PN_CHARS_U | '-' | [0-9] | #x00B7 | [#x0300-#x036F] | [#x203F-#x2040]
[168]	PN_PREFIX	::=	PN_CHARS_BASE ((PN_CHARS|'.')* PN_CHARS)?
[169]	PN_LOCAL	::=	(PN_CHARS_U | ':' | [0-9] | PLX) ((PN_CHARS | '.' | ':' | PLX)* (PN_CHARS | ':' | PLX))?
[170]	PLX	::=	PERCENT | PN_LOCAL_ESC
[171]	PERCENT	::=	'%' HEX HEX
[172]	HEX	::=	[0-9] | [A-F] | [a-f]
[173]	PN_LOCAL_ESC	::=	'\' ('_' | '~' | '.' | '-' | '!' | '$' | '&' | "'" | '(' | ')' | '*' | '+' | ',' | ';' | '=' | '/' | '?' | '#' | '@' | '%')

20 Conformance

See Section 19 SPARQL Grammar regarding conformance of SPARQL Query strings, and section 16 Query Forms for conformance of query results. See section 22. Internet Media Type for conformance to the application/sparql-query media type.

This specification is intended for use in conjunction with the SPARQL 1.1 Protocol [SPROT], the SPARQL Query Results XML Format [SPARQL XML Results], the SPARQL 1.1 Query Results JSON Format [SPARQL-JSON-Results] and the SPARQL 1.1 Query Results CSV and TSV Formats [SPARQL CSV and TSV Results]. See those specifications for their conformance criteria.

Note that the SPARQL protocol describes a means for conveying SPARQL queries to an SPARQL query processing service and returning the query results to the entity that requested them.

21 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.

SPARQL requests may cause additional requests to be issued from the SPARQL endpoint, such as FROM NAMED. The endpoint is potentially within an organisations firewall or DMZ, and so such queries may be a source of indirection attacks.

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.

22 Internet Media Type, File Extension and Macintosh File Type

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" (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 1.1 specification is a work product of the World Wide Web Consortium's SPARQL Working Group. The W3C has change control over these specifications.

A References

A.1 Normative References

[CHARMOD]
Character Model for the World Wide Web 1.0: Fundamentals, R. Ishida, F. Yergeau, M. J. Dürst, 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/. Updated 2012 by W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. (Latest version available at http://www.w3.org/TR/xmlschema11-2/).

[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 .

A.2 Other 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.

[SPARQL XML Results]
SPARQL Query Results XML Format (Second Edition), D. Beckett, J. Broekstra, Editors, W3C Recommendation, 21 March 2013, http://www.w3.org/TR/2013/REC-rdf-sparql-XMLres-20130321. Latest version available at http://www.w3.org/TR/rdf-sparql-XMLres.

[SPARQL JSON Results]
SPARQL 1.1 Query Results JSON Format, A. Seaborne, Editor, W3C Recommendation, 21 March 2013, http://www.w3.org/TR/2013/REC-sparql11-results-json-20130321. Latest version available at http://www.w3.org/TR/sparql11-results-json.

[SPARQL CSV and TSV Result]
SPARQL 1.1 Query Results CSV and TSV Formats, A. Seaborne, Editor, W3C Recommendation, 21 March 2013, http://www.w3.org/TR/2013/REC-sparql11-results-csv-tsv-20130321. Latest version available at http://www.w3.org/TR/sparql11-results-csv-tsv.

[SPROT]
SPARQL 1.1 Protocol, L. Feigenbaum, G. Williams, K. Clark, E. Torres, Editors, W3C Recommendation, 21 March 2013, http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321. Latest version available at http://www.w3.org/TR/sparql11-protocol.

[TURTLE]
Turtle: Terse RDF Triple Language, E Prud'hommeaux, G Carothers, Editors, W3C Candidate Recommendation, 19 February 2013, http://www.w3.org/TR/2013/CR-turtle-20130219/. Latest version available at http://www.w3.org/TR/turtle/.

[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/ .

[UCNR2]
SPARQL New Features and Rationale, Kjetil Kjernsmo, Alexandre Passant, Editors, W3C Working Draft, 2 July 2009, http://www.w3.org/TR/2009/WD-sparql-features-20090702/ . Latest version available at http://www.w3.org/TR/sparql-features/ .

[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/ .

Change Log

Changes since Proposed Recommendation

- Fixed error in example of inverse property path

Changes since Last Call

The following are the corrections made since last publication:

- Grammar: `DISTINCT` for paths had been left in the grammar - removed.
- Restore translation of `BIND` as per text in previous publications (first and second last call).

Since SPARQL 1.0

The new features in SPARQL 1.1 Query are:

- Aggregates
- Subqueries
- Negation
- Expressions in the SELECT clause
- Property Paths
- Assignment
- A short form for CONSTRUCT
- An expanded set of functions and operators