SPARQL 1.1 Update

W3C Recommendation 21 March 2013

This version:
http://www.w3.org/TR/2013/REC-sparql11-update-20130321/
Latest version:
http://www.w3.org/TR/sparql11-update/
Previous version:
http://www.w3.org/TR/2012/PR-sparql11-update-20121108/
Editors:
Paul Gearon <pgearon@revelytix.com>
Alexandre Passant, DERI Galway at the National University of Ireland, Galway, Ireland
<alexandre.passant@deri.org>
Axel Polleres, Siemens AG <axel.polleres@siemens.com>

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

See also **translations**.

Abstract

This document describes SPARQL 1.1 Update, an update language for RDF graphs. It uses a syntax derived from the SPARQL Query Language for RDF. Update operations are performed on a collection of graphs in a Graph Store. Operations are provided to update, create, and remove RDF graphs in a Graph Store.

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May Be Superseded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
3. SPARQL 1.1 Update (this document)
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
  1.1 Document Conventions
    1.1.1 Language Form
    1.1.2 Terminology
2 The Graph Store
  2.1 Graph Store and SPARQL Query Services
  2.2 SPARQL 1.1 Update Services
  2.3 Entailment and Consistency
3 SPARQL 1.1 Update Language
  3.1 Graph Update
    3.1.1 INSERT DATA
    3.1.2 DELETE DATA
    3.1.3 DELETE/INSERT
      3.1.3.1 DELETE (Informative)
      3.1.3.2 INSERT (Informative)
      3.1.3.3 DELETE WHERE
    3.1.4 LOAD
    3.1.5 CLEAR
  3.2 Graph Management
    3.2.1 CREATE
    3.2.2 DROP
    3.2.3 COPY
    3.2.4 MOVE
    3.2.5 ADD
4 SPARQL Update Formal Model
  4.1 General Definitions
    4.1.1 Graph Store
    4.1.2 Abstract Update Operation
  4.2 Auxiliary Definitions
    4.2.1 Dataset-UNION
    4.2.2 Dataset-DIFF
    4.2.3 Dataset(QuadPattern, μ, DS, GS)
    4.2.4 Dataset(QuadPattern, P, DS, GS)
  4.3 Graph Update Operations
    4.3.1 Insert Data Operation
    4.3.2 Delete Data Operation
    4.3.3 Delete Insert Operation
    4.3.4 Load Operation
    4.3.5 Clear Operation
  4.4 Graph Management Operations
    4.4.1 Create Operation
    4.4.2 Drop Operation
  4.5 Mapping Update Requests to the Formal Model
5 Conformance

Appendices

A Security Considerations (Informative)
B Internet Media Type, File Extension and Macintosh File Type
C SPARQL 1.1 Update Grammar
D References
  D.1 Normative References
  D.2 Other References

---

1 Introduction

SPARQL 1.1 Update is intended to be a standard language for specifying and executing updates to RDF graphs in a Graph Store.

SPARQL 1.1 Update provides the following facilities:

- Insert triples into an RDF graph in the Graph Store.
- Delete triples from an RDF graph in the Graph Store.
- Load an RDF graph into the Graph Store.
- Clear an RDF graph in the Graph Store.
- Create a new RDF graph in a Graph Store.

- Drop an RDF graph from a Graph Store.
- Copy, move, or add the content of one RDF graph in the Graph Store to another.
- Perform a group of update operations as a single action.

This document is particularly related to the following other specification documents:

- SPARQL 1.1 Query Language
- SPARQL 1.1 Graph Store HTTP Protocol
- SPARQL 1.1 Protocol for RDF

SPARQL 1.1 Update is a companion language and envisaged to be used in conjunction with the *SPARQL 1.1 Query language*. The present document refers to the grammar and several definitions from the *SPARQL 1.1 Query language* specification.

The *SPARQL 1.1 Graph Store HTTP Protocol* specification employs the HTTP protocol to perform update operations using standard HTTP methods, such as PUT and DELETE. While providing a simple and well known API, it is necessarily restricted in its operations due to the limited set of methods in the HTTP protocol. In contrast, SPARQL 1.1 Update permits multiple modifications in a single operation, and can use complex SPARQL queries for constructing data to be inserted, or choosing data to be deleted. Also, the use of an update language facilitates operations over proprietary APIs and connections that may not involve HTTP.

The *SPARQL 1.1 Protocol for RDF* specification describes a means of conveying SPARQL 1.1 Query and SPARQL 1.1 Update operations from clients to a SPARQL query processing service, and for returning appropriate results. Together with the SPARQL 1.1 Query and SPARQL 1.1 Update (this document) specifications, these form an alternative to the *SPARQL 1.1 Graph Store HTTP Protocol* with comprehensive, though more complex functionality.

1.1 Document Conventions

1.1.1 Language Form

The operations in this document contain language forms describing their use. These are meant as illustrative forms of the formal grammar described in the SPARQL 1.1 Query document. Any discrepancies between the language forms in this document and the grammar in SPARQL 1.1 Query will defer to the formal grammar in SPARQL 1.1 Query.

Language forms are shown informally in this document as for instance:

```
( WITH IRIref  )?
( ( DeleteClause InsertClause? ) | InsertClause )
( USING ( NAMED )? IRIref )*
WHERE GroupGraphPattern
```

Unlike other forms of EBNF where square brackets denote optionality, here `[]` is used for blank nodes, as in SPARQL Query. `|` is used to denote alternatives, `()` is used for grouping terms, `?` indicates 0 or 1 occurrence of a term, `*` indicates 0 or more occurrences, and `+` indicates 1 or more occurrences.

BOLD indicates language keywords. *Italics* indicate syntactic items defined in the SPARQL 1.1 Query Grammar, where we occasionally refer to productions by links. Unitalicized text indicates a local term that will have a more complex (and exact) definition in the formal grammar.

Example update requests are shown as follows:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT { <http://example/egbook> dc:title  "This is an example title" } WHERE {}
```

Note:

PREFIX definitions and the syntax for IRIs in update requests in general follow the same conventions as in the SPARQL1.1 Query Language.

Data is shown in Turtle syntax as follows:

```
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix :     <http://example.org/books/> .
:book0  dc:title  "SPARQL Tutorial" .
```

1.1.2 Terminology

When this document uses the words MUST, MUST NOT, SHOULD, SHOULD NOT, MAY and **recommended**, and the words appear as emphasized text, they must be interpreted as described in RFC 2119 [RFC2119].

The following terms are also in use throughout this document:

- Operation - An action to be performed that results in the modification of data in a Graph Store expressible as a single command, e.g. INSERT or DELETE.
- Request - A sequence of zero or more operations that is sent to a Graph Store. When using the SPARQL 1.1 Protocol for RDF a request will be one HTTP POST.

The following terms are also used in this document as defined in the SPARQL 1.1 Query Language:

- *QuadPattern* - A syntactic construct that refers to a set of triple patterns, similar to *ConstructTriples*, but potentially

involving the GRAPH keyword to indicate that a set of triples is to be inserted into/deleted from a named graph.
- *QuadData* - A *QuadPattern* without variables.
- *GroupGraphPattern* - A syntactic construct for referring to a set of triples, possibly with complex constraints.

2 The Graph Store

A Graph Store is a mutable container of RDF graphs managed by a single service. Similar to an RDF Dataset operated on by the SPARQL 1.1 Query Language, a Graph Store contains one (unnamed) slot holding a *default graph* and zero or more named slots holding *named graphs*. Operations MAY specify graphs to be modified, or they MAY rely on a *default graph* for that operation. Unless overridden (for instance, by the SPARQL protocol), the unnamed graph for the store will be the default graph for any operations on that store. Depending on implementation, the unnamed graph MAY refer to a separate graph, a graph describing the named graphs, a representation of a union of other graphs, etc.

Unlike an RDF Dataset, named graphs can be added to or deleted from a Graph Store. A Graph Store needs not be authoritative for the graphs it contains. That means a Graph Store can keep local copies of RDF graphs defined elsewhere on the Web and modify those copies independently of the original graph.

In the simple case where there is one unnamed graph and no named graphs, SPARQL 1.1 Update can be used as a graph update language (as opposed to a Graph Store update language).

The information how a Graph Store is accessed is defined in the protocol and Graph Store protocol specs. A Graph Store is accessible by either an update service (cf. protocol) or via the Graph Store protocol (cf. Graph Store protocol). In either case the Graph Store is hidden behind the service, making it accessible via the URI of a SPARQL update service or via a URI that responds to the Graph Store protocol.

A formal definition for Graph Stores and how SPARQL 1.1 Update affects them is described in the SPARQL 1.1 Update Formal Model section.

2.1 Graph Store and SPARQL Query Services

A service (often referred to by the informal term *SPARQL endpoint*) that accepts and processes update requests is referred to as an *update service*. There is no presumption that the Graph Store managed by an update service exactly corresponds to any RDF Dataset offered by some query service. A query service MAY offer an RDF Dataset formed from graphs that are part of an update service's Graph Store. The graphs in the query service's RDF Dataset MAY be a subset of the graphs in the update service's Graph Store. Furthermore, the query service's RDF Dataset and the update service's Graph Store MAY use different names for the same graphs.

2.2 SPARQL 1.1 Update Services

SPARQL 1.1 Update requests are sequences of operations.

Each request SHOULD be treated atomically by a SPARQL 1.1 Update service. The term 'atomically' means that a single request will result in either no effect or a complete effect, regardless of the number of operations that may be present in the request. Any resulting concurrency issues will be a matter for each implementation to consider according to its own architecture. In particular, using the SERVICE keyword in the WHERE clause of operations in an Update request will usually result in a loss of atomicity.

In the case of two different update services, whose respective Graph Stores contain graphs with the same names, there is no presumption that the updates done through one service will be propagated to the other, as the stores are independent entities. The behaviour of these services with respect to each other (such as automatic synchronization after updates) is implementation dependent.

2.3 Entailment and Consistency

If the store is capable of calculating entailed answers, see SPARQL 1.1 Entailment Regimes, then it is possible for update operations to interact with entailed data. In particular, a DELETE operation may attempt to remove entailed statements without actual effects.

After an update request is completed, a store that performs consistency checking with respect to a particular entailment regime on its graphs MAY check the new state of the Graph Store for consistency. If inconsistency is detected, such a store MAY return an error to the request.

Also of note is that some stores may be capable of performing entailments with respect to an ontology capable of higher level processing, such as RDFS or OWL. Updates may interact with these entailment regimes in these systems.

3 SPARQL 1.1 Update Language

SPARQL 1.1 Update supports two categories of update operations on a Graph Store:

- Graph Update - addition and removal of triples from some graphs within the Graph Store.
- Graph Management - creating and deletion of graphs within the Graph Store, as well as convenient shortcuts for graph update operations often used during graph management (to add, move, and copy graphs).

A request is a sequence of operations and is terminated by EOF (End of File). Multiple operations are separated by a ';' (semicolon) character. A semicolon after the last operation in a request is optional. Implementations MUST ensure that the operations of a single request are executed in a fashion that guarantees the same effects as executing them sequentially in the order they appear in the request.

Operations all result either in *success* or *failure*. A *failure* result MAY be accompanied by extra information, indicating that

some portion of the operations in the request were successful. This document does not stipulate the exact form of the result, as that will be dependent on the interface being used, for instance the SPARQL 1.1 protocol via HTTP or a programmatic API. If multiple operations are present in a single request, then a result of *failure* from any operation MUST abort the sequence of operations, causing the subsequent operations to be ignored.

The formal semantics of the following operations is defined in Section 4 of this document.

3.1 Graph Update

Graph update operations change existing graphs in the Graph Store but do not explicitly delete nor create them. Non-empty inserts into non-existing graphs will, however, *implicitly* create those graphs, i.e., an implementation fulfilling an update request SHOULD silently an automatically create graphs that do not exist before triples are inserted into them, and MUST return with failure if it fails to do so for any reason. (For example, the implementation may have insufficient resources, or an implementation may only provide an update service over a fixed set of graphs and the implicitly created graph is not within this fixed set). An implementation MAY remove graphs that are left empty after triples are removed from them.

If a graph is created implicitly by an update operation, then the behavior of the Graph Store MUST be functionally equivalent to its behavior if the graph had been created explicitly by a CREATE operation.

SPARQL 1.1 Update provides these graph update operations:

- The INSERT DATA operation adds some triples, given inline in the request, into a graph. This SHOULD create the destination graph if it does not exist. If the graph does not exist and it can not be created for any reason, then a failure MUST be returned.
- The DELETE DATA operation removes some triples, given inline in the request, if the respective graph contains those.
- The fundamental pattern-based actions for graph updates are INSERT and DELETE (which can co-occur in a single DELETE/INSERT operation). These actions consist of groups of triples to be deleted and groups of triples to be added. The specification of the triples is based on query patterns. The difference between INSERT / DELETE and INSERT DATA / DELETE DATA is that INSERT DATA and DELETE DATA do not substitute bindings into a template from a pattern. The DATA forms require concrete data (triple templates containing variables within DELETE DATA and INSERT DATA operations are disallowed and blank nodes are disallowed within DELETE DATA, see Notes 8+9 in the grammar). Having specific operations for concrete data means that a request can be streamed so that large, pure-data updates can be done.
- The LOAD operation reads the contents of a document representing a graph into a graph in the Graph Store.
- The CLEAR operation removes all the triples in (one or more) graphs.

3.1.1 INSERT DATA

The INSERT DATA operation adds some triples, given inline in the request, into the Graph Store:

```
INSERT DATA  QuadData
```

where *QuadData* are formed by *TriplesTemplate*s, i.e., sets of triple patterns, optionally wrapped into a GRAPH block.

```
( GRAPH  VarOrIri )? { TriplesTemplate? }
```

Variables in *QuadData*s are disallowed in INSERT DATA requests (see Notes 8 in the grammar). That is, the INSERT DATA statement only allows to insert ground triples. Blank nodes in *QuadData*s are assumed to be disjoint from the blank nodes in the Graph Store, i.e., will be inserted with "fresh" blank nodes.

If no graph is described in the *QuadData*, then the default graph is presumed. If data is inserted into a graph that does not exist in the Graph Store, it SHOULD be created (there may be implementations providing an update service over a fixed set of graphs which in such case MUST return with failure for update requests that insert data into an unallowed graph).

Note that a triple MAY be considered to be "processed" with no action if that triple already exists in the graph. Further, note that

```
INSERT DATA { GRAPH <g> {} } ...
```

does not create <g>. If a user intends to just create a graph, then the graph management operations (CREATE/LOAD) may be used prior to any insertion operations.

Example 1: Adding some triples to a graph

This snippet describes two RDF triples to be inserted into the default graph of the Graph Store.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA
{
  <http://example/book1> dc:title "A new book" ;
                         dc:creator "A.N.Other" .
}
```

Data before:

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

<http://example/book1> ns:price 42 .
```

Data after:

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

<http://example/book1> ns:price 42 .
<http://example/book1> dc:title "A new book" .
<http://example/book1> dc:creator "A.N.Other" .
```

Example 2:

This SPARQL 1.1 Update request adds a triple to provide the price of a book. As opposed to the previous example, which affected the default graph, the requested change happens in the named graph identified by the IRI `http://example/bookStore`.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>
INSERT DATA
{ GRAPH <http://example/bookStore> { <http://example/book1>  ns:price  42 } }
```

Data before:

```
# Graph: http://example/bookStore
@prefix dc: <http://purl.org/dc/elements/1.1/> .
<http://example/book1> dc:title "Fundamentals of Compiler Design" .
```

Data after:

```
# Graph: http://example/bookStore
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix ns: <http://example.org/ns#> .
<http://example/book1> dc:title "Fundamentals of Compiler Design" .
<http://example/book1> ns:price 42 .
```

3.1.2 DELETE DATA

The DELETE DATA operation removes some triples, given inline in the request, if the respective graphs in the Graph Store contain those:

```
DELETE DATA  QuadData
```

QuadData denotes triples to be removed and is as described in INSERT DATA, with the difference that in a DELETE DATA operation neither variables nor blank nodes are allowed (see Notes 8+9 in the grammar).

As with INSERT DATA, DELETE DATA is meant for deletion of ground triples data which is why *QuadData* that contains variables or blank nodes is disallowed in DELETE DATA operations. The DELETE/INSERT operation can be used to remove triples containing blank nodes.

Note that the deletion of non-existing triples has no effect, i.e., triples in the *QuadData* that did not exist in the Graph Store are ignored. Blank nodes are not permitted in the *QuadData*, as these do not match any existing data.

Example 3: Removing triples from a graph

This request describes 2 triples to be removed from the default graph of the Graph Store.

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

DELETE DATA
{
  <http://example/book2> dc:title "David Copperfield" ;
                         dc:creator "Edmund Wells" .
}
```

Data before:

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

<http://example/book2> ns:price 42 .
<http://example/book2> dc:title "David Copperfield" .
<http://example/book2> dc:creator "Edmund Wells" .
```

Data after:

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

<http://example/book2> ns:price 42 .
```

Example 4:

This SPARQL 1.1 Update request consists of two operations, including a triple to be deleted and a triple to be added (used here to correct a book title). As opposed to the previous example, which affected the default graph, the requested change happens in the named graph identified by the IRI `http://example/bookStore`.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
DELETE DATA
{ GRAPH <http://example/bookStore> { <http://example/book1>  dc:title  "Fundamentals of Compiler Desing" } } ;

PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA
{ GRAPH <http://example/bookStore> { <http://example/book1>  dc:title  "Fundamentals of Compiler Design" } }
```

Data before:

```
# Graph: http://example/bookStore
@prefix dc: <http://purl.org/dc/elements/1.1/> .
<http://example/book1> dc:title "Fundamentals of Compiler Desing" .
```

Data after:

```
# Graph: http://example/bookStore
@prefix dc: <http://purl.org/dc/elements/1.1/> .
<http://example/book1> dc:title "Fundamentals of Compiler Design" .
```

3.1.3 DELETE/INSERT

The DELETE/INSERT operation can be used to remove or add triples from/to the Graph Store based on bindings for a query pattern specified in a WHERE clause:

```
( WITH IRIref )?
( ( DeleteClause InsertClause? ) | InsertClause )
( USING ( NAMED )? IRIref )*
WHERE GroupGraphPattern
```

The *DeleteClause* and *InsertClause* forms can be broken down as follows:

```
DeleteClause ::= DELETE  QuadPattern
InsertClause ::= INSERT  QuadPattern
```

This operation identifies data with the WHERE clause, which will be used to compute solution sequences of bindings for a set of variables. The bindings for each solution are then substituted into the DELETE template to remove triples, and then in the INSERT template to create new triples. If any solution produces a triple containing an unbound variable or an illegal RDF construct, such as a literal in a subject or predicate position, then that triple is not included when processing the operation: INSERT will not instantiate new data in the output graph, and DELETE will not remove anything. The graphs used for computing a solution sequence may be different to the graphs modified with the DELETE and INSERT templates.

The WITH clause defines the graph that will be modified or matched against for any of the subsequent elements (in DELETE, INSERT, or WHERE clauses) if they do not specify a graph explicitly. If not provided, then the default graph of the Graph Store (or an explicitly declared dataset in the WHERE clause) will be assumed. That is, a WITH clause may be viewed as syntactic sugar for wrapping both the *QuadPattern*s in subsequent DELETE and INSERT clauses, and likewise the *GroupGraphPattern* in the subsequent WHERE clause into GRAPH patterns. This can be used to avoid refering to the same graph multiple times in a single operation.

Following the optional WITH clause are the INSERT and/or DELETE clauses. The deletion of the triples happens before the insertion. The pattern in the WHERE clause is evaluated only once, before the delete part of the operation is performed. The overall processing model is that the pattern is executed, the results used to instantiate the DELETE template, the deletes performed, the results used again to instantiate the INSERT template, and the inserts performed.

If the DELETE clause is omitted, then the operation only inserts data (see INSERT). If the INSERT clause is omitted, then the operation only removes data (see DELETE). The grammar does not permit both DELETE and INSERT to be omitted in the same operation.

The USING and USING NAMED clauses affect the RDF Dataset used while evaluating the WHERE clause. This describes a dataset in the same way as FROM and FROM NAMED clauses describe RDF Datasets in the SPARQL 1.1 Query Language. The keyword USING instead of FROM in update requests is to avoid possible ambiguities which could arise from writing "DELETE FROM". That is, the *GroupGraphPattern* in the WHERE clause will be matched against the dataset described by explicit USING or USING NAMED clauses, if specified, and against the Graph Store otherwise.

The WITH clause provides a convenience for when an operation primarily refers to a single graph. If a graph name is specified in a WITH clause, then - for the purposes of evaluating the WHERE clause - this will define an RDF Dataset containing a default graph with the specified name, but only in the absence of USING or USING NAMED clauses. In the presence of one or more graphs referred to in USING clauses and/or USING NAMED clauses, the WITH clause will be ignored

while evaluating the WHERE clause.

The *GroupGraphPattern* in the WHERE clause is evaluated as in a SPARQL query "SELECT * WHERE *GroupGraphPattern*" and all the solution bindings are applied to the preceding DELETE and INSERT templates for defining the triples to be deleted from or inserted into the Graph Store.

Again, *QuadPattern*s are formed by *TriplesTemplate*s, i.e., sets of triple patterns, optionally wrapped into a GRAPH block, where the GRAPH clause indicates the named graph in the Graph Store to be updated; on any *TripleTemplate*s without a GRAPH clause, the INSERT or DELETE clauses applies to the graph specified by the WITH clause, or the default graph of the Graph Store if no WITH clause is present.

To illustrate the use of the WITH clause, an operation of the general form:

```
WITH <g1> DELETE { a b c } INSERT { x y z } WHERE { ... }
```

is considered equivalent to:

```
DELETE { GRAPH <g1> { a b c } } INSERT { GRAPH <g1> { x y z } } USING <g1> WHERE { ... }
```

Note that explicit GRAPH clauses override a WITH clause. WITH provides a fallback to specify a graph (different from the default graph) to use when one is not explicitly stipulated via GRAPH.

Deleting triples that are not present, or from a graph that is not present will have no effect and will result in *success*. Blank nodes are prohibited in a DELETE template, since using a new blank node in a DELETE template would lead to nothing being deleted, as a new blank node cannot match anything in the Graph Store. It should be noted that this restriction is not in the EBNF for the *DeleteClause* itself, but made explicit in Note 9 to the grammar.

If an operation tries to insert into a graph that does not exist, then that graph SHOULD be created; again, there may be implementations providing an update service over a fixed set of graphs which in such case MUST return with failure for update requests that would create an unallowed graph. If no data is to be inserted, then no graph will be created. Particularly, note that

```
INSERT ... { GRAPH <g> {} } ...
```

does not create <g>. If a user intends to create a graph regardless of the data to be inserted, then the graph management operations (CREATE/LOAD) may be used prior to any insertion operations.

Blank nodes that appear in an INSERT clause operate similarly to blank nodes in the template of a CONSTRUCT query, i.e., they are re-instantiated for any solution of the WHERE clause; refer to Templates with Blank Nodes in SPARQL Query 1.1 and to the formal semantics of DELETE/INSERT below for details. Blank nodes in the WHERE clause match data in the same way as for any SPARQL Query.

Example 5:

An example to update the graph `http://example/addresses` to rename all people with the given name "Bill" to "William".

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

WITH <http://example/addresses>
DELETE { ?person foaf:givenName 'Bill' }
INSERT { ?person foaf:givenName 'William' }
WHERE
  { ?person foaf:givenName 'Bill'
  }
```

Data before:

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

<http://example/president25> foaf:givenName "Bill" .
<http://example/president25> foaf:familyName "McKinley" .
<http://example/president27> foaf:givenName "Bill" .
<http://example/president27> foaf:familyName "Taft" .
<http://example/president42> foaf:givenName "Bill" .
<http://example/president42> foaf:familyName "Clinton" .
```

Data after:

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

<http://example/president25> foaf:givenName "William" .
<http://example/president25> foaf:familyName "McKinley" .
<http://example/president27> foaf:givenName "William" .
<http://example/president27> foaf:familyName "Taft" .
<http://example/president42> foaf:givenName "William" .
<http://example/president42> foaf:familyName "Clinton" .
```

3.1.3.1 DELETE (Informative)

```
( WITH  IRIref )?
DELETE  QuadPattern
( USING ( NAMED )? IRIref )*
WHERE GroupGraphPattern
```

The DELETE operation is a form of the DELETE/INSERT operation having no INSERT section. A compliant implementation of DELETE/INSERT will already implement this operation correctly. The DELETE operation is described here separately for clarity. Analogous to DELETE/INSERT, deleting triples that are not present, or from a graph that is not present will have no effect and will result in success.

If any DELETE template specifies a GRAPH then this will be the graph affected. Otherwise, the operation will be applied to the graph specified in the WITH clause, if one was specified, or the default graph otherwise.

The WHERE clause identifies data in existing graphs, and creates bindings to be used by the template. The graphs to apply the *GroupGraphPattern* follow the same rules as for DELETE/INSERT.

Example 6:

This example request deletes all records of old books (with date before year 1970) from the store's default graph:

```
PREFIX dc:  <http://purl.org/dc/elements/1.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

DELETE
 { ?book ?p ?v }
WHERE
 { ?book dc:date ?date .
   FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime )
   ?book ?p ?v
 }
```

The pattern in WHERE is matched against the Graph Store. The resulting sequence of solutions to the WHERE clause is used to instantiate the triple patterns in the DELETE template similar to CONSTRUCT in SPARQL 1.1 Query. The resulting triples are then removed from the Graph Store.

Data before:

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

<http://example/book1> dc:title "Principles of Compiler Design" .
<http://example/book1> dc:date "1977-01-01T00:00:00-02:00"^^xsd:dateTime .

<http://example/book2> ns:price 42 .
<http://example/book2> dc:title "David Copperfield" .
<http://example/book2> dc:creator "Edmund Wells" .
<http://example/book2> dc:date "1948-01-01T00:00:00-02:00"^^xsd:dateTime .

<http://example/book3> dc:title "SPARQL 1.1 Tutorial" .
```

Data after:

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

<http://example/book2> ns:price 42 .
<http://example/book2> dc:title "David Copperfield" .
<http://example/book2> dc:creator "Edmund Wells" .
<http://example/book2> dc:date "1948-01-01T00:00:00-02:00"^^xsd:dateTime .

<http://example/book3> dc:title "SPARQL 1.1 Tutorial" .
```

Example 7:

This example request removes all statements about anything with a given name of "Fred" from the graph `http://example/addresses`. A WITH clause is present, meaning that graph `http://example/addresses` is both the one from which triples are being removed and the one which the WHERE clause is matched against.

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

WITH <http://example/addresses>
DELETE { ?person ?property ?value }
WHERE { ?person ?property ?value ; foaf:givenName 'Fred' }
```

Data before:

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

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .
<http://example/william> foaf:mbox <mailto:bill@example> .

<http://example/fred> a foaf:Person .
<http://example/fred> foaf:givenName "Fred" .
<http://example/fred> foaf:mbox  <mailto:fred@example> .
```

Data after:

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

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .
<http://example/william> foaf:mbox  <mailto:bill@example> .
```

Another example of DELETE is provided in the final example in the following section which demonstrates multiple operations combining an INSERT with a DELETE.

3.1.3.2 INSERT (Informative)

```
( WITH  IRIref )?
INSERT  QuadPattern
( USING ( NAMED )?  IRIref )*
WHERE GroupGraphPattern
```

The INSERT operation is a form of the DELETE/INSERT operation having no DELETE section. A compliant implementation of DELETE/INSERT will already implement this operation correctly. The INSERT operation is described here separately for clarity.

If the INSERT template specifies GRAPH blocks then these will be the graphs affected. Otherwise, the operation will be applied to the default graph, or, respectively, to the graph specified in the WITH clause, if one was specified. If no USING (NAMED) clause is present, then the pattern in the WHERE clause will be matched against the Graph Store, otherwise against the dataset specified by the USING (NAMED) clauses. The matches against the WHERE clause create bindings to be applied to the template for determining triples to be inserted (following the same rules as for DELETE/INSERT).

If any instantiation arising from the solution sequence 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 inserted. The template can contain triples with no variables (known as ground or explicit triples), and these will also be inserted, provided that the solution sequence is not empty.

Example 8:

This example copies triples from one named graph to another named graph based on a pattern:

```
PREFIX dc:  <http://purl.org/dc/elements/1.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

INSERT
  { GRAPH <http://example/bookStore2> { ?book ?p ?v } }
WHERE
  { GRAPH  <http://example/bookStore>
       { ?book dc:date ?date .
         FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime )
         ?book ?p ?v
  } }
```

Data before:

```
# Graph: http://example/bookStore
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example/book1> dc:title "Fundamentals of Compiler Design" .
<http://example/book1> dc:date "1977-01-01T00:00:00-02:00"^^xsd:dateTime .

<http://example/book2> ns:price 42 .
<http://example/book2> dc:title "David Copperfield" .
<http://example/book2> dc:creator "Edmund Wells" .
<http://example/book2> dc:date "1948-01-01T00:00:00-02:00"^^xsd:dateTime .

<http://example/book3> dc:title "SPARQL 1.1 Tutorial" .
```

```
# Graph: http://example/bookStore2
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://example/book4> dc:title "SPARQL 1.0 Tutorial" .
```

Data after:

```
# Graph: http://example/bookStore
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example/book1> dc:title "Fundamentals of Compiler Design" .
<http://example/book1> dc:date "1977-01-01T00:00:00-02:00"^^xsd:dateTime .

<http://example/book2> ns:price 42 .
<http://example/book2> dc:title "David Copperfield" .
<http://example/book2> dc:creator "Edmund Wells" .
<http://example/book2> dc:date "1948-01-01T00:00:00-02:00"^^xsd:dateTime .

<http://example/book3> dc:title "SPARQL 1.1 Tutorial" .
```

```
# Graph: http://example/bookStore2
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example/book1> dc:title "Fundamentals of Compiler Design" .
<http://example/book1> dc:date "1977-01-01T00:00:00-02:00"^^xsd:dateTime .

<http://example/book4> dc:title "SPARQL 1.0 Tutorial" .
```

Example 9:

This example copies triples of name and email from one named graph to another. Some individuals may not have an address, but the name is copied regardless:

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

INSERT
  { GRAPH <http://example/addresses>
    {
      ?person  foaf:name  ?name .
      ?person  foaf:mbox  ?email
    } }
WHERE
  { GRAPH  <http://example/people>
    {
      ?person  foaf:name  ?name .
      OPTIONAL { ?person  foaf:mbox  ?email }
    } }
```

Data before:

```
# Graph: http://example/people
@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> .

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

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

Data after:

```
# Graph: http://example/people
@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> .

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

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

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

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

Example 10:

This example request first copies triples from one named graph to another named graph based on a pattern; triples about all the copied objects that are classified as Physical Objects are then deleted. This demonstrates two operations in a single request, both of which share common PREFIX definitions.

```
PREFIX dc:  <http://purl.org/dc/elements/1.1/>
PREFIX dcmitype: <http://purl.org/dc/dcmitype/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

INSERT
  { GRAPH <http://example/bookStore2> { ?book ?p ?v } }
WHERE
  { GRAPH  <http://example/bookStore>
     { ?book dc:date ?date .
       FILTER ( ?date < "2000-01-01T00:00:00-02:00"^^xsd:dateTime )
       ?book ?p ?v
     }
  } ;

WITH <http://example/bookStore>
DELETE
 { ?book ?p ?v }
WHERE
 { ?book dc:date ?date ;
         dc:type dcmitype:PhysicalObject .
   FILTER ( ?date < "2000-01-01T00:00:00-02:00"^^xsd:dateTime )
   ?book ?p ?v
 }
```

Data before:

```
# Graph: http://example/bookStore
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix dcmitype: <http://purl.org/dc/dcmitype/> .

<http://example/book1> dc:title "Fundamentals of Compiler Design" .
<http://example/book1> dc:date "1996-01-01T00:00:00-02:00"^^xsd:dateTime .
<http://example/book1> a dcmitype:PhysicalObject .

<http://example/book3> dc:title "SPARQL 1.1 Tutorial" .
```

```
# Graph: http://example/bookStore2
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://example/book4> dc:title "SPARQL 1.0 Tutorial" .
```

Data after:

```
# Graph: http://example/bookStore
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix dcmitype: <http://purl.org/dc/dcmitype/> .

<http://example/book3> dc:title "SPARQL 1.1 Tutorial" .
```

```
# Graph: http://example/bookStore2
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix dcmitype: <http://purl.org/dc/dcmitype/> .

<http://example/book1> dc:title "Fundamentals of Compiler Design" .
<http://example/book1> dc:date "1996-01-01T00:00:00-02:00"^^xsd:dateTime .
<http://example/book1> a dcmitype:PhysicalObject .

<http://example/book4> dc:title "SPARQL 1.0 Tutorial" .
```

3.1.3.3 DELETE WHERE

```
DELETE WHERE  QuadPattern
```

The DELETE WHERE operation is a shortcut form for the DELETE/INSERT operation where bindings matched by the WHERE clause are used to define the triples in a graph that will be deleted. Analogous to DELETE/INSERT, deleting triples that are not present, or from a graph that is not present will have no effect and will result in success.

The *QuadPattern* is used both as a pattern for matching against triples and graphs, and as the template for deletion. If any *TripleTemplate*s within the *QuadPattern* appear in the scope of a GRAPH clause then this will determine the graph that that template is matched on, and also the graph from which any matching triples will be removed. Any *TripleTemplate*s not in the scope of a GRAPH clause will be matched against/removed from the default graph.

Example 11:

This example request removes all statements about anything with a given name of "Fred" from the default graph:

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

DELETE WHERE { ?person foaf:givenName 'Fred';
                       ?property      ?value }
```

Data before:

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

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .
<http://example/william> foaf:mbox  <mailto:bill@example> .

<http://example/fred> a foaf:Person .
<http://example/fred> foaf:givenName "Fred" .
<http://example/fred> foaf:mbox  <mailto:fred@example> .
```

Data after:

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

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .
<http://example/william> foaf:mbox  <mailto:bill@example> .
```

Example 12:

This example request removes both statements naming some resource "Fred" in the graph `http://example.com/names`, and also statements about that resource from the graph `http://example/addresses` (assuming that some of the resources in the graph `http://example.com/names` have corresponding triples in the graph `http://example/addresses`).

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

DELETE WHERE {
  GRAPH <http://example.com/names> {
    ?person foaf:givenName 'Fred' ;
            ?property1 ?value1
  }
  GRAPH <http://example.com/addresses> {
    ?person ?property2 ?value2
  }
}
```

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

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .

<http://example/fred> a foaf:Person .
<http://example/fred> foaf:givenName "Fred" .
```

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

<http://example/william> foaf:mbox  <mailto:bill@example> .

<http://example/fred> foaf:mbox  <mailto:fred@example> .
```

Data after:

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

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .
```

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

<http://example/william> foaf:mbox  <mailto:bill@example> .
```

3.1.4 LOAD

The LOAD operation reads an RDF document from a IRI and inserts its triples into the specified graph in the Graph Store. The specified destination graph SHOULD be created if required; again, implementations providing an update service over a fixed set of graphs MUST return with failure for a request that would create a disallowed graph. If the destination graph already exists, then no data in that graph will be removed.

```
LOAD ( SILENT )? IRIref_from ( INTO GRAPH IRIref_to )?
```

IRIref_from specifies the IRI of a document such that a store will be able to identify, locate and read the document. The most common form will be URLs with the *http* IRI schemes. Once the document has been read, the resulting triples will be inserted into the destination graph named by the IRI referred to by *IRIref_to*.

If no destination graph IRI (*IRIref_to*) is provided to load the triples into, then the data will be loaded into the default graph.

In case no RDF data can be retrieved (as opposed to the empty graph being retrieved) from the IRI denoted by *IRIref_from*, or in case the retrieval method returns an error (such as, for instance an HTTP error code), the SPARQL 1.1 Update service SHOULD return failure and the status of the Graph Store SHOULD remain in the same status as prior to the request; in case the keyword SILENT is present, however, the operation will still return success and the status of the Graph Store is not specified by the present document: implementations may create the destination graph or not and partially load data, in case of a transmission error where partial data has been received (which itself may be legal RDF).

3.1.5 CLEAR

The CLEAR operation removes all the triples in the specified graph(s) in the Graph Store.

```
CLEAR  ( SILENT )? (GRAPH IRIref | DEFAULT | NAMED | ALL )
```

Here, the DEFAULT keyword is used to remove all triples in the default graph of the Graph Store, the NAMED keyword is used to remove all triples in all named graphs of the Graph Store and the ALL keyword is used to remove all triples in all graphs of the Graph Store. The GRAPH keyword is used to remove all triples from a graph denoted by *IRIref*. This operation is not required to remove the empty graphs from the Graph Store, but an implementation MAY decide to do so.

```
# Remove all triples from a specified graph.
CLEAR GRAPH IRIref
```

in principle has the same effect as:

```
# Remove all triples from the graph named with the IRI denoted by IRIref.
DELETE { GRAPH IRIref { ?s ?p ?o } } WHERE { GRAPH IRIref { ?s ?p ?o } }
```

Note:

For services which form the default graph from the union of other graphs, CLEAR DEFAULT may have further implications which we leave unspecified here.

If the store records the existence of empty graphs, then the SPARQL 1.1 Update service, by default, SHOULD return *failure* if the specified graph does not exist. If SILENT is present, the result of the operation will always be success.

Stores that do not record empty graphs will always return *success*.

3.2 Graph Management

Graph management operations allow creating, destroying, moving and copying named graphs in the Graph Store, or adding the contents of one graph to another. Operations for creation and destruction are not required to result in any actions, since Graph Stores are not required to record the existence of empty named graphs.

The default graph in a Graph Store always exists.

SPARQL 1.1 Update provides these graph management operations:

- The CREATE operation creates a new graph in stores that support empty graphs.
- The DROP operation removes a graph and all of its contents.
- The COPY operation modifies a graph to contain a copy of another.
- The MOVE operation moves all of the data from one graph into another.
- The ADD operation reproduces all data from one graph into another.

3.2.1 CREATE

This operation creates a graph in the Graph Store:

```
CREATE ( SILENT )? GRAPH IRIref
```

For stores that record empty graphs, this will create a new empty graph in the store with a name specified by the IRI. If the graph already exists, then a *failure* SHOULD be returned, except when the SILENT keyword is used; in either case, the contents of already existing graphs remain unchanged. If the graph may not be created, then a *failure* MUST be returned, except when the SILENT keyword is used.

Stores that do not record empty named graphs will always return *success* on creation of a non-existing graph.

3.2.2 DROP

```
DROP  ( SILENT )? (GRAPH IRIref | DEFAULT | NAMED | ALL )
```

The DROP operation removes the specified graph(s) from the Graph Store. The GRAPH keyword is used to remove a graph denoted by *IRIref*, the DEFAULT keyword is used to remove the default graph from the Graph Store, the NAMED keyword is used to remove all named graphs from the Graph Store, and the ALL keyword is used to remove all graphs from the Graph Store, i.e., resetting the store. After successful completion of this operation, the specified graphs are no longer available for further graph update operations. However, in case the DEFAULT graph of the Graph Store is dropped, implementations MUST restore it after it was removed, i.e., DROP DEFAULT is equivalent to CLEAR DEFAULT.

If the store records the existence of empty graphs, then the SPARQL 1.1 Update service, by default, SHOULD return *failure* if the specified named graph does not exist. If SILENT is present, the result of the operation will always be success.

Stores that do not record empty graphs will always return *success*.

3.2.3 COPY

The COPY operation is a shortcut for inserting all data from an input graph into a destination graph. Data from the input graph is not affected, but data from the destination graph, if any, is removed before insertion.

```
COPY ( SILENT )? ( ( GRAPH )? IRIref_from | DEFAULT) TO ( ( GRAPH )? IRIref_to | DEFAULT )
```

is similar in operation to:

```
DROP SILENT (GRAPH IRIref_to | DEFAULT);
      INSERT { ( GRAPH IRIref_to )? { ?s ?p ?o } } WHERE { ( GRAPH IRIref_from )? { ?s ?p ?o } }
```

The difference between COPY and the DROP/INSERT combination is that if COPY is used to copy a graph onto itself then no operation will be performed and the data will be left as it was. Using DROP/INSERT in this situation would result in an empty graph.

If the destination graph does not exist, it will be created. By default, the service MAY return *failure* if the input graph does not exist. If SILENT is present, the result of the operation will always be success.

Example 13:

This example request copies all statements from the default graph to a named graph:

```
COPY DEFAULT TO <http://example.org/named>
```

Data before:

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

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .
<http://example/william> foaf:mbox  <mailto:bill@example> .
```

```
# Graph http://example.org/named
<http://example/fred> a foaf:Person .
<http://example/fred> foaf:givenName "Fred" .
```

Data after:

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

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .
<http://example/william> foaf:mbox  <mailto:bill@example> .
```

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

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .
<http://example/william> foaf:mbox  <mailto:bill@example> .
```

Note that the original content in `http://example.org/named` is lost by a COPY operation.

3.2.4 MOVE

The MOVE operation is a shortcut for moving all data from an input graph into a destination graph. The input graph is removed after insertion and data from the destination graph, if any, is removed before insertion.

```
MOVE (SILENT)? ( ( GRAPH )? IRIref_from | DEFAULT) TO ( ( GRAPH )? IRIref_to | DEFAULT)
```

is similar in operation to:

```
DROP SILENT (GRAPH IRIref_to | DEFAULT);
      INSERT { ( GRAPH IRIref_to )? { ?s ?p ?o } } WHERE { ( GRAPH IRIref_from )? { ?s ?p ?o } };
DROP ( GRAPH IRIref_from | DEFAULT)
```

The difference between MOVE and the DROP/INSERT/DROP combination is that if MOVE is used to move a graph onto itself then no operation will be performed and the data will be left as it was. Using DROP/INSERT/DROP in this situation would result in the graph being removed.

If the destination graph does not exist, it will be created. By default, the service MAY return *failure* if the input graph does not exist. If SILENT is present, the result of the operation will always be success.

Example 14:

This example request moves all statements from the default graph into a named graph:

```
MOVE DEFAULT TO <http://example.org/named>
```

Data before:

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

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .
<http://example/william> foaf:mbox  <mailto:bill@example> .
```

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

<http://example/fred> a foaf:Person .
<http://example/fred> foaf:givenName "Fred" .
```

Data after:

```
# Default graph
```

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

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .
<http://example/william> foaf:mbox  <mailto:bill@example> .
```

Note that the original content in `http://example.org/named` is lost by a MOVE operation.

3.2.5 ADD

The ADD operation is a shortcut for inserting all data from an input graph into a destination graph. Data from the input graph is not affected, and initial data from the destination graph, if any, is kept intact.

```
ADD ( SILENT )? ( ( GRAPH )? IRIref_from | DEFAULT) TO ( ( GRAPH )? IRIref_to | DEFAULT)
```

is equivalent to:

```
INSERT { ( GRAPH IRIref_to )? { ?s ?p ?o } } WHERE { ( GRAPH IRIref_from )? { ?s ?p ?o } }
```

If the destination graph does not exist, it will be created. By default, the service MAY return *failure* if the input graph does not exist. If SILENT is present, the result of the operation will always be success.

Example 15:

This example request adds all statements from the default graph to a named graph:

```
ADD DEFAULT TO <http://example.org/named>
```

Data before:

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

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .
<http://example/william> foaf:mbox  <mailto:bill@example> .
```

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

<http://example/fred> a foaf:Person .
```

Data after:

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

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .
<http://example/william> foaf:mbox  <mailto:bill@example> .
```

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

<http://example/fred> a foaf:Person .

<http://example/william> a foaf:Person .
<http://example/william> foaf:givenName "William" .
<http://example/william> foaf:mbox  <mailto:bill@example> .
```

4 SPARQL Update Formal Model

This section formally defines the semantics of Update Operations by describing their effects in terms of transformations of the Graph Store.

4.1 General Definitions

4.1.1 Graph Store

Definition: Graph Store

A Graph Store GS is a mutable container of RDF graphs. It has one unnamed (default) slot and zero or more named slots. The unnamed slot holds an RDF graph; each named slot is a pair of a graph and an associated IRI. The Graph Store can be viewed as a mutable RDF Dataset.

$GS = \{DG, (iri_1, G_1), \ldots , (iri_n, G_n) \}$

where

- the default graph DG is the RDF graph associated with the unnamed slot
- $n \geq 0$ and for each $1 \leq i \leq n$, G_i is an RDF graph associated with the named slot identified by IRI iri_i
- all IRIs are distinct, i.e., $i \neq j$ implies $iri_i \neq iri_j$

Note:

We will use GS for the Graph Store, but sometimes also - synonymously - for the RDF Dataset corresponding to the current Graph Store content in subsequent definitions. For convenience, we will also sometimes write $GS = \{DG\}$ union $\{(iri_i, G_i) \mid 1 \leq i \leq n\}$ as an alternative mathematical notation for $GS = \{DG, (iri_1, G_1), \ldots , (iri_n, G_n) \}$ in subsequent definitions.

4.1.2 Abstract Update Operation

Definition: Update Operation

An Update Operation Op is an atomic operation that accepts some arguments Args and transforms a Graph Store GS to another Graph Store GS', denoted as

Op(GS, Args) = GS'

By 'atomic operation' we mean that the operation performs the described transformation of the Graph Store either completely or leaves the Graph Store unchanged, i.e., the result is either GS' or GS (in case of error).

An Update Operation can create new slots and new RDF graphs, or can remove existing slots and the corresponding graphs. It can also alter the state of each slot individually.

We will define the semantics of each concrete update operation in terms of concrete instances of this abstract update operation definition.

4.2 Auxiliary Definitions

In the following we present auxiliary functions and basic operations for creating the union, and difference of RDF Datasets. The concrete update operations will be defined in terms of those basic operations.

Note:

In the following definitions, we write 'union', 'intersect' and 'minus' to denote the respective set operations (union, intersection, and set difference).

4.2.1 Dataset-UNION

This basic operation creates the union of two RDF Datasets.

Definition: Dataset-UNION

Let DS={DG} union {(iri_i, G_i) | 1 ≤ i ≤ n} and DS' = {DG'} union {(iri'_j, G'_j) | 1 ≤ j ≤ m} be two RDF Datasets. Let further graphNames(DS) = { iri_i | 1 ≤ i ≤ n} and graphNames(DS') = {iri'_j | 1 ≤ j ≤ m}. The Dataset-UNION between DS and DS' is defined as follows:

Dataset-UNION(DS, DS') = {DG union DG'} union {(iri, G) | iri in graphNames(DS) union graphNames(DS')}

and G defined as

- G_i for iri = iri_i such that iri_i in graphNames(DS) minus graphNames(DS')
- G_j for iri = iri'_j such that iri_j in graphNames(DS') minus graphNames(DS)
- G_i union G_j for iri = iri_i = iri'_j in graphNames(DS) intersect graphNames(DS')

where union between graphs is defined as set-union of triples in those graphs.

Note:

Note that, in the following, whenever we write Dataset-UNION(X) where X = {DS_1,DS_2,... ,DS_n} is a set of datasets, we understand this as a shorthand for Dataset-UNION(DS_1, Dataset-UNION(DS_2, ... , Dataset-UNION(DS_n,{})...)).

4.2.2 Dataset-DIFF

This operation removes the triples of a given dataset from another dataset.

Definition: Dataset-DIFF

Let DS={DG} union {(iri_i, G_i) | 1 ≤ i ≤ n} and DS' = {DG'} union {(iri'_j, G'_j) | 1 ≤ j ≤ m}) be two RDF Datasets. Let further graphNames(DS) = { iri_i | 1 ≤ i ≤ n} and graphNames(DS') = {iri'_j | 1 ≤ j ≤ m}. The Dataset-DIFF between DS and DS' is defined as follows:

Dataset-DIFF(DS, DS') = {DG minus DG'} union { (iri, G) | iri in graphNames(DS) })

and G defined as

- G_i for iri = iri_i such that iri_i in graphNames(DS) minus graphNames(DS')
- G_i minus G'_j for iri = iri_i = iri'_j in graphNames(DS) intersect graphNames(DS')

where G_i minus G'_j is defined as set-difference over the sets of triples in the two graphs.

4.2.3 Dataset(`QuadPattern`, μ, DS, GS)

The following auxiliary function constructs an RDF Dataset from a `QuadPattern`, given a solution mapping and an RDF Dataset.

Let μ be a solution mapping, DS={DG} union {(iri_i, G_i) | 1 ≤ i ≤ n} be an RDF Dataset and GS be the current state of the Graph Store. DS is distinguished from GS as they may differ, for instance, due to the use of `USING [NAMED]` to modify DS.

For a *QuadPattern* of the form

- `'{}'`

  Dataset(*QuadPattern*, μ, DS, GS) = {{}} i.e., the empty dataset consisting only of an empty default graph.

- `'{'` `TriplesTemplate`? `'}'`

  Dataset(*QuadPattern*, μ, DS, GS) is the Dataset consisting of only a default graph composed by all valid RDF triples obtained from substituting the variables in $sk_μ$(`TriplesTemplate`) according to μ and combining these triples into a single RDF graph by set union.

- `'GRAPH'` `VarOrIri` `'{'` `TriplesTemplate`? `'}'`

  Dataset(*QuadPattern*, μ, DS, GS) is the Dataset consisting of an empty default graph, plus - in case μ(`VarOrIri`) yields a valid IRI - a named graph (μ(`VarOrIri`), *G*) such that *G* is composed by all valid RDF triples obtained from substituting the variables in $sk_μ$(`TriplesTemplate`) according to μ and combining these triples into a single RDF graph by set union.

- `'{'` *QuadPattern1 QuadPattern2* `'}'`

  Dataset(*QuadPattern*, μ , DS, GS) = Dataset-UNION (Dataset(*QuadPattern1*, μ, DS, GS) , Dataset(*QuadPattern2*, μ, DS, GS))

Here, $sk_μ$(`TriplesTemplate`) stands for replacing any blank nodes occurring in the `TriplesTemplate` with a new, unique blank node (unique to the current update request and to each μ and different from any blank nodes used in DS or in GS).

The function sk_μ guarantees that "fresh" blank nodes in the QuadPattern are re-instantiated "per solution" μ (analogous to the treatment of blank nodes in CONSTRUCT templates in the SPARQL1.1 Query Language); cf. also the respective remarks on scoping of blank nodes within requests in the SPARQL grammar.

4.2.4 Dataset(QuadPattern, *P*, DS, GS)

The following auxiliary function constructs an RDF Dataset from a QuadPattern, given a graph pattern and an RDF Dataset.

> Let *P* be a Graph Pattern and DS={DG} union {(iri_i, G_i) | 1 ≤ i ≤ n} be an RDF Dataset and GS be the current state of the Graph Store. Then
>
> Dataset(*QuadPattern*, *P*, DS, GS) = Dataset-UNION({ Dataset(*QuadPattern*, μ, DS, GS) | μ in *eval'*(DS(DG),*P*) })
>
> i.e., the union over all μ such that μ is in the solutions of *P* over dataset DS.
>
> Here, *eval'*() is defined exactly like the evaluation function eval() in the SPARQL1.1 Query Language, with the only exception, that - as opposed to the treatment of blank nodes in BGP matching for SPARQL1.1 Query - here the *scoping graph* SG used for BGP matching is equal to the active graph, i.e., blank nodes from the active graph are preserved in solutions.

The definition of eval'() guarantees that co-referent blank nodes in DS are not "lost" during pattern evaluation, cf. Treatment of Blank Nodes in SPARQL1.1 Query. The latter is necessary to ensure that blank nodes in DS can be matched against existing blank nodes in GS to remove/add triples. In order to illustrate matching against existing blank nodes in the Graph Store, the following update request removes all triples with blank node as subject.

```
DELETE { ?S ?P ?O . } WHERE { ?S ?P ?O . FILTER ( isBlank(?S)) }
```

Data before:

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

_:b a foaf:Person .
:s  a foaf:Person .
```

Data after:

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

:s  a foaf:Person .
```

4.3 Graph Update Operations

4.3.1 Insert Data Operation

> **Definition: Insert Data Operation**
>
> A Insert Data Operation is an Update Operation in which new triples, given as a (ground) *QuadPattern*, are added in the Graph Store GS, in the default slot or in named slots.
>
> OpInsertData(GS, *QuadPattern*) = Dataset-UNION(GS, Dataset(*QuadPattern*,{},GS,GS))
>
> where {} is the empty solution mapping.

4.3.2 Delete Data Operation

> **Definition: Delete Data Operation**
>
> A Delete Data Operation OpDeleteData is an Update Operation in which triples, given as a (ground) *QuadPattern*, are removed from the Graph Store GS, from the default slot or from named slots.
>
> OpDeleteData(GS, *QuadPattern*) = Dataset-DIFF(GS, Dataset(*QuadPattern*,{},GS,GS))
>
> where {} is the empty solution mapping.

4.3.3 Delete Insert Operation

Definition: Delete Insert Operation

A Delete Insert Operation OpDeleteInsert is an Update Operation in which (1) triples are deleted from the Graph Store GS, either from the default slot or from named slot(s), and then (2) new triples are added in the Graph Store GS, either in the default slot or in named slot(s). Triples to be removed (and inserted, respectively) are identified by applying the pattern solutions for a Group Graph Pattern *P* against DS to the `QuadPattern` $QuadPattern_{DEL}$ (and $QuadPattern_{INS}$, respectively).

OpDeleteInsert(GS, DS, $QuadPattern_{DEL}$, $QuadPattern_{INS}$, *P*) = Dataset-UNION(Dataset-DIFF(GS, Dataset($QuadPattern_{DEL}$,*P*,DS,GS)), Dataset($QuadPattern_{INS}$, *P*,DS,GS)

4.3.4 Load Operation

Definition: Load Operation

A Load Operation OpLoad is an Update Operation in which new triples (from a remote graph) are added in the Graph Store, either in the default slot or in a named slot, if specified.

OpLoad(GS, *documentIRI*) = Dataset-UNION(GS, { graph(*documentIRI*) })

OpLoad(GS, *documentIRI*, *iri*) = Dataset-UNION(GS, { {}, (*iri*,graph(*documentIRI*)) })

where graph(*documentIRI*) is a function returning the RDF graph serialized by the RDF document retrieved from IRI *documentIRI*, where blank nodes present in the retrieved graph are supposed to be "standardized apart"; i.e., blank nodes from a loaded graph need to be disjoint with the blank nodes already present in the Graph Store GS.

4.3.5 Clear Operation

Definition: Clear Operation

A Clear Operation OpClear is an Update Operation in which triples are deleted from the Graph Store, either from a named slot, the default slot, all named slots or all slots. There are different variants of the Clear Operation, OpClear for clearing a named graph, $OpClear_{def}$ for clearing the default graph, $OpClear_{named}$ for clearing all named graphs, and $OpClear_{all}$ for clearing all graphs including the default graph.

Let GS = {DG} union $\{(iri_i, G_i) \mid 1 \le i \le n\}$ and graphNames(GS) = $\{ iri_i \mid 1 \le i \le n\}$, then

OpClear(GS, iri) = GS if iri not in graphNames(GS); otherwise, OpClear(GS, iri_j) = (GS minus $\{(iri_j, G_j)\}$) union $\{(iri_j,\{})\}$, where $(iri_j, G_j) \in$ GS and iri = iri_j

$OpClear_{def}$(GS) = {{}} union $\{(iri_i, G_i) \mid 1 \le i \le n\}$

$OpClear_{named}$(GS) = {DG} union $\{(iri_i, \{}) \mid 1 \le i \le n\}$

$OpClear_{all}$(GS) = {{}} union $\{(iri_i, \{}) \mid 1 \le i \le n\}$

Note:

Since Graph Stores may remove graphs that are left empty, for such Graph Stores any Clear Operation performed on a named graph may be viewed as immediately followed by a Drop Operation, see below.

4.4 Graph Management Operations

4.4.1 Create Operation

Definition: Create Operation

A Create Operation OpCreate is an Update Operation in which (1) a new named slot and (2) a new graph G are created in the Graph Store. The new graph is held in the new slot, and is empty. Other slots and graphs are not affected.

Let GS = {DG} union $\{(iri_i, G_i) \mid 1 \le i \le n\}$ and graphNames(GS) = $\{ iri_i \mid 1 \le i \le n\}$, then

OpCreate(GS, iri) = GS union {(iri, {})} if iri not in graphNames(GS); otherwise, OpCreate(GS, iri) = GS

Note:

Since Graph Stores may remove graphs that are left empty, for such Graph Stores any Create Operation performed on an empty or non-existent graph may be viewed as implicitly immediately followed by a Drop Operation (see next subsection), or simply as an operation with no effect.

4.4.2 Drop Operation

Definition: Drop Operation

A Drop Operation OpDrop is an Update Operation in which one or more slots (a named slot iri_i, the default slot, all named slots or all slots) and their corresponding graphs are removed from the Graph Store. There are different variants of the Drop Operation, OpDrop for dropping a named graph, $OpDrop_{def}$ for dropping the default graph (which is equivalent to $OpClear_{def}$, since the default graph cannot be removed, but dropping it means only to clear it), $OpDrop_{named}$ for dropping all named graphs, and $OpDrop_{all}$ for dropping all graphs including the default graph.

Let GS = {DG} union {(iri_i, G_i) | $1 \leq i \leq n$} and graphNames(GS) = { iri_i | $1 \leq i \leq n$}, then

OpDrop(GS, iri) = GS if iri not in graphNames(GS); otherwise, OpDrop(GS, iri_j) = {DG} union {(iri_i, G_i) | $i \neq j$ and $1 \leq i \leq n$} where iri = iri_j

$OpDrop_{def}$(GS) = $OpClear_{def}$(GS)

$OpDrop_{named}$(GS) = {DG}

$OpDrop_{all}$(GS) = {{}}

4.5 Mapping Update Requests to the Formal Model

In this section we show how to map Update Requests in the SPARQL 1.1. Update Language to Update Operations over the Graph Store as defined earlier in this section. This mapping assumes that in all Update requests, any `PREFIX`es have been expanded. Moreover, we assume that `WITH` clauses have been replaced by wrapping both the *`QuadPattern`*s in subsequent `DELETE` and `INSERT` clauses, and likewise - in the absence of `USING` and `USING NAMED` clauses - the *`GroupGraphPattern`* in the subsequent `WHERE` clause, into `GRAPH` patterns.

The mapping from requests to Update Operations is defined in terms of the recursive translation function *Tr*(GS,*R*) which takes the Graphstore GS - as before executing the request - and an update request *R* as input and exands it to an Update Operation call as shown in the following table. The `COPY`, `MOVE`, and `ADD` operations are not mentioned explicitly here, since they are understood as shortcuts.

Table 1: Mapping from Update Requests to Update Operations

Update request *R*	***Tr*(GS,*R*) =**
R_1 ; R_2	*Tr*(*Tr*(GS, R_1), R_2)
`INSERT DATA` *QuadData*	OpInsertData(GS, *QuadData*)
`DELETE DATA` *QuadData*	OpDeleteData(GS, *QuadData*)
`DELETE` $QuadPattern_{DEL}$ `INSERT` $QuadPattern_{INS}$ *UsingClause** `WHERE` *GroupGraphPattern*	OpDeleteInsert(GS, $Tr_{Dataset}$(GS,*UsingClause**), $QuadPattern_{DEL}$, $QuadPattern_{INS}$, *GroupGraphPattern*)
`DELETE` $QuadPattern_{DEL}$ *UsingClause** `WHERE` *GroupGraphPattern*	OpDeleteInsert(GS, $Tr_{Dataset}$(GS,*UsingClause**), $QuadPattern_{DEL}$, {}, *GroupGraphPattern*)
`INSERT` $QuadPattern_{INS}$ *UsingClause** `WHERE` *GroupGraphPattern*	OpDeleteInsert(GS, $Tr_{Dataset}$(GS,*UsingClause**), {}, $QuadPattern_{INS}$, *GroupGraphPattern*)
`DELETE WHERE` *QuadPattern*	OpDeleteInsert(GS, GS, *QuadPattern*, {}, *QuadPattern*)
`LOAD (SILENT)?` *IRIref*	OpLoad(GS, *IRIref*)
`LOAD (SILENT)?` $IRIref_{from}$ `INTO GRAPH` $IRIref_{to}$	OpLoad(GS, $IRIref_{from}$, $IRIref_{to}$)
`CLEAR (SILENT)? GRAPH` *IRIref*	OpClear(GS, *IRIref*)
`CLEAR (SILENT)? DEFAULT`	$OpClear_{def}$(GS)
`CLEAR (SILENT)? NAMED`	$OpClear_{named}$(GS)
`CLEAR (SILENT)? ALL`	$OpClear_{all}$(GS)
`CREATE (SILENT)? GRAPH` *IRIref*	OpCreate(GS, *IRIref*)
`DROP (SILENT)? GRAPH` *IRIref*	OpDrop(GS, *IRIref*)
`DROP (SILENT)? DEFAULT`	$OpDrop_{def}$(GS)
`DROP (SILENT)? NAMED`	$OpDrop_{named}$(GS)
`DROP (SILENT)? ALL`	$OpDrop_{all}$(GS)

This table uses one auxiliary translation function $Tr_{Dataset}$() which constructs a dataset from the optional set of `USING` and `USING NAMED` clauses and is defined as follows:

Table 2: Mapping *UsingClause*s to RDF Datasets

Translation Function	**Definition**

$Tr_{Dataset}$(GS,*UsingClause**) =	• the RDF Dataset DS described by the *UsingClause*s, if non-empty • the RDF Dataset corresponding to the current state of GS, otherwise

Note:

How exactly an RDF Dataset is obtained from the `USING` *and* `USING NAMED` *clauses (e.g. by dereferencing graph name IRIs and trying to retrieve them, or by picking those graphs from the existing Graph Store) is implementation dependent. Particularly, this specification does not mandate any assumptions about blank node identity beyond the consideration for the analogous* `FROM` *and* `FROM NAMED` *clauses in Section Specifying RDF Datasets of the SPARQL 1.1 Query Language specification.*

5 Conformance

See appendix B SPARQL 1.1 Update Grammar regarding conformance of SPARQL Update strings.

This specification is intended for use in conjunction with: the SPARQL 1.1 Graph Store HTTP Protocol and the SPARQL 1.1 Protocol for RDF.

A Security Considerations (Informative)

Exposing RDF data for update creates many security issues which all deployments must be aware of, and consider the risks involved. This submission discusses some of the potential issues. New security problems are discovered regularly, and each implementation introduces its own concerns. Consequently implementers should be aware that this is only a partial list containing possible issues, and cannot be considered complete nor authoritative.

- Write access to data makes it inherently vulnerable to malicious access. Standard access and authentication techniques should be used in any networked environment. In particular, HTTPS should be used, especially when implementing the SPARQL HTTP-based protocols. (i.e., encryption with challenge/response based password presentation, encrypted session tokens, etc). Some of the weak points addressed by HTTPS are: authentication, active session integrity between client and server, preventing replays, preventing continuation of defunct sessions.
- SPARQL Update incurs all of the security concerns of SPARQL Query. In particular, stores which treat IRIs as dereferenceable need to protect against dereferenced IRIs from being used to invoke cross-site scripting attacks.
- Implementations will need to enforce their standard permissions scheme carefully. Permissions schemes always require careful design, and it is important to ensure that privileges in one area are not inadvertently applied to other parts of the system.
- Systems that provide both read-only and writable interfaces can be subject to injection attacks in the read-only interface. In particular, a SPARQL endpoint with a Query service should be careful of injection attacks aimed at interacting with an Update service on the same SPARQL endpoint. Like any client code, interaction between the query service and the update service should ensure correct escaping of strings provided by the user.
- While SPARQL Update and SPARQL Query are separate languages, some implementations may choose to offer both at the same SPARQL endpoint. In this case, it is important to consider that an Update operation may be obscured to masquerade as a query. For instance, a string of unicode escapes in a PREFIX clause could be used to hide an Update Operation. Therefore, simple syntactic tests are inadequate to determine if a string describes a query or an update.

B Internet Media Type, File Extension and Macintosh File Type

The Internet Media Type / MIME Type for the SPARQL Update Language is "`application/sparql-update`".

It is recommended that SPARQL Update files have the extension ".ru" (lowercase) on all platforms.

It is recommended that SPARQL Update files stored on Macintosh HFS file systems be given a file type of "TEXT".

Type name:
application
Subtype name:
sparql-update
Required parameters:
None
Optional parameters:
None
Encoding considerations:
The syntax of the SPARQL Update 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 Update appendix A, 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):
".ru"

Base IRI:
The SPARQL 'BASE <IRIref>' term can change the current base IRI 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.

C SPARQL 1.1 Update Grammar

The formal definition for the SPARQL 1.1 Update grammar is provided with the SPARQL 1.1 Query grammar. This is because the grammar for SPARQL 1.1 Update shares most of its structure with SPARQL 1.1 Query.

D References

D.1 Normative References

IANA-CHARSETS
(Internet Assigned Numbers Authority) *Official Names for Character Sets*, ed. Keld Simonsen et al. (See http://www.iana.org/assignments/character-sets.)
RFC3987
Internationalized Resource Identifiers (IRIs), M. Dürst , M. Suignard (See http://www.ietf.org/rfc/rfc3987.txt.)
RDF-MT
RDF Semantics, P. Hayes, Editor, W3C Recommendation, 10 February 2004, see http://www.w3.org/TR/2004/REC-rdf-mt-20040210/, Latest version available at http://www.w3.org/TR/rdf-mt/. (See http://www.w3.org/TR/2004/REC-rdf-mt-20040210/.)

D.2 Other References

Aho/Ullman
Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Reading: Addison-Wesley, 1986, rpt. corr. 1988.
Brüggemann-Klein
Brüggemann-Klein, Anne. *Formal Models in Document Processing*. Habilitationsschrift. Faculty of Mathematics at the University of Freiburg, 1993. (See ftp://ftp.informatik.uni-freiburg.de/documents/papers/brueggem/habil.ps.)
Brüggemann-Klein and Wood
Brüggemann-Klein, Anne, and Derick Wood. *Deterministic Regular Languages*. Universität Freiburg, Institut für Informatik, Bericht 38, Oktober 1991. Extended abstract in A. Finkel, M. Jantzen, Hrsg., STACS 1992, S. 173-184. Springer-Verlag, Berlin 1992. Lecture Notes in Computer Science 577. Full version titled *One-Unambiguous Regular Languages* in Information and Computation 140 (2): 229-253, February 1998.
Clark
James Clark. *Comparison of SGML and XML*. (See http://www.w3.org/TR/NOTE-sgml-xml-971215.)
IANA-LANGCODES
(Internet Assigned Numbers Authority) *Registry of Language Tags* (See http://www.iana.org/assignments/language-subtag-registry.)
IETF RFC 2141
IETF (Internet Engineering Task Force). *RFC 2141: URN Syntax*, ed. R. Moats. 1997. (See http://www.ietf.org/rfc/rfc2141.txt.)
IETF RFC 3023
IETF (Internet Engineering Task Force). *RFC 3023: XML Media Types*. eds. M. Murata, S. St.Laurent, D. Kohn. 2001. (See http://www.ietf.org/rfc/rfc3023.txt.)
IETF RFC 2781
IETF (Internet Engineering Task Force). *RFC 2781: UTF-16, an encoding of ISO 10646*, ed. P. Hoffman, F. Yergeau. 2000. (See http://www.ietf.org/rfc/rfc2781.txt.)
IETF RFC 3629
IETF (Internet Engineering Task Force). *RFC 3629: UTF-8, a transformation format of ISO 10646*, F. Yergeau. November 2003. (See http://www.ietf.org/rfc/rfc3629.txt.)
ISO 639
(International Organization for Standardization). *ISO 639:1988 (E). Code for the representation of names of languages.* [Geneva]: International Organization for Standardization, 1988.
ISO 3166
(International Organization for Standardization). *ISO 3166-1:1997 (E). Codes for the representation of names of countries and their subdivisions — Part 1: Country codes* [Geneva]: International Organization for Standardization, 1997.
ISO 8879
ISO (International Organization for Standardization). *ISO 8879:1986(E). Information processing — Text and Office Systems — Standard Generalized Markup Language (SGML).* First edition — 1986-10-15. [Geneva]: International Organization for Standardization, 1986.
ISO/IEC 10744
ISO (International Organization for Standardization). *ISO/IEC 10744-1992 (E). Information technology — Hypermedia/Time-based Structuring Language (HyTime).* [Geneva]: International Organization for Standardization, 1992. *Extended Facilities Annexe.* [Geneva]: International Organization for Standardization, 1996.

UNICODE
The Unicode Consortium. *The Unicode Standard, Version 5.0.0.* Boston, MA, Addison-Wesley, 2007. ISBN 0-321-48091-0. (See http://www.unicode.org/unicode/standard/versions/.)

WEBSGML
ISO (International Organization for Standardization). *ISO 8879:1986 TC2. Information technology — Document Description and Processing Languages*. [Geneva]: International Organization for Standardization, 1998. (See http://www.sgmlsource.com/8879/n0029.htm.)

XML Names
Tim Bray, Dave Hollander, and Andrew Layman, editors. *Namespaces in XML*. Textuality, Hewlett-Packard, and Microsoft. World Wide Web Consortium, 1999. (See http://www.w3.org/TR/xml-names/.)

Change Log

Changes since Proposed Recommendation

- Fixed a broken fragment link into SPARQL 1.1 Query

Changes since Last Call

- Simplified explaining text as per the ban of shared bnodes across operations in a request.
- Editorial fix to Definition 4.2.3 and explaining remarks.
- Added explanation of QuadData to "Terminology" section.
- Added comment indicating that existing graphs do not lose triples during a LOAD
- Several minor editorial changes including the removal of "(non graph-aware)" since - essentially - every graph store is graph-aware.
- Changed SHOULD to MAY in 3.2.3 Copy
- Various editorial