SPARQL_edit: Editing RDF Literals in Knowledge Graphs via View-Update Translations

Sascha Meckler[1(✉)] and Andreas Harth[1,2]

[1] Fraunhofer IIS, Fraunhofer Institute for Integrated Circuits IIS, Erlangen, Germany
{sascha.meckler,andreas.harth}@iis.fraunhofer.de
[2] University of Erlangen-Nuremberg, Erlangen, Germany

Abstract. In order to improve the adoption of knowledge graphs (KG) in everyday work, non-technical users must be empowered to not only view but to write data in a KG. Whereas most available software tools focus on displaying information, the presented solution helps business users to execute write operations for correcting wrong values or inserting missing values. SPARQL_edit is a Web application that enables users without any knowledge of SPARQL or RDF to view query results and instantly edit RDF literal values of the knowledge graph. The main concept can be summarized as 'editable SPARQL result table'. If a user modifies the value of an RDF literal in the view, a generic view-update algorithm translates the change into a SPARQL/Update query that updates the correct RDF triple in the KG. Similar to the view update problem in databases, there are restrictions of the SPARQL language features that can be used for creating a view with unambiguous updates to the graph.

1 Introduction

The application of Semantic Web technologies in enterprises has received increasing attention from both the research and industrial side. Information is often stored in a knowledge graph (KG), "a graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent relations between these entities" [19]. There are openly available knowledge graphs such as DBpedia, Wikidata or Freebase and commercial enterprise-related KGs. Enterprise knowledge graphs (EKG) are designed for the use inside a company with applications from search and recommendations, commerce and finance to social networks [19]. The adoption of KGs for connecting data and knowledge inside an organisation was popularised by the big tech corporations, namely Google, Facebook, Microsoft, eBay and IBM [22]. EKG can be seen as an implementation of linked enterprise data by means of a "semantic network of concepts, properties, individuals and links representing and referencing foundational and domain knowledge relevant for an enterprise" [15]. Despite their different application domains, EKGs have common features: EKGs integrate data from different internal and external heterogeneous data

sources resulting in millions or billions of nodes and edges. Ontologies are typically used to define relationships between concepts and serve as a data schema. Artificial intelligence is used for knowledge extraction from unstructured data or for reasoning, the inference of new facts from existing ones. Every EKG requires an initial refinement and techniques to keep the graph up-to-date with corporate operations to guarantee a high quality knowledge base.

KGs can be built with different graph-structured data models but EKG are typically labelled property graphs or directed edge-labelled graphs built upon the Resource Description Framework (RDF) [9], the standard data model for the Web of Data. This paper focuses on RDF KGs that are queried with SPARQL [17], the query language of the Semantic Web.

The life-cycle of a KG can be divided into the creation, hosting, curation and the deployment phase [28]. In the creation and hosting process the KG is built from semantically annotated data which was generated by mapping data from heterogeneous sources into the graph format. The curation process includes tasks for cleaning and enriching in order to improve the correctness and completeness of the newly created KG. If errors are detected, the mapping rules are modified to correct the graph output. Even after the KG cleaning, there are still faulty values in the graph data that cannot be found by cleaning algorithms or the data engineers. There is still the need for manual changes after the deployment, e.g. for updating values or the correction of wrong values.

To simplify the work with KGs, the effort for maintaining and updating single values in a KG must be reduced. Instead of contacting a data engineer or re-executing the whole creation process, non-expert users should be able to update wrong values immediately by themselves. However, non-technical users are usually not familiar with RDF graph structures and the SPARQL query language. This leads to the overarching research question of how to enable lay users to update single values in an EKG. This question can be divided into more specific sub-questions:

– Is it possible to create an editable view of KG data where changes made by the user are automatically handed back to the original graph?
– Can we automatically translate a change to a SPARQL query result (view) into a SPARQL/Update query? Under what circumstances can an algorithm create unambiguous database updates?
– Which requirements and features are necessary for an end-user application?

This paper presents SPARQL_edit, a demo implementation of a generic solution that allows users to display arbitrary SPARQL query results in a table with editable input fields for RDF literal results. We explain an algorithm that translates the user's changes into a SPARQL/Update query that alters the correct RDF triples in the KG. The view-update algorithm constructs a SPARQL/Update query from the initial SPARQL/Select query, the query results and the changed RDF literal value. Similar to the view update problem in databases, there are restrictions to the SPARQL language features that can be used inside the initial query. This paper evaluates the benefits and shortcomings of this generic approach and discusses restrictions to the SPARQL grammar. Our main

contribution is the practical study of a SPARQL-based view-update application for the use with RDF knowledge graphs.

The following section introduces a practical application example that showcases the functioning of SPARQL_edit. Section 3 describes related read-write Linked Data systems and the relation to the view update problem which is known from relational databases. The view-update algorithm and its restrictions are discussed in Sect. 4. It is followed by the description of the implementation in Sect. 5 and a study of the applicability of the presented solution in terms of performance and collaborative work in Sect. 6.

2   Practical Example

The concept of SPARQL_edit is based on the assumption that users are familiar with working on tabular data in spreadsheets. Graph data from a KG is presented in a table where cells of RDF literal solutions can be edited. A user story for the application in an enterprise environment is the correction of a product catalog in the EKG: While preparing the delivery bill for a customer, Bob notices that the billing software fetched an incorrect weight attribute for a product from the EKG. Bob wants to update the product weight in the EKG but he is not familiar with SPARQL. Bob opens SPARQL_edit in his browser and loads the 'Product Catalog View' configuration that Alice from the IT department shared with him. SPARQL_edit then displays a table with the product data. After filtering the table rows, Bob changes the cell with the wrong weight value and submits the change which is immediately applied to the company's EKG.
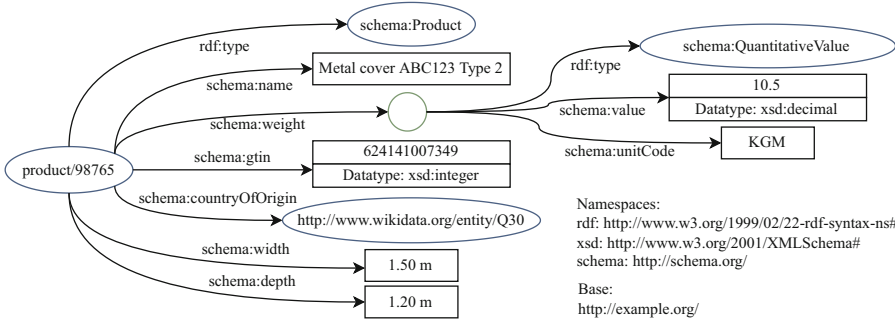
Fig. 1. Product graph from a fictional graph G of a dataset D

Figure 1 shows a graph of the exemplary product whose weight property shall be changed. After loading the 'Product Catalog View' in the SPARQL_edit Web application, Bob filters the displayed table for the desired product with a certain GTIN number. He uses the form controls to change the weight value as shown in Fig. 2. The 'Product Catalog View' has been created with the SPARQL/Select query from Listing 1.1. When Bob changes the weight value, the algorithm

generates the SPARQL/Update query shown in Listing 1.2. If Bob submits his change, the application will send this update query to the graph store and refresh the values in the table by re-executing the original SPARQL/Select query. After the initial loading of the view, the literal update process consists of four steps:

1. Algorithm generates a SPARQL/Update query
2. SPARQL/Update query from step 1 is sent to the SPARQL server
3. Original SPARQL/Select query of the view is rerun
4. User interface is updated with values from step 3.

3   Related Work

Different software tools have been developed to support users who do not know the concepts of RDF and Semantic Web and also expert users in their work with Linked Data and KG. Most tools and techniques focus on the exploration and visualisation of semantically connected data which is either distributed across the Web or stored in a central location. However, there are only few software tools that allow users to execute write operations in a generic and reusable way. Generic read-write systems without explicit mapping rules between the presentation of data and the database struggle with the view update problem. This section presents related read-write Linked Data systems and gives an introduction to the database view update problem. After this, solutions for creating database-to-RDF mappings with write capabilities are listed.

3.1   Read-Write Linked Data Systems

Tabulator. Following Tim Berners-Lee's concept of a read-write Linked Data Web, "it is essential that the Data Web should not be read-only" [2]. Tim Berners-Lee and his colleagues developed Tabulator [3], a Web of Data browser with write capabilities that allows users to modify and extend information within the browser interface. In Tabulator, changes to a resource are submitted back to the Web server using an HTTP and SPARQL-based protocol. Berners-Lee et al. [3] discuss the difficulties that arise with write functionality. These difficulties are related to the view update problem which is described in Sect. 3.2.

RDF-to-Form Mapping. Software tools such as RDFauthor [32] facilitate RDF editing by creating a bidirectional mapping between RDF data and Web HTML forms by making use of semantic markup annotations like RDFa. Rdf Edit eXtension (REX) [30] uses RDFa-like HTML attributes to create a mapping between an RDF graph and the model of the HTML form controls. When a user changes a value in the form controls, REX generates and executes a SPARQL/Update query based on the semantic annotations. Moreover, RDForms [11] is a library for creating form-based RDF editors by using a templating mechanism which uses bidirectional mapping rules to translate between the Web form and the RDF graph. In contrast to RDFauthor, REX and RDForms, SPARQL_edit

Fig. 2. Editing of an RDF literal value in the user interface of SPARQL_edit

```
PREFIX rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#>
PREFIX schema: <http://schema.org/>
SELECT ?product ?name ?gtin ?weight
FROM <http://example.org/dataset/graph>
WHERE {
   ?product rdf:type schema:Product ;
      schema:name ?name ;
      schema:gtin ?gtin ;
      schema:weight [
         schema:value ?weight
      ]
   FILTER (lang(?name) = 'en')
}
```

Listing 1.1. SPARQL/Select query for the product catalog view in Fig. 2

```
PREFIX rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX schema: <http://schema.org/>
WITH <http://example.org/dataset/graph>
DELETE { ?g_0 schema:value "10.5"^^xsd:decimal . }
INSERT { ?g_0 schema:value "8.8"^^xsd:decimal . }
WHERE {
   <http://example.org/product/98765> rdf:type schema:Product;
      schema:name "Metal cover ABC123 Type 2"@en;
      schema:gtin 624141007349;
      schema:weight ?g_0.
   ?g_0 schema:value "10.5"^^xsd:decimal .
}
```

Listing 1.2. SPARQL/Update query generated for changing the weight RDF triple

does not require the upfront definition of mapping rules or form annotations. In SPARQL_edit, an algorithm automatically translates a change to a form control into an appropriate SPARQL/Update query that will manipulate the underlying RDF graph. Instead of embedding information in the HTML form, SPARQL_edit gathers the necessary inputs for the algorithm from the view defined by a SPARQL/Select query and the current database state.

Solid Pod Managers. Today, the read-write Linked Data Web is extended with Social Linked Data (Solid)[1], a new set of technical concepts and standards for building a decentralized social Web. The key concept is the storage of personal data in a user-managed personal online data store (Pod) whose resource management is based on the Linked Data Platform (LDP) [29]. With Solid, a new category of read-write Linked Data tools emerged: Pod managers or browsers. Pod managers can be seen as successors of LDP clients [34] that use Solid authentication and authorization. Pod managers like SolidOS (Databrowser)[2] or Pod-Browser[3] provide a GUI for resource management and triple editing of RDF documents in a Solid Pod. The Solid protocol [8] allows document updates via HTTP PUT requests and via PATCH requests that include Notation 3 (N3) [1] rules defining the update operation. Although the document-based update is different to our query-based approach, the N3 update rules have similarities with the queries in SPARQL_edit. A Solid `InsertDeletePatch` typically includes an insertion, a deletion and a condition formula that is evaluated against the target RDF document similar to a `DELETE-INSERT-WHERE` SPARQL/Update query.

PoolParty GraphEditor. Besides the open-source tools, there are proprietary solutions that are often incorporated into commercial Semantic Web or KG platforms. PoolParty's GraphEditor[4] provides a graphical user interface that guides users in the creation of views for graph editing. The proprietary tool helps users to visually build queries, filter the results and perform inline editing to single or multiple triples of the KG. GraphEditor automatically generates a SPARQL query based on the selected graphs, the chosen properties from the selected vocabulary as well as the user-defined filters and is able to "refine triples, create or change relations among their properties or their names and labels and even create new resources" [27]. Since SPARQL_edit has a similar motivation as PoolParty's GraphEditor – to build an easily usable application for directly editing data in a KG – it shares the same design principle of having customizable views on a KG and follows the same concept of in-line editing by means of editable fields in query results. Whereas GraphEditor is strongly integrated in the proprietary platform of PoolParty, SPARQL_edit is a lightweight stand-alone application that can be used with any SPARQL 1.1 query/update endpoint.

---

[1] cf., https://solidproject.org/.

[2] cf., https://github.com/SolidOS/solidos.

[3] cf., https://github.com/inrupt/pod-browser.

[4] cf., https://www.poolparty.biz/poolparty-grapheditor.

3.2   Database View Update Problem

The (database) view update problem with its origin in the field of relational databases describes the problem of translating updates on a view into equivalent updates on the database. There are cases in which the translation of a view update is not unique, ill-defined or even impossible. Calculated update queries may create inconsistencies in the database or have side-effects on the derived view [14]. In general, there are two opposed approaches for solving the view update problem [14]: In the first approach, the view includes all permissible view updates together with their translations. This approach requires the explicit definition of translation rules on how to update the database for each possible update operation in the view. The second solution are general view update translators which are based on the analysis of the conceptual schema dependencies or on the concept of a view complement in order to create unique view update translations. The database update is calculated from the view definition, the view update, additional information and the current database state. This automatic translation is comfortable because the database engineer must not define the translation rules like in the first approach. However, the general approach only allows a limited range of view updates that can be translated correctly and the database engineer has to check if the calculated translations are right [14]. For answering the question whether a view is updatable, it is necessary to check if the mapping from the database relations to the view is invertible and if a view update has a translation [13].

The research for relational databases and the SQL query language can be transferred to graph databases, RDF and SPARQL. Berners-Lee et al. [3] discuss the problem of mapping changes in a Web page (presentation) to the underlying Web data that may originate from different Web servers in the context of their tool Tabulator. Whereas Tabulator is focused on linked documents on the Web, SPARQL_edit is working on views defined by SPARQL queries. SPARQL_edit belongs to the group of general view update translators that trade expressivity for generality. We accepted the restrictions for the set of possible view updates to create a lightweight tool which does not require much upfront configuration and is therefore more efficient than developing a domain-specific application.

3.3   Database-to-RDF Mappings

Several techniques and software tools have been developed to create a (virtual) RDF graph from a relational database (RDB) by applying specific mapping rules. The use of an ontology-based mediation layer for accessing data from relational databases is often labeled as Ontology-Based Data Access (OBDA). RDB-to-RDF solutions like D2RQ [5], R2RML [31] and OBDA implementations such as Ontop [7] or Mastro [6] provide RDF views on relational databases or expose a virtual SPARQL endpoint for querying. Although most RDB-to-RDF and OBDA systems are limited to read operations, there have been efforts to create a bidirectional read-write access by reversing the RDB-to-RDF mappings. OntoAccess [18] explicitly supports write access by means of translating

SPARQL/Update operations to SQL. Information and integrity constraints for the database schema are included in the mapping rules in order to identify update requests that cannot be executed by the database engine. D2RQ++ and D2RQ/Update are similar extensions to the D2RQ mapping platform which allow the execution of SPARQL/Update statements on the virtual RDF graph. D2RQ++ and D2RQ/Update differ in the way they handle the translation of SPARQL to a number of SQL statements [12]. Unbehauen and Martin [33] presented a solution for read-write access to graphs that are mapped with the R2RML language. Ontology- and source-level updates have also been examined for OBDA systems [10]. Writing RDF/SPARQL updates back into the RDB naturally leads to the view update problem explained in Sect. 3.2. The translation of a SPARQL/Update query into one or more SQL statements is especially complicated because SPARQL/Update can potentially modify the data and schema triples at the same time [33]. Most solutions with write capabilities are limited to basic relation-to-class and attribute-to-property mappings [20].

4 View-Update Algorithm

The result of a SPARQL/Select query Q is a solution sequence Ω, an unordered list of zero or more solutions from graph pattern matching and operations on the initial matching solutions. Each solution (mapping) $\mu \in \Omega$ has a set of bindings of variables to RDF terms $\mu : V \rightarrow T$ [17]. In the JSON serialization of SPARQL results [26], Ω is represented in an array which is the value of the `bindings` key inside the `results` key. Each member ("result binding") representing a solution mapping μ is a JSON object that encodes the bound values, i.e. RDF terms, for each variable v from Q. In the result table, each column represents a variable and each solution produces a row.

4.1 Translation of a Literal Update into a SPARQL/Update Query

The SPARQL_edit algorithm illustrated in Fig. 3 translates the change of a literal value l in the solution μ_{ed} into a `DELETE-INSERT-WHERE` SPARQL/Update query Q^-. The basic idea behind the algorithm is to rebuild the original SPARQL/Select query Q so that it only affects the single RDF triple whose object literal value shall be changed. This is done by replacing every variable $v \in V$ in Q with the URIs or literal values from the solution $\mu_{ed} \in \Omega$ that is represented in the row with the edited value. In order to have solution mappings for every variable, Q is altered to select all variables (`SELECT *`). The triple patterns with replaced variables are then used for building the update query Q^-. In order to output Q^-, the algorithm requires the input of

– the original SPARQL/Select query (Q),
– the solution mapping/result binding (μ_{ed}) for the edited table row
– and the previous and new literal values (l_{old}, l_{new}).
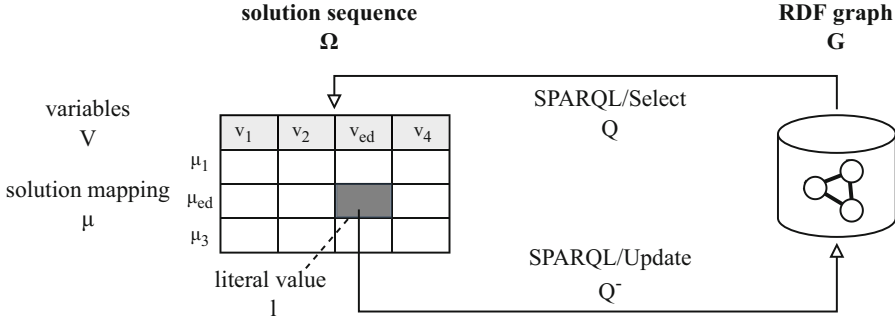
Fig. 3. Principle of the SPARQL_edit algorithm

Algorithm Steps. The algorithm follows four sequential steps for building Q^-:

1. In a first step, information about the edited literal value is collected. This includes the name, datatype, language, old and new values of l.
2. Then, the graph patterns in the WHERE clause of Q are analysed. The algorithm filters and stores Basic Graph Patterns (BGP) and Optional Graph Patterns (OGP). The algorithm finds the triple pattern that includes the particular variable v_{ed} in object position whose solution mapping μ_{ed} was edited. If the triple pattern is part of an OGP, the OGP must be treated like a BGP.
3. In the third step, the WHERE clause for Q^- is built. For each BGP that was collected in the second step, all named variables are replaced with named nodes (URIs) or concrete literal values from μ_{ed}. A variable in subject position is either set to a named node or, in case of an unnamed variable (blank node), set to a named variable. Blank nodes are replaced with named variables so that the subject can be referenced later in the DELETE and INSERT templates. Blank nodes are prohibited in a SPARQL/Update DELETE template, because a new blank node never matches anything in the graph and therefore never results in any deletion [24]. A predicate variable is always replaced with a named node whereas a variable in object position is bound to either a named node, a literal value or a blank node.
4. Next, the DELETE and INSERT templates for Q^- are defined and the update query is assembled. Both templates are built from the same triple pattern which included v_{ed}. In case of the DELETE template, the triple's object holds the old value l_{old} from μ_{ed}. The triple in the INSERT template has the new user-defined literal value l_{new} in the object position. Finally, Q^- is completed by replicating the prefixes of Q. If Q uses a FROM statement to specify the default graph for the query, a matching WITH clause is added to Q^-.

Insert Mode. The algorithm is capable of generating SPARQL/Update queries for updating existing literal values, but also for inserting missing RDF literals. However, the insert mode has two prerequisites: First, the variable v_{ed} must be defined inside an optional graph pattern (OGP) in the WHERE clause of Q.

Otherwise, there would be no table row representing a solution mapping which is missing a literal value for v_{ed}. The second requirement is that Ω includes at least one solution mapping that binds v_{ed} to an RDF literal term $\mu : v_{ed} \rightarrow lit$. This literal result is used as a template for new literals the user wants to insert. The insert mode introduces a new edge case for the algorithm. If the OGP is composed of an n-ary relation [25], the INSERT template has to include all successive triples needed for the construction of a new blank node with the user-defined literal. In practice, the insert mode is useful to fill in gaps in the KG data which may originate from mapping incomplete data sources to RDF.

4.2   Query Restrictions

According to the two classes of solutions for the view update problem [14], the SPARQL_edit algorithm belongs to the general view update translators where the database update is automatically calculated from the view definition (Q), the view update (l_{new}) and the current database state (μ_{ed}). However, this comfortable update generation only allows a limited range of view updates that can be translated correctly. The database engineer who creates the views has to check if the calculated translations are right.

The presented algorithm supports simple SPARQL/Select queries on the default graph specified with or without a single FROM clause. If the original query Q defines a certain graph as the default graph with a FROM clause, a corresponding WITH clause is attached to Q^-. There can only be one WITH clause as it "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" [24]. In the case of multiple FROM clauses, the default graph is the RDF merge of the graphs which is ambiguous for any update translations. Despite that, the algorithm might be extended to support SPARQL/Select queries with named graphs using the FROM NAMED clause. With explicitly named graphs, the changed graph could be tracked and included in a SPARQL/Update query of the form:

DELETE { GRAPH <g1>{ s p o } } INSERT { GRAPH <g1>{ s p o2 } }
    USING NAMED <g1> WHERE { . . . }

The algorithm supports SPARQL/Select queries with a graph pattern that is composed of one or more BGPs and OGPs that may contain triple patterns with variables, blank nodes and filters (FILTER). The query can include filter statements and solution sequence modifier (ORDER, LIMIT or OFFSET). When creating Q^-, the algorithm removes any filters because the replacement of the variables with explicit values makes filters obsolete. Any other query constructs are currently not supported by the algorithm. For example, the algorithm cannot support SPARQL/Select queries with irreversible functions like aggregation (GROUP BY) or bindings (BIND). Table 1 specifies the restrictions to the original SPARQL 1.1 grammar. The restrictions have been studied empirically and must be further examined based on the formal models for the SPARQL language. To help expert users with the creation of "updatable" views, the query editor in

Table 1. Restricted SPARQL grammar for SPARQL_edit in EBNF notation. The table lists rules that differ from the original SPARQL 1.1 grammar [17]. Unsupported language features are crossed out.

[2]   Query   ::=
         Prologue
         (SelectQuery | ~~ConstructQuery | DescribeQuery |~~
           ~~AskQuery~~)
         ValuesClause

[7]   SelectQuery   ::=
         SelectClause DatasetClause?~~*~~ WhereClause
           SolutionModifier

[9]   SelectClause   ::=
         'SELECT' ('DISTINCT' | 'REDUCED')? (
         (Var | ~~('(' Expression 'AS' Var ')')~~)+ | '*')

[13]   DatasetClause   ::=
         'FROM' (DefaultGraphClause
           ~~| NamedGraphClause~~)

[18]   SolutionModifier   ::=
         ~~GroupClause?~~
           ~~HavingClause?~~ OrderClause? LimitOffsetClauses?

[53]   GroupGraphPattern   ::=
         '{' (~~SubSelect |~~ GroupGraphPatternSub) '}'

[56]   GraphPatternNotTriples   ::=
         ~~GroupOrUnionGraphPattern |~~
         OptionalGraphPattern | ~~MinusGraphPattern~~
         ~~| GraphGraphPattern | ServiceGraphPattern |~~
         Filter ~~| Bind | InlineData~~

[83]   PropertyListPathNotEmpty ::=
         (~~VerbPath |~~ VerbSimple) ObjectListPath
           (';' ((~~VerbPath |~~ VerbSimple)
           ObjectList)?)∗

[98]   TriplesNode   ::=
         ~~Collection |~~ BlankNodePropertyList

[100]   TriplesNodePath   ::=
         ~~CollectionPath |~~ BlankNodePropertyListPath

[109]   GraphTerm   ::=
         iri | RDFLiteral | NumericLiteral |
         BooleanLiteral | BlankNode ~~| NIL~~

SPARQL_edit marks queries which are not compliant with this subset of the SPARQL grammar.

Unambiguous Update Guaranty. Due to the structural freedom of RDF and the ambiguous nature of RDF blank nodes, it is possible to construct graphs where Q^- has unwanted side-effects on other RDF triples. In these edge cases with RDF blank nodes, the graph pattern of Q^- matches more than one RDF triple in the graph. An example in the context of the product graph from Fig. 1 would be a product node which has two identical weight properties connected to two different blank nodes which have again the same predicate (`schema:value`) and equal literals (e.g. `"10.5"^^xsd:decimal`). The two blank nodes might have different triples apart from the value triple. In this special case, Q^- from Listing 1.2 would change the weight literal value of the first and the second blank node. To guarantee safe RDF literal updates for cases with RDF blank nodes, an additional SPARQL/Select query Q_{check} is used to verify that Q^- updates exactly one RDF triple. Q_{check} is a simple SPARQL/Select query that uses the `WHERE` clause from Q^-. Depending on the size of its solution sequence $|\Omega|$, we can distinguish three cases:

- $|\Omega| = 0$: The graph pattern matching has no result because the relevant triples in the graph have been changed in the meantime.
- $|\Omega| = 1$: The ideal case in which Q^- will affect exactly one RDF triple.
- $|\Omega| \geq 2$: Q^- is ambiguous and would alter more than one RDF triple.

URI Editing. In principle, the algorithm could be generalized for editing not only literal objects but also object URIs. However, changes to the URI of an RDF triple's object can have drastic effects on the whole graph. Any URI alteration could disconnect or connect parts of the graph. The consequences of a URI modification for the graph structure cannot be foreseen from the tabular representation. From a user perspective, a lay user should not be able to change the graph structure by accident.

5   Implementation

SPARQL_edit is designed as a standalone Web application which is usable on different devices without installation. SPARQL_edit is implemented in JavaScript and TypeScript as a React single-page-application that can be served from any Web server or content delivery network. The application acts as a SPARQL client and renders the query results as an HTML table with HTML5 input controls for literal results. In order to simulate in-place editing of cells, a change submission immediately triggers an update query and refreshes the table to give feedback to the user. SPARQL_edit requires that the SPARQL server provides an endpoint for SPARQL 1.1 updates [24] and that it allows Cross-Origin Resource Sharing, a browser security feature for cross-domain HTTP requests.

Similar to the approach of rdfedit [23], SPARQL_edit uses configuration objects that are created by experts to simplify the work for non-technical users. Lay users can simply load predefined "SPARQL views" that include all information that is necessary for SPARQL_edit to create a certain table of data from a KG. A "SPARQL view" configuration is an RDF graph including information about:

– the SPARQL endpoint (query and update URL of an RDF dataset),
– the SPARQL/Select query for the view,
– metadata (description, originator, modification date, etc.) and
– the security mechanism for requests to the SPARQL server.
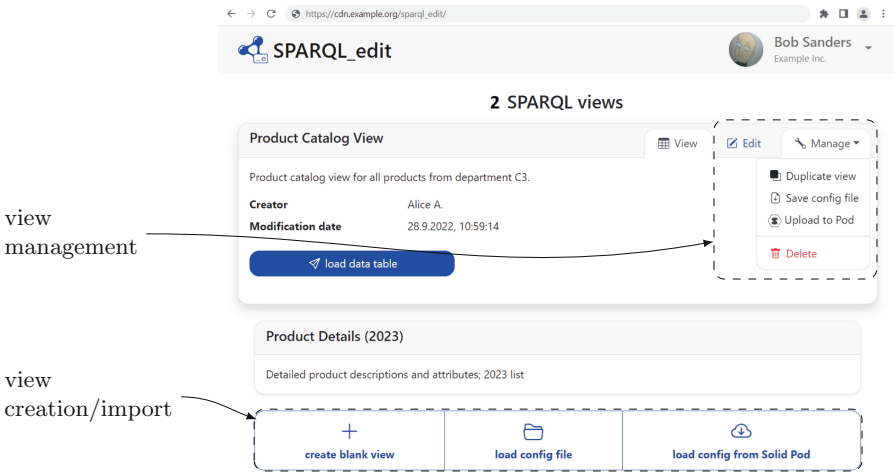
Fig. 4. Management of "SPARQL views" in SPARQL_edit

Instead of a back-end (database), the client-side application uses the HTML5 Web Storage to save view configurations in the browser. To allow for easy sharing of view configurations, views can be exported to or imported from text-based files in the JSON-LD RDF format. A KG expert is able to create and test a specific view configuration, upload it to a Web server or a Solid Pod and share it with one or more colleagues. In this way, non-technical users can work with predefined views on the KG without knowledge about SPARQL and RDF. Figure 4 shows the UI controls for the management of views in SPARQL_edit.

6   Applicability

The applicability of the presented solution in an enterprise scenario concerns the cooperation of several people and the performance in case of large data sets.

6.1 Performance

Many Linked Data exploration and visualisation solutions pay too little attention
to performance and scalability because they load a complete KG or dataset into
memory [4]. SPARQL_edit only loads the SPARQL query results for a specific
view which is usually only a subset of the whole graph. Due to the SPARQL/Up-
date mechanism, single changes are applied directly via the SPARQL endpoint
of the KG without re-writing the graph and without writing any RDF files.

The execution times of RDF literal update cycles for different sizes of solu-
tions sequences are shown in Fig. 5. The comparison of the running times for
the four steps from Sect. 2 reveal that the transmission and processing of the
SPARQL results are the key drivers for the execution time of a complete lit-
eral update cycle for growing numbers of query solutions. The algorithm and
the SPARQL/Update query are independent of the number of SPARQL results.
The algorithm is responsible for less than one percent of the total execution
time because it only uses the solution $\mu_{ed} \in \Omega$ which was modified. The running
time of the algorithm merely depends on the number of triple patterns in the
SPARQL/Select query. Experiments revealed a linear rise from `0.10` to `0.18` ms
on average when continuously increasing the number of triple patterns with vari-
ables from two to thirty. The reason for the increase are several loops over the
triple patterns of Q in the algorithm.
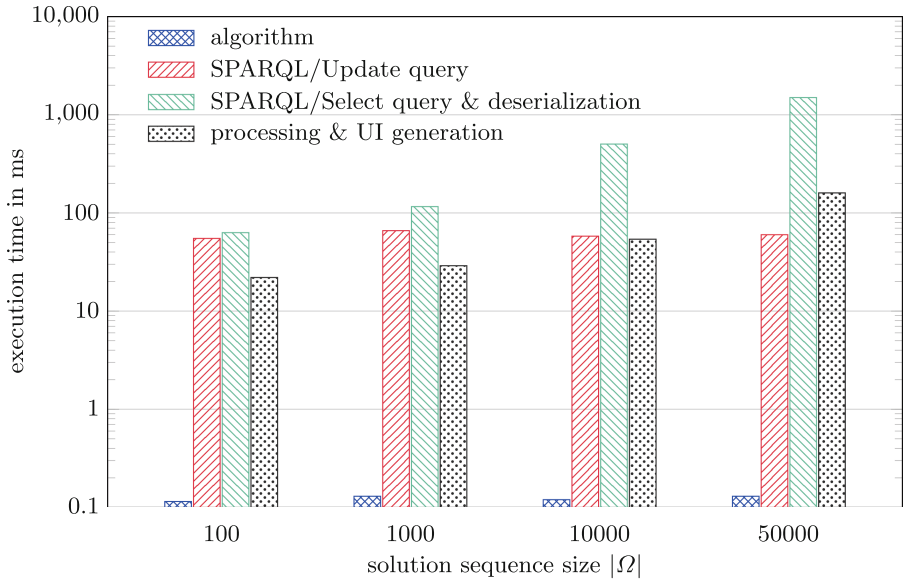
Fig. 5. Average execution times of the steps of a SPARQL_edit literal update cycle

6.2   Concurrent Literal Editing

If two SPARQL_edit users are concurrently editing the same triples of the same graph, inconsistencies may occur because the algorithm uses a slightly outdated state of the database for generating the update query Q^-. When the user loads a "SPARQL view", the application sends the specified SPARQL/Select query Q to the specified SPARQL endpoint of a dataset D and receives the solution sequence Ω. This Ω represents the solution for Q on a graph $G \in D$ at the time of the query evaluation. If relevant triples in G are modified by another user before Q^- is executed, Q^- is ineffective because its triple patterns do not match with the new triples in G. Although the ineffectiveness of Q^- can be considered as fail-safe, this behavior is very irritating to the user. This race condition can by prevented by means of the ambiguity check from Sect. 4.2. Prior to Q^-, SPARQL_edit executes the SPARQL/Select query Q_{check} to verify that Q^- will update exactly one RDF triple. If Q_{check} has no query results ($|\Omega| = 0$), the application notifies the user that the values of the selected table row have been changed in the meantime. The user can reload the view to see the current state of the KG.

6.3   Update Logging

Until today, an EKG is typically built with data from different databases which is translated to RDF as part of the KG building pipeline. If the EKG is only used for query answering, data changes are executed on the source databases and the building pipeline is rerun to update the graph. But if triples are updated directly in the KG, these changes would be lost after repeating the building pipeline. One solution is the logging of every KG update so that it can be reapplied after every new instantiation of the KG.

SPARQL_edit offers a feature for logging the update queries that are executed when a user changes an RDF literal value. On every literal update, a log is inserted into a user-defined provenance graph $G_{prov} \in D$. Instead of executing a separate insert query, the logging is included into Q^- in order to take advantage of the transactional query execution at the SPARQL endpoint. Q^- is extended with additional insert statements for the update log. The update log is modeled with PROV-O, the provenance ontology of the W3C [21], and includes at least Q^- itself and the captured execution time. Based on the timestamp for the query execution, the SPARQL/Update queries from multiple editors can be sorted chronologically. Following the principle of event sourcing, the states of the database can be reconstructed by adding up update increments. A detailed study of dynamic provenance for SPARQL updates was done by Halpin and Cheney [16] who researched the combination of the W3C PROV model and the semantics of SPARQL for RDF provenance.

If SPARQL_edit demanded users to login, e.g. using Solid authentication, the update logging could be used to track user activities – which user has made which change at which time.

7   Conclusion

To improve the adoption of enterprise knowledge graphs, non-expert users must be able to easily maintain and update single values in the graph. We present SPARQL_edit, a Web application that enables users without any knowledge about SPARQL or RDF to view query results and instantly edit or insert missing RDF literal values. The browser application connects to a SPARQL endpoint and presents query results in an interactive table that helps users to make changes to literal contents of the underlying RDF data. Expert users can create configurations for loading specific views on the KG data and share them with business users. The functionality, performance and applicability of our solution was studied by means of a demo implementation. In addition to the update algorithm, the open-source application implements features for view management and collaborative work.

The algorithm behind the SPARQL_edit application translates a change in a view into a SPARQL/Update query for updating the original KG. The generic view update translation approach reduces the upfront work for data engineers. In contrast to most existing read-write Linked Data tools, it is not necessary to explicitly define RDF translation rules. However, the automatic update query generation comes with restrictions which limit the possible views on the KG data. Therefore, we defined restrictions to the SPARQL query language that are necessary for the creation of "updatable" views. Although the restrictions have been studied empirically, they must be further examined based on the formal models for the SPARQL query language. Furthermore, the generic approach could be extended so that data engineers can define translations for problematic cases that exceed the capabilities of the algorithm.

Resource Availability Statement. Source code for the view-update algorithm and the Web application described in Sect. 5 are available from Github (cf., https://github.com/wintechis/sparqledit).

References

1. Berners-Lee, T., Connolly, D.: Notation3 (N3): a readable RDF syntax. W3C team submission. W3C (2011). http://www.w3.org/TeamSubmission/2011/SUBM-n3-20110328/
2. Berners-Lee, T., O'Hara, K.: The read-write linked data web. Phil. Trans. R. Soc. A **371**, 20120513 (2013). https://doi.org/10.1098/rsta.2012.0513
3. Berners-Lee, T., et al.: Tabulator redux: browsing and writing linked data. In: Bizer, C., et al. (eds.) Proceedings of the WWW2008 Workshop on Linked Data on the Web, LDOW. CEUR-WS (2008)

4. Bikakis, N., Sellis, T.: Exploration and visualization in the web of big linked data: a survey of the state of the art. In: Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference (2016)
5. Bizer, C., Seaborne, A.: D2RQ - treating non-RDF databases as virtual RDF graphs. In: Proceedings of the 3rd International Conference on Semantic Web Conference, ISWC 2004. LNCS. Springer, Heidelberg (2004)
6. Calvanese, D., et al.: The MASTRO system for ontology-based data access. Semant. Web **2**, 43–53 (2011). https://doi.org/10.3233/SW-2011-0029
7. Calvanese, D., et al.: OBDA with the Ontop framework. In: Lembo, D., Torlone, R., Marrella, A. (eds.) 23rd Italian Symposium on Advanced Database Systems, SEBD (2015)
8. Capadisli, S., Berners-Lee, T., Verborgh, R., Kjernsmo, K.: Solid protocol. Technical report. solidproject.org (2021). https://solidproject.org/TR/2021/protocol-20211217
9. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 concepts and abstract syntax. W3C recommendation. W3C (2014). https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/
10. De Giacomo, G., Lembo, D., Oriol, X., Savo, D.F., Teniente, E.: Practical update management in ontology-based data access. In: d'Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10587, pp. 225–242. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68288-4_14
11. Dörr, T.: RDF in HTML-forms. https://rdforms.org/. Accessed 18 Jul 2023
12. Eisenberg, V., Kanza, Y.: D2RQ/Update: updating relational data via virtual RDF. In: Proceedings of the 21st International Conference on World Wide Web, WWW 2012, Companion, pp. 497–498. ACM, New York (2012). https://doi.org/10.1145/2187980.2188095
13. Franconi, E., Guagliardo, P.: The view update problem revisited (2012). https://doi.org/10.48550/ARXIV.1211.3016
14. Furtado, A.L., Casanova, M.A.: Updating relational views. In: Kim, W., Reiner, D.S., Batory, D.S. (eds.) Query Processing in Database Systems. Topics in Information Systems, pp. 127–142. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-82375-6_7
15. Galkin, M., Auer, S., Scerri, S.: Enterprise knowledge graphs: a backbone of linked enterprise data. In: 2016 IEEE/WIC/ACM International Conference on Web Intelligence (WI), Omaha, USA, pp. 497–502. IEEE Computer Society (2016). https://doi.org/10.1109/WI.2016.0083
16. Halpin, H., Cheney, J.: Dynamic provenance for SPARQL updates. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 425–440. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_27
17. Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C recommendation. W3C (2013). https://www.w3.org/TR/2013/REC-sparql11-query-20130321/
18. Hert, M., Reif, G., Gall, H.C.: Updating relational data via SPARQL/update. In: Proceedings of the 2010 EDBT/ICDT Workshops, EDBT 2010. ACM, New York (2010). https://doi.org/10.1145/1754239.1754266
19. Hogan, A., et al.: Knowledge Graphs. Synthesis Lectures on Data, Semantics, and Knowledge, Springer, Cham (2022). https://doi.org/10.1007/978-3-031-01918-0
20. Konstantinou, N., Spanos, D.: Materializing the Web of Linked Data. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16074-0
21. McGuinness, D., Lebo, T., Sahoo, S.: PROV-O: The PROV Ontology. W3C recommendation. W3C (2013). https://www.w3.org/TR/2013/REC-prov-o-20130430/

22. Noy, N., et al.: Industry-scale knowledge graphs: lessons and challenges. Commun. ACM **62**(8), 36–43 (2019). https://doi.org/10.1145/3331166
23. Pohl, O.: rdfedit: user supporting web application for creating and manipulating RDF instance data. In: Closs, S., Studer, R., Garoufallou, E., Sicilia, M.-A. (eds.) MTSR 2014. CCIS, vol. 478, pp. 54–59. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13674-5_6
24. Polleres, A., Gearon, P., Passant, A.: SPARQL 1.1 update. W3C recommendation. W3C (2013). https://www.w3.org/TR/2013/REC-sparql11-update-20130321/
25. Rector, A., Noy, N.: Defining n-ary relations on the semantic web. W3C note. W3C (2006). https://www.w3.org/TR/2006/NOTE-swbp-n-aryRelations-20060412/
26. Seaborne, A.: SPARQL 1.1 query results JSON format. W3C recommendation. W3C (2013). https://www.w3.org/TR/2013/REC-sparql11-results-json-20130321/
27. Semantic Web Company: PoolParty GraphEditor - overview (2022). https://help.poolparty.biz/en/user-guide-for-knowledge-engineers/enterprise-features/poolparty-grapheditor---overview.html. Accessed 18 Jul 2023
28. Simsek, U., et al.: Knowledge Graph lifecycle: building and maintaining knowledge graphs. In: Proceedings of the 2nd International Workshop on Knowledge Graph Construction, KGC 2021, vol. 2873. CEUR-WS (2021)
29. Speicher, S., Malhotra, A., Arwe, J.: Linked data platform 1.0. W3C recommendation. W3C (2015). https://www.w3.org/TR/2015/REC-ldp-20150226/
30. Stadler, C., Arndt, N., Martin, M., Lehmann, J.: RDF editing on the web with REX. In: SEMANTICS 2015, SEM 2015. ACM (2015)
31. Sundara, S., Das, S., Cyganiak, R.: R2RML: RDB to RDF mapping language. W3C recommendation. W3C (2012). https://www.w3.org/TR/2012/REC-r2rml-20120927/
32. Tramp, S., Heino, N., Auer, S., Frischmuth, P.: RDFauthor: employing RDFa for collaborative knowledge engineering. In: Cimiano, P., Pinto, H.S. (eds.) EKAW 2010. LNCS (LNAI), vol. 6317, pp. 90–104. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16438-5_7
33. Unbehauen, J., Martin, M.: SPARQL update queries over R2RML mapped data sources. In: Eibl, M., Gaedke, M. (eds.) INFORMATIK 2017, pp. 1891–1901. Gesellschaft für Informatik, Chemnitz, Germany (2017). https://doi.org/10.18420/in2017_189
34. W3C: LDP implementations. https://www.w3.org/wiki/LDP_Implementations. Accessed 18 Jul 2023