Updating relational data via SPARQL/Update

Hert, M; Reif, G; Gall, H C

Abstract: Relational Databases (RDBs) are used in most current enterprise environments to store and manage data. The semantics of the data is not explicitly encoded in the relational model, but implicitly at the application level. Ontologies and Semantic Web technologies provide explicit semantics that allows data to be shared and reused across application, enterprise, and community boundaries. Converting all relational data to RDF is often not feasible, therefore we adopt a mediation approach for ontology-based access to RDBs. Existing mapping approaches focus on read-only access via SPARQL or as Linked Data but other data access interfaces exist, including approaches for updating RDF data. In this paper we present OntoAccess, an extensible platform for ontology-based read and write access to existing relational data. It encapsulates the translation logic in the core layer that provides the foundation of an extensible set of data access interfaces in the interface layer. We further present the formal definition of our RDB-to-RDF mapping, the architecture of our mediator platform, and a performance evaluation of the prototype implementation.

Posted at the Zurich Open Repository and Archive, University of Zurich
ZORA URL: https://doi.org/10.5167/uzh-42656

Updating relational data via SPARQL/Update

Hert, M; Reif, G; Gall, H C
Updating relational data via SPARQL/Update

Abstract

Relational Databases (RDBs) are used in most current enterprise environments to store and manage data. The semantics of the data is not explicitly encoded in the relational model, but implicitly at the application level. Ontologies and Semantic Web technologies provide explicit semantics that allows data to be shared and reused across application, enterprise, and community boundaries. Converting all relational data to RDF is often not feasible, therefore we adopt a mediation approach for ontology-based access to RDBs. Existing mapping approaches focus on read-only access via SPARQL or as Linked Data but other data access interfaces exist, including approaches for updating RDF data. In this paper we present OntoAccess, an extensible platform for ontology-based read and write access to existing relational data. It encapsulates the translation logic in the core layer that provides the foundation of an extensible set of data access interfaces in the interface layer. We further present the formal definition of our RDB-to-RDF mapping, the architecture of our mediator platform, and a performance evaluation of the prototype implementation.
ABSTRACT

Relational Databases are used in most current enterprise environments to store and manage data. The semantics of the data is not explicitly encoded in the relational model, but implicitly on the application level. Ontologies and Semantic Web technologies provide explicit semantics that allows data to be shared and reused across application, enterprise, and community boundaries. Converting all relational data to RDF is often not feasible, therefore we adopt an ontology-based access to relational databases. While existing approaches focus on read-only access, we present our approach OntoAccess that adds ontology-based write access to relational data. OntoAccess consists of the update-aware RDB to RDF mapping language R3M and algorithms for translating SPARQL/Update operations to SQL. This paper presents the mapping language, the translation algorithms, and a prototype implementation of OntoAccess.

1. INTRODUCTION

Relational Databases (RDBs) are used in most current enterprise environments to store and manage data. While RDBs are well suited to handle large amounts of data, they were not designed to preserve the data semantics. The meaning of the data is implicit on the application level and not explicitly encoded in the relational model. Ontologies and Semantic Web technologies provide explicit semantics in a common framework that allows data to be shared and reused across application, enterprise, and community boundaries [4]. Applying Semantic Web technologies in an enterprise environment enables data processing and exchange on a semantic level. Ontologies and RDF are used to build a semantic layer on top of existing databases that lifts data processing from the syntax to the semantic level. RDF and a shared ontology can be used to exchange data even if the individual relational schemata do not match. The introduction of background knowledge from an ontology can also be valuable in the implementation of a data integration layer on top of multiple relational data sources.

Converting all data in an RDB to RDF is often not feasible due to existing applications that rely on the relational representation of the data. Also, the performance of current triple store implementations remains below RDBs as recent benchmarks show [7]. Therefore, a mediation approach that performs an on demand translation of Semantic Web requests to SQL is the alternative that preserves the compatibility with existing relational applications while enabling access for ontology-based software to (co-)operate on the same data. In addition, mediation allows to further exploit the advantages of the well established database technology such as query performance, scalability, transaction support, and security.

Existing approaches for mapping RDBs to RDF focus on exposing the relational data to the Semantic Web. They provide SPARQL endpoints to query the data, but they neither address data updates nor the explicit application in an enterprise environment. Our contribution in this paper is the ontology-based write access to relational data via SPARQL/Update [19], the upcoming data manipulation language (DML) of the Semantic Web. We present the update-aware RDB to RDF mapping language R3M and algorithms for translating SPARQL/Update to SQL DML.

The remainder of this paper is organized as follows. Section 2 presents an overview of related work. The challenges of ontology-based write access to relational data and our approach OntoAccess are presented in Section 3. In Section 4, we introduce our update-aware RDB to RDF mapping language R3M and Section 5 specifies the algorithms for translating SPARQL/Update to SQL DML. Our prototype implementation is briefly described in Section 6, while Section 7 presents a feasibility study as a first evaluation of our approach. Section 8 concludes this paper with an outlook on future work.

2. RELATED WORK

Relational.OWL [11] defines an ontology to represent relational schemata and data in RDF. It maps tables and attributes to terms in that ontology and records information about primary/foreign keys as well as the data types of the attributes. This approach exposes the structure and syntax of the relational schema to the RDF representation and prohibits the direct reuse of existing domain vocabulary. RDQuery [12] adds a SPARQL interface on top of Relational.OWL that provides an on demand translation of SPARQL queries to SQL.

D2R [6, 5] is an approach for publishing RDBs on the Semantic Web. It enables the browsing of relational data as
RDF via dereferencable URIs and also provides an endpoint for SPARQL queries. D2Rs main goal is to provide content for the Web of Data, a web of interconnected data sets expressed in RDF (cf. the Linked Open Data initiative\(^1\)).

Virtuoso\(^2\) is a commercial database system from OpenLink Software that features RDF Views \(^3\) over relational data. A declarative meta-schema language is used to map terms of an ontology to concepts in the database schema. This enables the use of SPARQL as an alternative query language for the relational data. RDF Views are limited to read-only queries, updating the base data through these RDF Views is not supported.

Triplify \(^2\) is a light-weight approach to expose information from Web applications (e.g., discussion boards, content management systems) in RDF. It uses a set of application-specific SQL queries to extract data from the underlying RDB to generate RDF data from the results. The SQL queries have to be defined manually for each Web application, but the RDF generation is performed automatically according to a fixed process. Reuse of existing ontologies is possible via result column renaming in the SQL queries.

Mastro-I \(^9\) is an ontology-based data integration approach based on global-as-view (GAV) mappings. The individual source schemata are integrated through ontologies and a relational data federation tool. The mappings to the target ontology rely on SQL queries over the federated source schemata and bindings of the query results to terms in an ontology. Hence, the Mastro-I approach is limited to read-only data access as unrestricted data manipulations would be affected by the relational view update problem.

The World Wide Web Consortium (W3C) has recognized the importance of mapping relational data to the Semantic Web by starting the RDB2RDF incubator group\(^3\) (XG) to investigate the need for standardization. The XG recommends \(^17\) that the W3C starts a working group to define a standard RDB to RDF mapping language. However, they will not address the requirements for updating the relational data in a first version of the language.

View updates are a well-known problem in database research (e.g., \([3, 10, 15, 16]\)). Mapping RDBs to RDF can also be seen as defining RDF views over the relational data, therefore these views may be affected by the view update problem. Research in this area has shown that the requirements of updates have to be considered already in the specification of a view definition language (VDL). If a VDL is constructed to allow only the definition of bijective mappings (i.e., updates on the base data as well as the views can unambiguously be propagated to the opposite side), the hardest problems of the relational view update problem can be avoided (e.g., \([8]\)).

Object-relational mapping (ORM) is an approach to bridge the conceptual gap between object-oriented systems and the relational data model. ORMs such as Hibernate\(^3\) aim at using existing RDB infrastructure to persist data objects in object-oriented applications. This allows to benefit from established database technology while providing an object-oriented abstraction to the relational model. A mapping language is used to define the mappings of classes and attributes in the object-oriented system to tables and attributes in the RDB. The ORM component then generates the RDB schema according to this mapping and also provides means to store and retrieve objects.

### 3. ONTOACCESS APPROACH

ONTOACCESS \(^14\) is our approach for ontology-based access to RDBs that provides read and write access to the relational data. It currently consists of the update-aware mapping language R3M that bridges the conceptual gap between a RDB and an ontology as well as an access interface based on SPARQL that supports the upcoming SPARQL/Update language for data manipulations.

Updating relational data through Semantic Web technologies presents new challenges for mapping languages and mediation tools. The conceptual gap between the relational model and RDF (tuples vs. triples) causes that constraints from the RDB are transferred to the Semantic Web layer. As a consequence, some update requests are no longer valid compared to their application in a native triple store. The tuple-oriented nature of the relational model requires that a certain amount of data is known about each entity (i.e., attributes declared as mandatory). This and other requirements can be enforced in the database schema with integrity constraints that may not be equally reflected in ontologies and RDF, especially if existing vocabularies are reused. However, to enable ontology-based write access to RDBs these constraints must be respected and errors resulting from constraint violations should be handled appropriately. If information about these constraints is stored in the mapping, it can be used to detect invalid update requests and to provide semantically rich feedback to the client.

We take the RDB schema of a publication system as the use case for this paper. The database stores information about authors and their publications. Figure 1 depicts the database schema used in this example with the tables, their attributes and data types. Each table has a distinct primary key called id of type integer. The publication and author tables represent the main concepts in the use case. A publication is composed of a title, a publication year, a publication type, and a publisher. While title and year are data attributes, type and publisher are foreign keys to the tables pubtype and publisher respectively. Each of those tables contains one textual attribute as a label for the publisher/the type of the publication. All valid publications must have

---

1. \(\text{http://linkeddata.org}\)
2. \(\text{http://virtuoso.openlinksw.com}\)
3. \(\text{http://www.hibernatedb.org} /\text{incubator/rdb2rdf/}\)
4. \(\text{http://www.hibernatedb.org}\)

---

**Figure 1:** RDB schema of the publication use case

<table>
<thead>
<tr>
<th>publication</th>
<th>publication_author</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>– id : INTEGER</td>
<td>– publication : INTEGER</td>
<td>– id : INTEGER</td>
</tr>
<tr>
<td>– title : VARCHAR</td>
<td>– publication_author : INTEGER</td>
<td>– title : INTEGER</td>
</tr>
<tr>
<td>– year : INTEGER</td>
<td>– author : INTEGER</td>
<td>– id : INTEGER</td>
</tr>
<tr>
<td>– type : INTEGER</td>
<td></td>
<td>– title : VARCHAR</td>
</tr>
<tr>
<td>publisher : INTEGER</td>
<td></td>
<td>– firstname : VARCHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– lastname : VARCHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– email : VARCHAR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>publisher</th>
<th>pubtype</th>
<th>team</th>
</tr>
</thead>
<tbody>
<tr>
<td>– id : INTEGER</td>
<td>– id : INTEGER</td>
<td>– id : INTEGER</td>
</tr>
<tr>
<td>– name : VARCHAR</td>
<td>– type : INTEGER</td>
<td>– firstname : VARCHAR</td>
</tr>
<tr>
<td>– price : DECIMAL</td>
<td></td>
<td>– lastname : VARCHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– email : VARCHAR</td>
</tr>
</tbody>
</table>

---

**Table 1:** RDF schema of the publication use case

<table>
<thead>
<tr>
<th>publication</th>
<th>publication_author</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>– id : INTEGER</td>
<td>– publication : INTEGER</td>
<td>– id : INTEGER</td>
</tr>
<tr>
<td>– title : VARCHAR</td>
<td>– publication_author : INTEGER</td>
<td>– title : INTEGER</td>
</tr>
<tr>
<td>– year : INTEGER</td>
<td>– author : INTEGER</td>
<td>– id : INTEGER</td>
</tr>
<tr>
<td>– type : INTEGER</td>
<td></td>
<td>– title : VARCHAR</td>
</tr>
<tr>
<td>publisher : INTEGER</td>
<td></td>
<td>– firstname : VARCHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– lastname : VARCHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– email : VARCHAR</td>
</tr>
</tbody>
</table>

---

**Figure 1:** RDB schema of the publication use case

<table>
<thead>
<tr>
<th>publication</th>
<th>publication_author</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>– id : INTEGER</td>
<td>– publication : INTEGER</td>
<td>– id : INTEGER</td>
</tr>
<tr>
<td>– title : VARCHAR</td>
<td>– publication_author : INTEGER</td>
<td>– title : INTEGER</td>
</tr>
<tr>
<td>– year : INTEGER</td>
<td>– author : INTEGER</td>
<td>– id : INTEGER</td>
</tr>
<tr>
<td>– type : INTEGER</td>
<td></td>
<td>– title : INTEGER</td>
</tr>
<tr>
<td>publisher : INTEGER</td>
<td></td>
<td>– id : INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– title : INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– firstname : VARCHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– lastname : VARCHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– email : VARCHAR</td>
</tr>
</tbody>
</table>

---

**Table 1:** RDF schema of the publication use case
a title and a publication year, therefore the corresponding two attributes have a NOT NULL constraint. An author consists of a title, an email address, a firstname, a lastname, and an affiliation to a research team. A valid author must have at least a last name, therefore a NOT NULL constraint is defined on the lastname attribute. The team attribute is a foreign key to the table of the same name, while the rest of the attributes contain data values. The team table stores information about research groups, in particular the name of the team and a code for abbreviation. The table publication_author is a link table that represents the N:M relationship between publications and authors.

Figure 2 depicts the domain ontology for our example. We reused vocabulary from the Friend of a Friend (FOAF) project\(^5\) and the Dublin Core (DC) metadata standard\(^6\) to form this ontology. We also added our own ontology elements (ONT) if there were no adequate terms in the existing two vocabularies. The figure shows the five classes of our domain ontology as well as the properties that are used with each class and their respective range.

![Figure 2: Domain ontology](image)

### 4. RDB TO RDF MAPPING LANGUAGE

R3M is an update-aware RDB to RDF mapping language that records additional information of the database schema to support data manipulations and to detect invalid update requests during the translation process. Updatability and simplicity were two of the main design goals of this mapping language. It is expressed in RDF and uses the R3M mapping-wide URI prefix mechanism to generate the instance URIs of all the classes defined in the mapping. The URI of an instance is composed of two parts, the mapping-wide URI prefix defined here and an individual URI pattern defined in each TableMap. The main purpose of this mapping-wide URI prefix is to ease the definition of mappings similar to the prefix mechanism in XML Namespaces.\(^7\) Finally, all tables that belong to this database schema are listed as TableMaps (lines 7 to 12).

```xml
Listing 1: Example DatabaseMap

A TableMap represents the mapping of an individual database table (Listing 2). It contains the name of the table (line 2) and the ontology class it is mapped to (line 3). The URI pattern (line 4) is appended to the mapping-wide URI prefix to generate the instance URIs for this class or overrides it if the pattern itself forms a valid URI (i.e., if it starts with http://, mailto:, etc.). Attribute values from the database table can be included in the pattern by specifying the name of the attribute between double percentage signs. Typically, at least the primary key attributes are included in the URI pattern (e.g., %id% where id is the name of the primary key attribute). A TableMap further contains a list of AttributeMaps (lines 5 to 10) which map attributes of this table to properties in the ontology.

```xml
Listing 2: Example TableMap

Each attribute of a database table is represented by an AttributeMap (Listings 3) that contains the name of the attribute in the database schema (line 2) as well as the name of the type of attribute expressed as a URI. Attributes without a URI pattern are mapped to literal values (or another instance (e.g., foreign keys). Link tables are used in RDBs to describe N:M relationships among relations. In RDF, such auxiliary constructs are not needed, which is why R3M features explicit support to map these tables to object properties instead of classes.

The root element of a mapping in R3M is called DatabaseMap (Listing 1). It abstractly represents the database and contains access information for the mediator (lines 2 to 5). Optionally, a URI prefix can be specified (line 6) that is used in generating the instance URIs of all the classes defined in the mapping. The URI of an instance is composed of two parts, the mapping-wide URI prefix defined here and an individual URI pattern defined in each TableMap. The main purpose of this mapping-wide URI prefix is to ease the definition of mappings similar to the prefix mechanism in XML Namespaces.\(^7\) Finally, all tables that belong to this database schema are listed as TableMaps (lines 7 to 12).

\(^5\)http://www.foaf-project.org/
\(^6\)http://dublincore.org/
\(^7\)http://www.w3.org/TR/xml-names11/
of the ontology property it is mapped to (line 3). Depending on the type or value of the attribute, the property can be an Object- or a DataProperty. This is reflected in the mapping vocabulary as either r3m:mapsToObjectProperty or r3m:mapsToDataProperty. Additionally, an AttributeMap includes information about constraints defined on the attribute (e.g., that it is a foreign key and the table it references: lines 4 and 5). In the current implementation, the following constraints are supported: r3m:PrimaryKey, r3m:ForeignKey, r3m:NotNull, and r3m:Default.

Listing 3: Example AttributeMap

A LinkTableMap is provided to map link tables to properties in the ontology (Listing 4). It specifies the name of the link table in the database (line 2) and the object property it is mapped to (line 3). A link table always contains two foreign key attributes that point to the tables of the N:M relationship. Therefore, a triple with the property representing this link table has a subject and an object mapped from two tables. The attribute pointing to the table of the subject is represented as the subject attribute (line 4) and the attribute pointing to the table of the object as the object attribute (line 5). They link to AttributeMaps that are not mapped to any property but record the names of the attributes and the tables they reference (e.g., Listing 5).

Listing 4: Example LinkTableMap

A basic R3M mapping can be generated automatically from the database schema if it explicitly provides information about foreign key relationships. The only part of the mapping definition that cannot easily be automated is the assignment of domain ontology terms to the individual concepts in the database. However, (graphical) tool support can and will be provided to further decrease the user’s effort in defining a mapping.

5. SPARQL/UPDATE TO SQL DML

SPARQL [18] is the W3C recommendation of a query language for the Semantic Web. It is currently limited to read-only access to RDF data as it does not provide any means to insert, delete, or modify data. The Semantic Web community made efforts to close this gap, which lead to the SPARQL/Update [19] proposal for an RDF data manipulation language. SPARQL/Update does also serve as the basis for the update functionality in the relaunched W3C SPARQL working group (WG). The proposed version of SPARQL/Update consists of three update operations: (1) INSERT DATA (Listing 6) to insert new triples into an RDF graph; (2) DELETE DATA (Listing 7) to remove known triples from a graph; and (3) MODIFY (Listing 8) to delete and/or insert data based on triple templates that are matched against a triple pattern in a shared WHERE clause. The MODIFY operation basically corresponds to two SPARQL CONSTRUCT queries (with the same WHERE clause) where the resulting RDF triples get removed from and added to the data.

Angles and Gutierrez showed in [1] that SPARQL has the same expressive power as relational algebra and consequently that SPARQL can be fully translated to SQL. From these findings and the fact that SPARQL/Update is based on SPARQL follows that SPARQL/Update is also fully translatable to SQL DML, albeit not directly as we will see later.

5.1 INSERT DATA / DELETE DATA

INSERT DATA and DELETE DATA operations consist of sets of triples that are either added to or removed from the existing data. Their translation to SQL is therefore very similar and differs mainly in the type of SQL statement that is generated. It is important for the understanding of the translation algorithm to recall how a database schema is mapped to an ontology: tables representing domain concepts are mapped to classes, while attributes and link tables are represented as ontology properties. We will use

Algorithm 1 RDF triples to SQL DML translation

http://www.w3.org/2009/sparql/wiki/Main_Page
the `INSERT DATA` operation depicted in Listing 9 as an example to explain the translation algorithm (Algorithm 1). In the first step (line 1), the triples need to be grouped according to equal subjects as these triples all represent data about the same entity and therefore target the same table. The triples in our example operation all use the same subject, hence this step returns one group containing all original triples. Each such group is then handled individually (line 2). In step 2 (line 3), the table affected by this group of triples is identified through the URI of their subject. The subject URI in our example is `http://example.org/db/author1`. If we recall the mapping (cf. Listing 1 and Listing 2), we find that this URI matches the pattern `http://example.org/db/author%id%` and therefore identifies the table as `author`. Further, we can extract the value 1 for the primary key attribute `id`. Next, the validity of the request is checked in step three (line 4), i.e., it is tested if the data in the request meets the constraints in the relational schema. For instance, in the case of an `INSERT DATA` operation a triple must be present containing a property for every corresponding database attribute that has a `NotNull` constraint but no `Default` value. This requirement is trivially met in our `INSERT DATA` operation as it contains triples with properties matching every attribute of the `author` table. Step four (line 5) generates the respective SQL statement by looking up the properties in the corresponding `TableMap` of the current subject and then adding the attribute name as well as the value extracted from the triple’s object to the SQL statement. In the example this means for instance that the property `foaf:firstName` is looked up and matched to the `firstname` attribute (cf. Listing 3). The attribute name is added to the SQL statement together with the extracted value from the object, namely `5`. The other triples are processed likewise. Steps 2 to 4 are repeated for each group of triples and the generated SQL statements are collected (line 6). After all groups are processed, in step five (line 11) the collected SQL statements are sorted according to the foreign key relationships among the affected tables. Although, from a theoretical point of view this is not necessary if all statements are executed in the context of a single transaction, existing RDB systems check constraints such as referential integrity already during a transaction. Consequently, executing the generated statements in an arbitrary order may result in the failure of the transaction whereas their execution in the sorted order would succeed. Sorting in our example is trivial as there is only one SQL statement. The sixth and last step (line 12) executes the SQL statements in the previously generated sort order. All generated SQL statements that correspond to a single SPARQL/Update operation are executed within the context of one database transaction to ensure the atomicity of the SPARQL/Update operation. Listing 10 shows the translated SQL `INSERT` statement generated from our example SPARQL/Update `INSERT DATA` operation.

```
Listing 10: Translated SQL INSERT statement
```

The `INSERT DATA` operation of SPARQL/Update can be translated to SQL DML according to the algorithm described in the prior section. Depending on the state of the database, the translation results in either an `INSERT INTO` or an `UPDATE` SQL statement. The triple-oriented nature of RDF permits to insert only the minimal data about an entity with a first `INSERT DATA` operation (e.g., just the last name of an author) and later add more information with a second `INSERT DATA` (e.g., the first name and email address of said author). From the RDB perspective, this results first in a SQL `INSERT` statement that creates a new row in a database table for this entity with `NULL` values for all missing attributes (if this complies with the given constraints). The second `INSERT DATA` operation (with the additional data) translates to an SQL `UPDATE` statement that replaces the `NULLs` with actual values. This means, it has to be checked if the entity already exists in the database as this determines the type of the generated SQL statement.

```
DELETE DATA
```

The SPARQL/Update `DELETE DATA` operation is translated according to Algorithm 1 as well. The translation of this operation can also result in two different types of SQL statements depending on the state of the database and the operation. If the data in the operation represents only a subset of the data in the database, the operation is translated to a SQL `UPDATE` statement that sets all mentioned attributes to `NULL` (if this complies with the given constraints). Only if the data in the request operation equals all remaining (i.e., non-null) data in the database, the resulting SQL statement is a `DELETE` that removes the complete row from the database. Therefore, the tuple for the affected entity must be retrieved and analyzed during the translation.

```
5.2 MODIFY
```

The `MODIFY` operation in SPARQL/Update cannot directly be translated to SQL as there is no equivalent statement in the SQL DML. `MODIFY` is an atomic combination of a `DELETE` and an `INSERT` that in general is not limited to replacing triples, but can also add/remove arbitrary triples. In contrast, the `UPDATE` statement in SQL is limited to modifying existing data. However, the reuse of the SPARQL grammar in SPARQL/Update makes a translation in multiple steps possible. Algorithm 2 describes how the `MODIFY` operation is translated to SQL. We will use the `MODIFY` operation depicted in Listing 11 as an example to explain the algorithm. It replaces any email address of the author "Matthias Hert" with a new address (`hert@ifi.uzh.ch`).

```
Listing 9: Example INSERT DATA operation
```

```
Listing 11: Example MODIFY operation
```

```
Listing 12: Example SQL DELETE statement
```
Algorithm 2 MODIFY to SQL DML translation

1: delete ← extractDelete(modify)
2: insert ← extractInsert(modify)
3: where ← extractWhere(modify)
4: select ← createSelect(where)
5: selectSQL ← translateSelect(select)
6: results ← executeSQL(selectSQL)
7: for all binding in results do
8: deleteData ← createDeleteData(delete, binding)
9: insertData ← createInsertData(insert, binding)
10: deleteSQL ← translateDelete(deleteData)
11: insertSQL ← translateInsert(insertData)
12: executeSQL(deleteSQL, insertSQL)
13: end for

First, the MODIFY operation is separated into its individual parts, the INSERT, DELETE, and WHERE clauses (lines 1 to 3). The WHERE part is used to create a SPARQL SELECT query (line 4) that retrieves the data needed for the DELETE and INSERT templates. It is translated to SQL (line 5) and evaluated on the relational data (line 6). Based on the result bindings of that query, one DELETE DATA (line 8) and one INSERT DATA (line 9) operation are built for each binding (line 7) according to the DELETE and INSERT templates of the original MODIFY operation. In our example, the SELECT query returns just one result binding, namely `ex:author6` for the variable `x` and `mailto:hert@ifi.uzh.ch` for `mbox`. Therefore, one DELETE DATA and one INSERT DATA operations are built based on that binding as shown in Listing 12. These are then translated (lines 10 and 11) and executed (lines 12) according to Algorithm 1 described in the previous sections.

Listing 11: Example MODIFY operation

```sql
MODIFY
DELETE {
}
INSERT {
  ?x foaf:mbox <mailto:hert@example.com> .
}
WHERE {
  ?x rdf:type foaf:Person ;
  foaf:firstName "Matthias" ;
  foaf:family_name "Hert" ;
  foaf:mbox ?mbox .
}
```

Listing 12: Generated DELETE DATA and INSERT DATA operations

In many cases the MODIFY will actually represent a modification of data or rather a replacement of triples. Then, one optimization is possible by omitting those DELETE DATA operations that have a corresponding INSERT DATA, i.e., the triples differ only in their object. In these cases, the delete would set an attribute value to NULL and the insert sets the same attribute to a new value, therefore the delete is redundant and can be omitted.

6. PROTOTYPE IMPLEMENTATION

Based on our mapping language R3M and the SPARQL/Update to SQL DML translation algorithms described in the previous sections, we developed a prototype that mediates between SPARQL/Update requests and an RDB. Implemented as a HTTP endpoint, it allows clients to remotely manipulate the relational data. Incoming SPARQL/Update operations are parsed from the HTTP requests and forwarded to the translation module. There, the algorithm of Section 5.1 is used to generate equivalent SQL statements based on a R3M mapping definition. The translated operation is executed by the database engine and a confirmation or error message is returned to the translation module. This message is then converted to an RDF representation and sent back to the client.

Currently, the implementation is limited to INSERT DATA and DELETE DATA operations, but support for MODIFY and SPARQL queries is under development. Also, a more powerful feedback protocol is planned that will provide semantically rich error information to the client. A future version of the prototype implementing these features will be released to the public.

7. FEASIBILITY STUDY

For a first evaluation of our approach we present a feasibility study based on the RDF schema and the domain ontology introduced in Section 3. Table 1 summarizes the mapping from tables and attributes of the database schema to classes and properties of the domain ontology. The first column specifies the table and the corresponding class. For each table, column two lists the attributes and the properties they are mapped to.

The `publication` table is mapped to `foaf:Document`. The attributes `title` and `publisher` are mapped to corresponding properties from DC, while `year` and `type` use properties from our own ontology ONT. The tables `publisher` and `subtype` as well as their attributes are all mapped to terms of our application-specific ontology. The `author` table is represented as `foaf:Person`. Its attributes are mapped to equivalent concepts from the FOAF vocabulary with the exception of `team` that uses a property from ONT. The table `team` is represented as the class `foaf:Group` with its `name` attribute mapped to `foaf:name` and `code` to `ont:teamCode`. The `publication_author` table is a link table that represents the N:M relationship between `publications` and `authors`. Therefore, as described in Section 4, it is not mapped to a class but to the property `dc:creator` instead.

This mapping definition enables our mediation prototype to process SPARQL/Update operations. In the remainder of this section, we present example SPARQL/Update operations and the translated SQL statements as generated by our prototype.

Listing 13 shows a simple SPARQL/Update INSERT DATA request that inserts data about a team. It affects only a single database table and is therefore translated to one SQL INSERT statement (Listing 14).

Listing 15 depicts a more complex INSERT DATA request.
Table 1: Use case mapping overview

<table>
<thead>
<tr>
<th>table</th>
<th>attribute</th>
<th>property</th>
</tr>
</thead>
<tbody>
<tr>
<td>publication</td>
<td>title</td>
<td>dc:title</td>
</tr>
<tr>
<td></td>
<td>year</td>
<td>ont:pubYear</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>ont:pubType</td>
</tr>
<tr>
<td></td>
<td>publisher</td>
<td>dc:publisher</td>
</tr>
<tr>
<td>publisher</td>
<td>name</td>
<td>ont:name</td>
</tr>
<tr>
<td>ptype</td>
<td>type</td>
<td>ont:type</td>
</tr>
<tr>
<td>author</td>
<td>email</td>
<td>foaf:mbox</td>
</tr>
<tr>
<td></td>
<td>firstname</td>
<td>foaf:firstName</td>
</tr>
<tr>
<td></td>
<td>lastname</td>
<td>foaf:family.name</td>
</tr>
<tr>
<td>team</td>
<td>team</td>
<td>ont:team</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>code</td>
<td>ont:teamCode</td>
</tr>
</tbody>
</table>

LISTING 13: Example INSERT DATA operation

```sql
INSERT DATA {
  ont:teamCode "DATABASE" ;
  foaf:name "Database Technology" ;
}
```

LISTING 14: Translated SQL INSERT statement

It contains a complete data set, i.e., the request inserts new data into every database table and will therefore generate multiple SQL statements. The order of the triples in the request is irrelevant as the translated SQL statements are sorted based on the foreign key dependencies between the affected tables. Listing 16 shows the generated SQL statements sorted in their order of execution.

Listing 17 shows an example SPARQL/Update DELETE DATA operation that removes the email address of an existing author. As the respective entry in the author table contains more information than just the email address, this request is translated to a SQL UPDATE statement (Listing 18) according to Algorithm 1 described in Section 5.1.

The SPARQL/Update requests presented in this section are just examples, a user is free to phrase arbitrary requests. They will be translated to SQL DML successfully as long as they adhere to the ontology terms from the mapping and respect the constraints of the database schema.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented our approach ONTOACCESS that enables the manipulation of relational data via SPARQL/Update. We introduced the update-aware RDB to RDF mapping language R3M that captures additional information about the database schema, in particular about integrity constraints. This information enables the detection of update requests that are invalid from the RDB perspective. Such requests cannot be executed by the database engine as they would violate integrity constraints of the database schema. The information can also be exploited to provide semantically rich feedback to the client. Therefore, the causes for the rejection of a request and possible directions for improvement can be reported in an appropriate format.

Future work is planned for various aspects of ONTOACCESS. Further research needs to be done on bridging the conceptual gap between RDBs and the Semantic Web. Ontology-based write access to the relational data creates completely new challenges on this topic with respect to read-only approaches. The presence of schema constraints in the database can lead to the rejection of update requests that would otherwise be accepted by a native triple store. A feedback protocol that provides semantically rich information about the cause of a rejection and possible directions for improvement plays a major role in bridging the gap. Other database constraints such as assertions have to be evaluated as well to see if they can reasonably be supported in the mapping. Also,
a more formal definition of the mapping language will be provided. Furthermore, we will extend our prototype implementation to support the SPARQL/Update MODIFY operation, SPARQL queries, and the just mentioned feedback protocol.

9. REFERENCES