University of Zurich[UZH]

Matthias Hert
Gerald Reif
Harald Gall

Ontoaccess – an extensible platform for RDF-based read and write access to relational databases

TECHNICAL REPORT – No. IFI-2012.05

2012

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Matthias Hert, Gerald Reif, Harald Gall
Ontoaccess – an extensible platform for RDF-based read and write access to relational databases
Technical Report No. IFI-2012.05
Software Evolution and Architecture Lab
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
http://seal.ifi.uzh.ch

OntoAccess – An Extensible Platform for RDF-based Read and Write Access to Relational Databases

Matthias Hert[a], Gerald Reif[b], Harald Gall[a]

[a] *s.e.a.l. – Software Evolution and Architecture Lab, Department of Informatics, University of Zurich, Binzmuehlestrasse 14, 8050 Zurich, Switzerland*
[b] *ipt – Innovation Process Technology, Switzerland*

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 RDF-based access to RDBs. Existing RDB-to-RDF mapping approaches focus on read-only access via SPARQL or Linked Data but other data access interfaces exist, including approaches for updating RDF data (*e.g.,* Semantic Web frameworks such as Jena, Sesame, and RDF2Go; ChangeSet). In this paper we present OntoAccess, an extensible platform for RDF-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 and implementation of our mediator platform, a semantic feedback protocol to bridge the conceptual gap between the relational model and RDF as well as a performance evaluation of the prototype implementation.

Keywords: RDF-based Data Access, Mediation Platform, Relational Databases

1. Motivation

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 at the application level but not explicitly encoded in the relational model.

The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries [1]. Although developed for the Web, these Semantic Web technologies have proven to be useful in other domains as well, especially if data from different sources has to be exchanged or integrated (*e.g.,* [2, 3, 4]).

Ontologies and RDF are used to build a semantic layer that lifts data processing and exchange from the syntactic to the semantic level. In existing systems, however, it is not always possible or desirable to convert all relational data to RDF as other applications rely on the *relational* representation. Adapting or replacing these applications would require a prohibitive migration effort. Therefore, we suggest a mediation approach that performs an on demand translation of Semantic Web requests. This results in a cooperative use of the data in RDF-based as well as relational applications. In addition, mediation allows one to further exploit the advantages of the well established database technology such as query performance, scalability, transaction support, and security.

In the area of Semantic Web technologies, SPARQL [5] is the standard language for querying RDF data but other popular access interfaces exist such as Semantic Web frameworks (*e.g.,* Jena,[1] Sesame[2], RDF2Go[3]) and Linked Data.[4] Further, the Semantic Web currently lacks a standard data manipulation language (DML). SPARQL/Update [6] was proposed to the World Wide Web Consortium (W3C) as a DML and is being incorporated in the upcoming SPARQL 1.1[5] recommendation. In the meantime, several other approaches for updating RDF data have emerged (*e.g.,* ChangeSet,[6] GUO[7]). Although not approved as standards, these approaches are implemented and used in applications. The upcoming SPARQL 1.1 will further introduce a new data access interface in the form of the *Graph Store HTTP Protocol* [7]. Therefore, a RDB-to-RDF mediator should not be limited to a single data access interface (*e.g.,* SPARQL). Instead, it should be flexible and extensible to support multiple and also future data access interfaces.

The conceptual gap between the relational model and RDF affects the processing of Semantic Web requests if ontology terms are referenced that cannot be mapped to the RDB schema

Email addresses: hert@ifi.uzh.ch (Matthias Hert), gerald.reif@ipt.ch (Gerald Reif), gall@ifi.uzh.ch (Harald Gall)

[1] http://openjena.org/
[2] http://openrdf.org/
[3] http://rdf2go.semweb4j.org/
[4] http://linkeddata.org/
[5] http://www.w3.org/2009/sparql/wiki/Main_Page
[6] http://n2.talis.com/wiki/ChangeSets
[7] http://webr3.org/specs/guo/

or if a (write) request would violate constraints of the RDB schema. While a read-only query will simply return no results, a write request may not be processable and result in an error. Rejecting such requests may be confusing to an RDF-based client if the request is valid in RDF. An RDB-to-RDF mediator should provide feedback about rejected requests in a semantic format understandable by the client, *i.e.,* in RDF.

The contributions of this paper are the extensible RDB-to-RDF mediation platform OntoAccess [8] and a formal definition of its RDB-to-RDF mapping. The formal definition includes proofs that mappings expressed in this language are bidirectional, *i.e.,* support for read and write access to the data is provided. We present the architecture and implementation of the OntoAccess platform, including a semantic feedback protocol that provides recommendations to the client on how to change invalid write requests for the better.

The remainder of this paper is structured as follows. Section 2 presents an overview of related work in the area of RDB-to-RDF mapping. In Section 3, we formally define our RDB-to-RDF mapping and present examples in the RDF-based syntax called R3M. Section 4 explains the architecture and implementation of our mediator platform in detail. Section 5 introduces our semantic feedback protocol that bridges the conceptual gap between the relational model and RDF. The evaluation in Section 6 demonstrates the extensibility of our platform and shows that our approach performs comparable or better than the state-of-the-art. It further summarizes a case study we performed in the domain of software evolution analysis. Section 8 concludes this paper with a summary and an outlook on future work.

2. Related Work

Mapping RDBs to RDF is an active field of research resulting in many mapping languages and approaches (*e.g.,* [9, 10, 11, 12, 13, 14]).

D2R [13] is an approach for publishing existing relational databases on the Semantic Web. Based on mappings expressed in the D2RQ [9] mapping language, it enables browsing the relational data as RDF via dereferenceable URIs (*i.e.,* as Linked Data) and querying it via SPARQL. Further, D2R provides extensions for the Semantic Web frameworks Jena and Sesame that enable accessing the mapped RDBs via those APIs. However, D2R is limited to read-only data access, updating RDF data is not supported. *D2R/Update*[8] was an attempt to add write access to the D2R approach, but it turned out to be impractical without restricting the existing D2RQ mapping language.

The *Virtuoso* Universal Server features RDF Views [10] to expose relational data on the Semantic Web. A declarative Meta Schema Language is used for defining the mapping of SQL data to RDF vocabularies. This enables the use of SPARQL as an alternative query language for the relational data. Likewise, Virtuoso implements a Linked Data interface to these views. RDF Views are limited to read-only queries, updating the relational base data is not supported.

[8] http://d2rqupdate.cs.technion.ac.il/

R_2O [11] is an extensible and fully declarative language to describe mappings between relational database schemata and ontologies. R_2O is aimed at situations where the similarity between the ontology and the database model is low. It has been conceived to be expressive enough to cope with complex mapping cases where one model is richer, more generic/specific, or better structured than the other. This high expressiveness renders R_2O mappings read-only.

The W3C has recognized the importance of mapping relational data to the Semantic Web by starting the RDB2RDF Incubator Group[9] (XG) to investigate the need for standardization. The XG recommended [15] that the W3C initiates a working group (WG) to define a vendor-independent RDB-to-RDF mapping language. The RDB2RDF WG[10] started its work on R2RML [14] in late 2009. According to their charter [16], the requirements for updating relational data are out of scope and are therefore not addressed by the WG. It was further shown in [17] that adding write support to the R2RML approach is impractical.

pushback[11] is a W3C community project to develop a methodology for writing back changes in RDF data generated by wrappers of Web 2.0 APIs. Write support is enabled with RDF-annotated HTML forms, so-called RDForms, and mappings to the native write interfaces of the Web 2.0 APIs. Since *pushback* is focused on Web applications, direct support for modifying RDBs is not in the scope of the project.

We further refer the reader to the survey [18] conducted by the W3C RDB2RDF Incubator Group[12] for a detailed overview of existing RDB-to-RDF mapping approaches and to [19] for a feature-based comparison of the mapping *languages*.

3. OntoAccess Mapping

Mediation requires a mapping from concepts in a RDB schema to vocabulary terms defined in an ontology. Several of such mapping languages exist. However, thorough investigations revealed that they are unsuitable for RDF-based write access to relational data (*cf.* Section 2). Existing mapping languages would need to be extended to include additional information about the database schema required to support write access. For instance, detailed information about foreign key relationships and other integrity constraints is needed to detect invalid write requests. The mapping languages would also need to be restricted to avoid the view update problem [20]. Most existing approaches employ SQL views on the relational schema to define mappings. While this results in a high expressivity, it also means that in the case of write access such mappings are affected by the view update problem, *i.e.,* write access in general is impractical. Therefore, we designed our mapping language R3M to explicitly address write support. It extends the mapping

[9] http://www.w3.org/2005/Incubator/rdb2rdf/
[10] http://www.w3.org/2001/sw/rdb2rdf/
[11] http://esw.w3.org/PushBackDataToLegacySources
[12] http://www.w3.org/2005/Incubator/rdb2rdf/

approach described in [21] (*cf.* [19] for a feature-based comparison of R3M and other state-of-the-art RDB-to-RDF mapping languages). R3M is sufficient to cover use cases as described in [22] and, in general, to map normalized RDB schemata (*e.g.,* schemata generated by object-relational mappers such as Hibernate[13]).

In [23], we presented an example-driven definition of our RDB-to-RDF mapping approach. We now introduce a formal definition of our RDB-to-RDF mapping language R3M followed by examples in the RDF-based syntax. The rationale of the formal definition and the proofs is to show that our mapping language R3M is bidirectional and therefore not affected by the view update problem.

Definition 1. *In* OntoAccess*, a mapping is defined as an eight-tuple* $\mathcal{M} = \{R, A, L, C, P, \mathcal{T}, \mathcal{A}, \mathcal{L}\}$ *with:*

- $R = \{R_1, ..., R_l\}$ *the set of relations of the source database schema*
- $A = \{A_1, ..., A_l\}$ *the set of attribute sets with* $A_i = \{a_{i1}, ..., a_{im}\}$ *the set of attributes of relation* R_i
- $L = \{L_1, ..., L_k\} \subset R$ *the set of relations that represent N:M relationships with all* L_i *satisfying the condition:* $(\forall a_{ij} \in A_i : a_{ij} \in PK(L_i) \vee a_{ij} \in FK(L_i)) \wedge |FK(L_i)| = 2$ *with* $PK(X)$ *and* $FK(X)$ *being the sets of primary and foreign keys of relation X and* $|FK(X)|$ *being the number of foreign keys of relation X*
- C *the set of classes of the target ontology*
- P *the set of properties of the target ontology*
- $\mathcal{T} : R \backslash L \rightarrow C$ *an injective and surjective partial function mapping tables (without link tables) to ontology classes*
- $\mathcal{A} = \{\mathcal{A}_1, ..., \mathcal{A}_l\}$ *the set of injective and surjective partial functions* $\mathcal{A}_i : A_i \rightarrow P$ *mapping attributes of relation* $R_i \in R \backslash L$ *to ontology properties*
- $\mathcal{L} : L \rightarrow P$ *an injective and surjective partial function mapping link tables to ontology properties*

Lemma 1. *Mappings as defined in Definition 1 are bidirectional.*

Proof. The mapping functions $\mathcal{T}, \mathcal{A}_1, ..., \mathcal{A}_l, \mathcal{L}$ are defined as injective and surjective partial functions, hence there exist unique inverses $\mathcal{T}^{-1}, \mathcal{A}_1^{-1}, ..., \mathcal{A}_l^{-1}, \mathcal{L}^{-1}$ that map ontology terms to elements of the database schema. □

The functions $\mathcal{T}$, $\mathcal{A}_1$, ..., $\mathcal{A}_l$, and $\mathcal{L}$ are defined as partial because we do not require that all tables or attributes of a database schema are mapped to terms in the ontology. But defining these functions incautiously may result in invalid mappings. For example, if a foreign key attribute of a table is mapped but the referenced table is not. This results in an invalid mapping since the mapped attribute would be dangling, referencing resources that are not mapped to a table in the database schema. We therefore define the notion of a valid mapping.

Definition 2. *A mapping* $\mathcal{M}$ *is called* ***valid*** *if and only if it is defined according to Definition 1 and if and only if it satisfies the following two conditions:*

(i) $\forall a_{ij} \in A_i : a_{ij} \in NN(R_i) \Rightarrow a_{ij} \in dom\mathcal{A}_i$ *with* $NN(X)$ *being the set of attributes with a* not null *constraint of relation X and* $dom\mathcal{F}$ *the domain of function* $\mathcal{F}$*, i.e., all attributes of a mapped relation with a* not null *constraint must be mapped*

(ii) $\forall a_{ij} \in dom\mathcal{A}_i : a_{ij} \in FK(R_i) \Rightarrow ref(a_{ij}) \in dom\mathcal{T}$ *with* $FK(X)$ *being the set of foreign keys of a relation X,* $ref(Y) : \bigcup A_i \rightarrow R$ *being a function that returns the referenced relation for a foreign key attribute Y, and* $dom\mathcal{F}$ *the domain of function* $\mathcal{F}$*, i.e., if a foreign key attribute is mapped, the relation that it references must also be mapped*

We further define the notion of a complete mapping.

Definition 3. *A mapping* $\mathcal{M}$ *is called* ***complete*** *if and only if the mapping functions* $\mathcal{T}$*,* $\mathcal{A}_1$*, ...,* $\mathcal{A}_l$*,* $\mathcal{L}$ *are total, i.e., all relations and attributes of a source database schema are mapped to classes and properties in the target ontology.*

Lemma 2. *Complete mappings are valid.*

Proof. A complete mapping $\mathcal{M}$ has the properties:

(i) $\forall R_i \in R \backslash L : R_i \in dom\mathcal{T}$*, i.e.,* all relations (without link tables) are mapped to ontology classes

(ii) $\forall A_i \in A : \forall a_{ij} \in A_i : a_{ij} \in dom\mathcal{A}_i$*, i.e.,* all attributes are mapped to ontology properties

(iii) $\forall L_i \in L : L_i \in dom\mathcal{L}$*, i.e.,* all link tables are mapped to ontology properties

and therefore trivially satisfies conditions (i) and (ii) of Definition 2. □

For the remainder of the paper, we assume that mappings are always at least valid if not complete.

We now present selected examples in the RDF-based syntax of our mapping language R3M to further illustrate our mapping approach. The namespace prefixes used in the examples are defined as follows: `r3m` represents our mapping language ontology `http://ontoaccess.org/r3m#` while `ex` is used for the namespace `http://example.com/mapping/` of our example mapping. `foaf` and `dc` represent the namespaces of the well-known *Friend of a Friend*[14] and *Dublin Core*[15] projects. Listing 1a) depicts a *TableMap* representing the mapping of a database table to a class in the ontology. The set of all *TableMaps* in a mapping definition implements the mapping function $\mathcal{T}$ of Definition 1. A *TableMap* contains the name of the table (line 2) and the ontology class it is mapped to

[13] `http://hibernate.org/`

[14] urlhttp://xmlns.com/foaf/0.1/

[15] `http://purl.org/dc/elements/1.1/`

(line 3). The URI pattern (line 4, abbreviated) is used to generate the URIs for instances of this table based on values of table attributes that are specified between double percentage signs (*e.g., %%id%%* where *id* is the name of the primary key attribute). A *TableMap* further contains a list of *AttributeMap*s (lines 5 to 8).

Listing 1b) presents an example of an *AttributeMap* that maps a database attribute to a property in the ontology. The set of all *AttributeMap*s in a mapping definition implements the set of mapping functions $\mathcal{A}$ of Definition 1. An *AttributeMap* contains the name of the attribute in the database schema (line 11) and the ontology property it is mapped to (line 12). Additionally, an *AttributeMap* includes information about constraints defined on that attribute (*e.g.,* a *not null* constraint; line 13). Currently the constraints `r3m:PrimaryKey`, `r3m:ForeignKey`, `r3m:NotNull`, and `r3m:Default` are supported.

Listing 1c) shows a *LinkTableMap* representing the mapping of a link table to an ontology property. The set of all *LinkTableMap*s in a mapping definition implements the mapping function $\mathcal{L}$ of Definition 1. A *LinkTableMap* specifies the name of the link table in the database (line 16) and the property it is mapped to (line 17). A link table always contains two foreign key attributes that point to the tables of the N:M relationship. These attributes are represented as *AttributeMap*s (line 18 and 19; the definitions of those *AttributeMaps* are not shown in the example) that provide the names of the attributes, the foreign key references to the tables, and the direction of the relationship (from subject to object).

Listing 1: Example mappings

```
1  a) ex:author a r3m:TableMap;
2       r3m:hasTableName   "author";
3       r3m:mapsToClass    foaf:Person;
4       r3m:uriPattern     "http://.../author%%id%%";
5       r3m:hasAttribute   ex:author_id,
6                          ex:author_email,
7                          ex:author_firstname,
8                          ex:author_lastname.

10 b) ex:author_email a r3m:AttributeMap;
11      r3m:hasAttributeName        "email";
12      r3m:mapsToObjectProperty    foaf:mbox;
13      r3m:hasConstraint           [ a r3m:NotNull ].

15 c) ex:publication_author a r3m:LinkTableMap;
16      r3m:hasTableName            "publication_author";
17      r3m:mapsToObjectProperty    dc:creator;
18      r3m:hasSubjectAttribute     ex:pa_publication;
19      r3m:hasObjectAttribute      ex:pa_author.
```

4. OntoAccess Platform Architecture and Implementation

The goal of OntoAccess is to provide a platform for RDF-based read and write access to data stored in existing RDBs. It supports a broad number of data access interfaces and is extensible for future development in data access approaches. The main idea of the OntoAccess platform is to encapsulate the RDB-to-RDF translation logic into basic core operations and thus avoid repeated implementation of the translation functionality. This simplifies the development of additional data access interfaces and increases the flexibility of the platform.

On the basis of the well-known *CRUD*[16] operations, we define three basic operations for Semantic Web data access: (1) querying for a single triple pattern; (2) adding a set of triples to the data; and (3) removing a set of triples from the data. In principle, it is possible to implement any data access based on these core operations, ignoring for the moment any concerns about performance and atomicity of requests.

Figure 1 depicts the architecture of the OntoAccess platform. It is split into two layers. The lower part, called core layer, is responsible for the actual RDB-to-RDF translation as well as the interaction with the database system. The upper part, called interface layer, exposes the functionality of the OntoAccess core to the individual data access approaches. The interfaces are either accessed directly by applications or over the network via a service endpoint. We explain both layers in more detail in the following sections.

4.1. Core Layer

The core layer is composed of four modules, namely *Uni Core*, *Query Core*, *Update Core*, and *TA Manager*. It implements the three basic operations described above for RDB-to-RDF translation and it is responsible for the interactions with the database system.

The *Uni Core* module provides the API of the core layer to the data access interface layer. It acts as a controller of the other three modules to manage the correct execution of requests including their encapsulation in database transactions. This API basically consists of two methods, one for query requests and one for update requests. The query method is a simple wrapper for the query method of the *Query Core* described below. The update method, on the other hand, takes as parameters one set of insert requests and one set of delete requests to execute all of them in the scope of a single database transaction (*e.g.,* for a SPARQL/Update request). Data access interfaces are required to collect requests that belong to a single transaction themselves and submit them all at once. This has the advantages that data access interfaces are relieved from managing transactions and the runtime of the transactions can be kept as short as possible. The *Uni Core* further isolates the *Query Core* and the *Update Core* from the database as it is responsible for collecting translated requests and for passing them to the database system.

The *TA Manager* module is responsible for database transaction management. It is used for starting, committing, and rolling back transactions based on instructions it receives from the *Uni Core* module.

The main parts of the core layer are the *Query Core* and *Update Core* modules that implement the RDB-to-RDF translation logic. In the following, we present them in more detail.

[16] **C**reate, **R**ead, **U**pdate, **D**elete

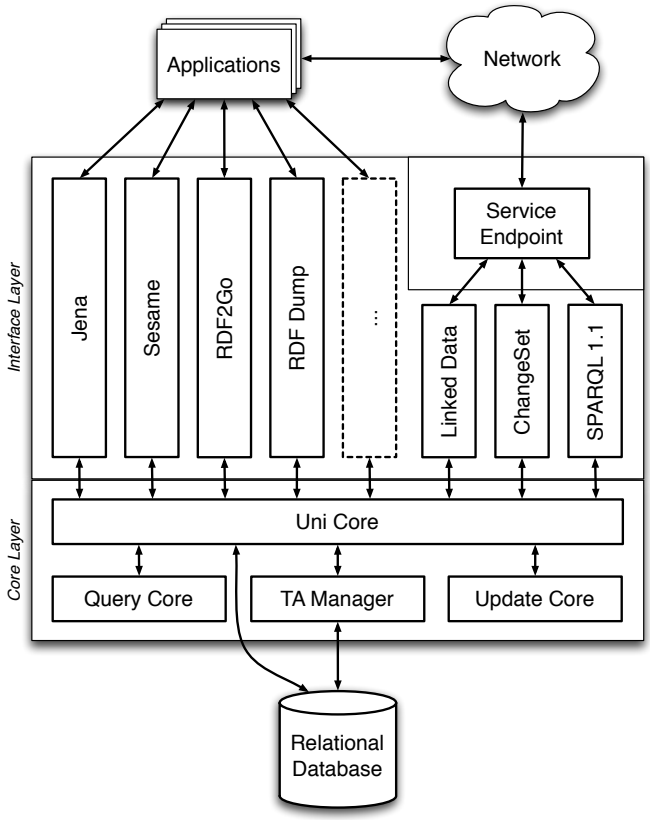

Figure 1: Platform Architecture

4.1.1. Query Core

The *Query Core* implements the basic operation of querying for an arbitrary triple pattern. Based on a given mapping, it translates the pattern to a single or multiple SQL queries depending on the type of pattern. For instance, a pattern asking for the object of a given subject and predicate will generate just a single SQL query whereas a pattern with variable subject and predicate but given object will result in multiple SQL queries as this object value could appear in multiple tables (and attributes) of the database.

Algorithms 1–4 illustrate this translation of triple patterns to SQL queries. First, we differentiate triple patterns that feature a concrete subject and such with a variable as subject (Algorithm 1). If the subject is concrete (*i.e.,* a resource URI) we use it to identify the table the pattern matches (line 2). On the other hand, a variable subject means that we potentially have to generate a SQL query for each mapped table (lines 5–7). Next, we check if the predicate of the pattern is concrete or a variable (Algorithm 2). A concrete predicate is translated to the matching attribute (line 2) while in the case of a variable predicate we can not know which attribute will match a given (object) value. We therefore have to incorporate all mapped attributes of a given table into the query (lines 5–6). Next, we differentiate concrete and variable objects (Algorithm 3). The database value is extracted from a concrete value according to the mapping definition (*e.g.,* extracting part of an URI). Otherwise, if the object is a variable we mark the value as *null* (line 4). Finally, we assemble the SQL query (line 6; Algorithm 4). The primary key of the affected table is added to the list of projected variables (*i.e.,* the *select* clause; line 1) and the table itself is added to the *from* clause. Then, we iterate over the attributes to add conditions to the *where* clause (lines 3–8). If the value is *null*, we add the attribute to the *select* list (lines 4–6) and a condition that this attribute must not be *null* to the *where* clause. Otherwise, a condition is added that states that the attribute must match the value. In any case, multiple conditions are combined using the *or* operator (line 7). Last, the query is built from the *select*, *from*, and *where* parts and returned (line 9).

Algorithm 1 translatePattern(subject, predicate, object)

```
1: if subject.isVariable() is false then
2:     table ← identifyTable(subject)
3:     queries ← translateTable(table, predicate, object)
4: else
5:     for all table in getTables() do
6:         queries ← translateTable(table, predicate, object)
7:     end for
8: end if
9: return queries
```

After the translation, the resulting SQL queries are encapsulated in a `TripleIterator` that implements the

Algorithm 2 translateTable(table, predicate, object)

```
1: if predicate.isVariable() is false then
2:     attribute ← identifyAttribute(table, predicate)
3:     queries ← translateAttribute(table, attribute, object)
4: else
5:     attributes ← getAttributes(table)
6:     queries ← translateAttribute(table, attributes, object)
7: end if
8: return queries
```

Algorithm 3 translateAttributes(table, attributes, object)

```
1: if object.isVariable() is false then
2:     value ← extractValue(object)
3: else
4:     value ← NULL
5: end if
6: queries ← assembleQuery(table, attributes, value)
7: return queries
```

`java.util.Iterator` interface to provide a standard means for iterating over the results of a triple pattern query. The triples are generated on demand from the results of the SQL queries, which are evaluated sequentially. At any moment there is only one active SQL query, this means in the beginning the first SQL query is evaluated and its result are used for generating the result triples. Only after this SQL `ResultSet` is exhausted the next SQL query is evaluated, and so on. This has two major advantages. First, it reduces the memory consumption as at any time only one SQL `ResultSet` object must be held in main memory, the remaining queries are stored as strings. Second, if the caller is not interested in all results (*e.g.,* only the first twenty result triples are of interest) it is possible that only a subset of the generated SQL queries need to be evaluated. In that case, this approach can also have a positive effect on performance by exploiting the given partitioning of the data into tables.

The `TripleIterator` is implemented as a look-ahead iterator, *i.e.,* it always generates the next triple in advance and caches it until `next()` is called. Then the cached triple is returned and the next one is generated and cached. This approach was taken due to the differences in the APIs of SQL `ResultSet` and Java `Iterator`. Iterators have a `hasNext()` method to check if there are any further results, the SQL API does not offer such a method. Instead, a boolean value is returned after moving the cursor to the next result that indicates if there are additional results. Look-ahead iterators allow to bridge this API gap elegantly and with good performance by avoiding unnecessary movement of the result cursor.

Algorithm 4 assembleQuery(table, attributes, value)

```
1: select ← table.getPK()
2: from ← table
3: for all attribute in attributes do
4:     if value == NULL then
5:         select ← attribute
6:     end if
7:     where ←(OR) getCondition(attribute, value)
8: end for
9: return buildQuery(select, from, where)
```

4.1.2. Update Core

The *Update Core* implements the remaining two basic operations of adding and removing sets of triples. In either case, the triples are translated to (typically multiple) SQL DML statements. The translation is performed according to a generalized version of the algorithm presented in [23] for translating SPARQL/Update *insert data* and *delete data* operations. We briefly recapitulate the basic idea of the algorithm and refer the reader to [23] for more details. The translation for adding and removing triples is basically the same, the difference is only in the generated SQL statements (insert vs. delete). First, the triples are grouped into so-called subject groups based on equal subjects (*i.e.,* these triples have the same subject and therefore affect the same record in the database). This allows us to translate each such group of triples individually. Second, the affected table is identified via the subject URI. In a third step, the mapping is used to check if the submitted triples satisfy certain integrity constraints of the database schema. Step four generates the SQL statement for adding or deleting this group of triples based on the mapping definition. The predicate of each triple is translated to an attribute of the affected table. The object is used as the data value either directly or if it is a resource URI by matching it against the template in the mapping and extracting the predefined substring. Each subject group is processed that way and the resulting SQL statements are collected. Finally, the statements are executed within a single database transaction to ensure the atomicity of the original request.

4.2. Data Access Interface Layer

The data access interface layer is responsible for establishing the connection between the core layer of OntoAccess and Semantic Web applications. This is realized with an extensible set of data access interfaces that are exposed to the applications either directly (*e.g.,* the Jena interface) or indirectly via the service endpoint described in the next section (*e.g.,* the Linked Data interface). The job of the individual interfaces is to translate the interface-specific operations to the basic OntoAccess operations and possible results back into the interface-specific format. The idea is that such interfaces are very lightweight and therefore simple to develop as the main translation work is performed in the core layer. Currently, the OntoAccess platform implements data access interfaces for multiple Semantic Web Frameworks (Jena, Sesame, RDF2Go), RDF Dump (for dumping all data as RDF to a file), Linked Data, ChangeSet, and a subset of SPARQL 1.1 that includes SPARQL/Update as proposed in [6].

As an example, we present the Jena data access interface in more detail. Jena [24] uses a two-layer API to interact with RDF data. It is composed of the *Model* and the *Graph* APIs.

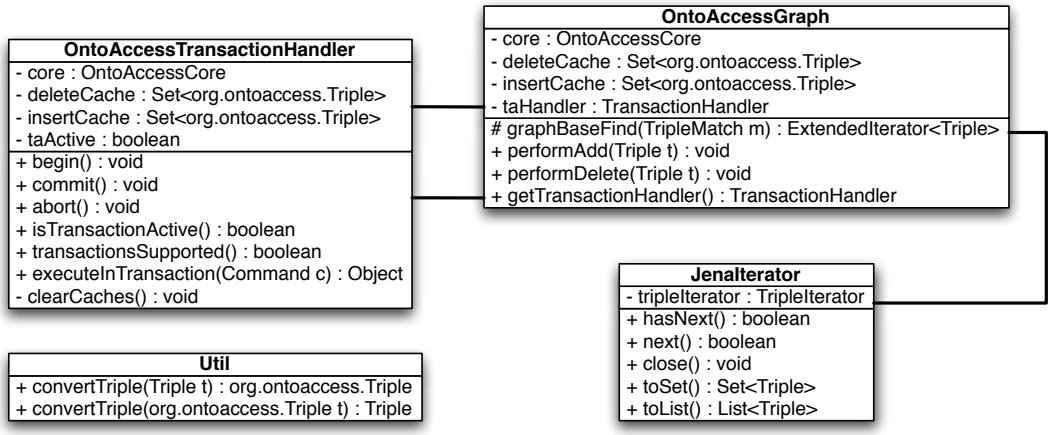

Figure 2: Jena Interface UML Class Diagram

The *Graph* API represents the lower layer and is responsible for retrieving and storing triples. It provides methods to add, delete, and find triples. The *Model* API is the layer facing the user and provides convenience methods for working with RDF data. These APIs are designed to be extended, *i.e.,* to add support for additional triple storage schemes. Jena itself provides multiple implementation for storing triples in memory or on disk. For the Jena interface of OntoAccess, we provide an implementation of the *Graph* API. Extending the *Model* API was not necessary as Jena contains a bridge to create a standard model from any implementation of the *Graph* API. Figure 2 depicts the UML class diagram of our Jena interface. It provides an implementation that interacts with the core layer of OntoAccess. The `OntoAccessGraph` class represents the main API class that contains the methods for adding, deleting, and finding triples. It maps them to the basic operations of OntoAccess. Further, the `JenaIterator` class implements the iterator interface that is returned from a call of the find method of the `OntoAccessGraph` class. It is basically a wrapper of our `TripleIterator` described in Section 4.1.1. Jena provides support for transactions by introducing dedicated transaction handler classes. Our `OntoAccessTransactionHandler` implements this by collecting the triples intended to be added to or deleted from the data and forwarding them to the OntoAccess core to commit the transaction. The `Util` class contains convenience methods for converting between the triple representations of Jena and OntoAccess.

4.3. Service Endpoint

The *Service Endpoint* of OntoAccess is a server application that exposes data access interfaces as services to the network. Data access interfaces can be registered at the endpoint by providing the name of the implementing handler class and the HTTP request target (*e.g.,* `http://example.org/sparql?query=...` where *sparql* is the HTTP request target for the SPARQL service). If a request matches the target, its query string is forwarded to the respective data access interface for processing. Request strings that do not match any defined target are per default interpreted as Linked Data requests if the corresponding data access interface is installed or else as an error.

The service endpoint is implemented based on Java servlet technology[17] and the Jetty embedded Web server.[18]

5. Semantic Feedback Protocol

The conceptual gap between the relational model and RDF has a greater effect on translating RDF-based write requests to the database level than on read-only queries. If a query uses ontology terms (or instances) that cannot be mapped to the database schema (or data), the query can be processed without error but simply returns no results. However, if a write request contains such non-mappable ontology terms it cannot be fully processed and results in an error. Further, a write request can be underspecified *w.r.t.* the constraints defined in the database schema (*e.g., not null* constraints). In such situations it is possible to simply reject the whole request or ignore the parts that cannot be mapped. In both approaches, the client does not know why the request was not fully processed, especially if the client is unaware of the RDB-based foundation of the data storage system. Having said that, it cannot be expected (neither is it desirable) that all clients know about the specifics of the RDB schema if their usage is limited to the RDF-based data access interfaces. We therefore propose a semantic feedback protocol to alleviate this problem. The main idea of this feedback approach is to detect requests that are invalid *w.r.t.* the RDB schema already during request translation and in a second step provide feedback to the client in a semantic format such as RDF. In this way, there is no conceptual break between request and (error) response.

In OntoAccess, the semantic feedback protocol is implemented as a cross-layer feature. The detection of invalid write

[17] `http://java.sun.com/products/servlet/`
[18] `http://jetty.codehaus.org/jetty/`

requests is performed in the *Update Core* where incoming requests are analyzed and translated. Requests are always fully analyzed to identify and report all invalid elements. The resulting feedback is stored in an internal format in the *Update Core* and it is exposed in a feedback interface to the *Data Access Interface Layer* and the *Service Endpoint*. The data access interfaces can implement the processing according to their needs (*e.g.,* the Jena interface could convert the feedback to Java exceptions). The *Service Endpoint* converts the feedback to an RDF-based format adhering to our semantic feedback ontology described later in this section. The feedback is published in this RDF-based format at the HTTP request target *feedback* of the *Service Endpoint* (*i.e.,* `http://example.org/feedback`) and can be retrieved with a simple *GET* request on that URL.

The remainder of this sections introduces the different types of feedback supported by our approach and our semantic feedback ontology. At last, we present a concrete example of feedback in the RDF-based format implemented in the *Service Endpoint*.

5.1. Feedback Types

We identified five causes for invalid write requests that can be detected during request translation. All of them arise from the conceptual gap between the relational model and RDF. We defined five corresponding feedback types which we present in detail.

5.1.1. MissingTriple

A *MissingTriple* feedback is generated if a request lacks data for a mandatory attribute in the RDB, *i.e.,* an attribute with a *not null* constraint. There are two cases that can lead to such a feedback. First, if data should be inserted that would create a new record in the database but the data of at least one mandatory attribute is missing in the request. Second, if data should be delete that corresponds to a subset of an existing record and includes data of a mandatory attribute. Both cases would lead to a database record with mandatory attributes set to *null* – a violation of constraints that would be prevented by the database management system. Requests that generate a *MissingTriple* feedback are always aborted.

5.1.2. UnknownSubject

An *UnknownSubject* feedback is generated if a request contains an RDF triple with a subject that cannot be mapped to a table of the RDB schema and can therefore not be stored. This type of feedback is exclusive to insert requests, because deleting a non-existing triple results in no operation on the data and can be silently ignored according to [25]. It is a matter of configuration if requests that generate an *UnknownSubject* feedback should be aborted or continued without the affected triples. However, the feedback is always generated for information purposes.

5.1.3. UnknownTriple

An *UnknownTriple* feedback is generated if a request contains an RDF triple with a predicate that cannot be mapped to an attribute of the RDB schema and can therefore not be stored. In this feedback case the subject of the triple can be mapped to a table of the RDB schema or else it is classified as a *UnknownSubject* feedback as mentioned above. This type of feedback is exclusive to insert requests, because deleting a non-existing triple results in no operation on the data and can be silently ignored according to [25]. It is a matter of configuration if requests that generate an *UnknownTriples* feedback should be aborted or continued without the affected triples. However, the feedback is always generated for information purposes.

5.1.4. NonMatchingTriple

A *NonMatchingTriple* feedback is generated if a request contains an RDF triple that can be mapped to the RDB schema but the value of the affected database record is already set and is different from the value of the object in the triple. This leads to an error because RDBs do not support storing multiple values for a single attribute. If the object value and the database value match it is not an error as inserting an already existing triple is silently ignored according to [25]. In this feedback case the subject and predicate can be mapped to the RDB schema or else it would be classified as either a *UnknownSubject* or an *UnknownTriple* feedback as mentioned above. This type of feedback is exclusive to insert requests, because deleting a non-existing triple can be silently ignored [25]. It is a matter of configuration if requests that generate an *NonMatchingTriple* feedback should be aborted or continued without the affected triples. However, the feedback is always generated for information purposes.

5.1.5. DefaultTripleAdded

A *DefaultTripleAdded* feedback is generated if a request lacks data for an attribute in the RDB that has a default value defined. Default values in RDBs are silently added to new records if they are not provided by the client. The *DefaultTripleAdded* feedback informs the client about the additional data that was generated by translating it to an RDF triple. There are two cases that can lead to such a feedback. First, if data is inserted that would create a new record in the database but the data of at least one attribute with default value is not explicitly given in the request. Second, if data is deleted that includes the data of an attribute with default value. In this case the original data is deleted but the data is restored with the default value by the RDB system. The feedback is always generated for information purposes. Requests are never aborted because of a *DefaultTripleAdded* feedback.

An invalid write request may generate multiple feedback instances of the same or different types. The semantic feedback ontology in the next section is used to combine all feedback in a single feedback message for the client.

5.2. Semantic Feedback Ontology

The semantic feedback that is collected during the translation of write requests is provided to the client in an RDF-based format. This format is defined by our semantic feedback ontology described in this section. Table 1 presents an overview of the ontology with a list of all classes and short descriptions. The

descriptions also include the most important ontology properties that are used with the respective class (*i.e.,* have that class as their *rdfs:domain*).

5.2.1. Examples

Listing 2 shows an example feedback document in the RDF-based format. It contains three individual feedback instances that we will explained in detail.

The first part of the feedback document is the definition of namespace prefixes as required by the Turtle RDF serialization [27] (lines 1 to 5). Then, the main feedback message is listed (lines 7 to 11) that consists of the individual feedback instances (lines 8 to 10) and the date this request was processed (line 11). The rest of the document contains the three feedback instances. First, *fb:FB1* describes a *MissingTriple* feedback (line 13). As it is not possible to execute a request where mandatory triples are missing, the request was therefore aborted (line 14) leading to a severity of *fb:Fatal* (line 15). It can further be seen in the feedback that the request was an insert request (line 16). A *MissingTriple* feedback always includes information about what kind of triple was missing. The expected subject (line 17) is taken from the original request, the expected predicate (line 18) and the expected datatype of the object (line 19) are extracted from the mapping definition. At last, human readable descriptions of the feedback are given (lines 20 and 21; abbreviated). The second feedback is of type *DefaultTripleAdded* (line 23). This feedback would not result in the request being aborted, it is for information purposes (line 25) and could be ignored (line 23). The triple representation of the automatically added data is provided in the feedback as well using RDF reification [26] (lines 27 to 29). The final feedback for this request is a *NonMatchingTriple* feedback (line 34). It shows that it also leads to the the request being aborted (line 35) with a severity of *fb:Error* (line 36). This severity means that it would be possible to execute the request but this specific triple would not be stored. Instead the existing triple would remain valid. It is a matter of configuration if requests are aborted on feedback of severity *fb:Error*. The feedback contains the submitted triple (lines 38 to 40) as well as the object that exists already in the database (line 41).

Note that although each feedback instance has a *fb:action* property, a request is aborted if at least one of the feedback instances sets this property to *fb:Abort*.

6. Evaluation

The evaluation of our approach is split into three parts. First, in Section 6.1 we demonstrate by example how simple it is to extend the OntoAccess platform with data access interfaces. Second, in Section 6.2 we compare the performance of OntoAccess with D2R, Jena SDB,[19] a RDB-backed triple store, and Jena TDB,[20] a native RDF triple store. The benchmark experiment is based on basic Jena API calls for querying, adding, and deleting triples since all evaluated approaches provide support for the Jena framework. Lastly, we summarize a case study we performed with OntoAccess in the domain of software analysis platforms.

6.1. Extensibility

In this paper, we claim that the OntoAccess platform provides simple extensibility to meet the requirement for developing additional data access interfaces. We will demonstrate this simple extensibility with three example implementations of data access interfaces, namely for *Jena*, *Linked Data*, and *ChangeSet*.

6.1.1. Jena

The Jena interface is an example that requires both read and write data access. Its implementation was already described in detail in Section 4.2, therefore we just add that it is one of the more complex interfaces and that it was implemented in about 300 lines of Java code.

6.1.2. Linked Data

The Linked Data interface is an example for read-only data access. It is one of the simpler interfaces and was implemented in about 100 lines of Java code. It provides support for linked data typed queries via the *Service Endpoint*. It takes an URI as input and returns all triples that have this URI as their subject. For that, it constructs a triple pattern with the given URI as subject and variables as predicate and object. This pattern is forwarded to the OntoAccess core for translation and evaluation. The resulting `TripleIterator` is wrapped in a `HtmlPartIterator` that emits the individual triples in HTML markup so that the *Service Endpoint* can directly stream the result page to the caller.

6.1.3. ChangeSet

The ChangeSet interface is an example for write-only data access. It is accessible via the *Service Endpoint* and it implements the ChangeSet protocol.[21] Its implementation was realized in about 200 lines of Java code and consists of the actual interface, the ChangeSet parser, and the implementation of the protocol. ChangeSet requests are RDF graphs adhering to the ChangeSet ontology.[22] They contain a so-called subject of change and two sets of matching triples. One triple set is meant for removal and the other for addition. Our ChangeSet interface implementation uses the Jena framework to parse the request and to extract the subject of change as well as the two triple sets. It then converts the parsed triples to the triple representation of OntoAccess and passes them via the *Uni Core* to the *Update Core* for addition and removal. To ensure the atomicity of a ChangeSet request, the addition and removal of the triples are executed within a single database transaction.

This brief description of implemented data access interfaces demonstrates how simple it is to develop such interfaces. It

[19] `http://openjena.org/SDB/`
[20] `http://openjena.org/TDB/`
[21] `http://n2.talis.com/wiki/Changeset_Protocol`
[22] `http://purl.org/vocab/changeset`

Listing 2: Semantic Feedback Example

```
@prefix  fb:    <http://ontoaccess.org/feedback/> .
@prefix  rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix  rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix  xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix  dc:    <http://purl.org/dc/elements/1.1/> .


fb:FeedbackMessage1  a  fb:FeedbackMessage;
    fb:hasFeedback  fb:FB1,
                    fb:FB2,
                    fb:FB3;
    dc:date         "2011-10-05T13:37:21" .


fb:FB1  a  fb:MissingTriple;
    fb:action                   fb:Abort;
    fb:level                    fb:Fatal;
    fb:source                   fb:Insert;
    fb:expectedSubject          http://localhost:4040/uuc/Student3002;
    fb:expectedPredicate        http://xmlns.com/foaf/0.1/mbox;
    fb:expectedObjectDatatype   xsd:anyURI;
    rdfs:label                  "MissingTriple";
    rdfs:comment                "A mandatory triple is missing ...".


fb:FB2  a  fb:DefaultTripleAdded, rdf:Statement;
    fb:action       fb:Ignore;
    fb:level        fb:Info;
    fb:source       fb:Insert;
    rdf:subject     http://localhost:4040/uuc/Student3002;
    rdf:predicate   http://ontoaccess.org/edu#grade;
    rdf:object      "1";
    rdfs:label      "DefaultTripleAdded";
    rdfs:comment    "A default triple was added ..." .


fb:FB3  a  fb:NonMatchingTriple, rdf:Statement;
    fb:action           fb:Abort;
    fb:level            fb:Error;
    fb:source           fb:Insert;
    rdf:subject         http://localhost:4040/uuc/Student1001;
    rdf:predicate       http://xmlns.com/foaf/0.1/firstName;
    rdf:object          "John";
    fb:expectedObject   "Bob";
    rdfs:label          "NonMatchingTriple";
    rdfs:comment        "A triple was detected ..." .
```

Table 1: Semantic Feedback Ontology – Overview

Class	Description (including class specific properties)
FeedbackMessage	The collection of all feedback generated by a single write request. It lists all feedback instances (→ *hasFeedback*) that can be of any subclass of *FeedbackType*. It further contains the date (→ *dc:date*) this request was processed.
FeedbackType	The superclass for all types of feedback. It lists information about the severity of the feedback (→ *level: info, warn, error, fatal*), if it was generated during an *insert* or *delete* request (→ *source*), and if the request was *abort*ed or the source of the feedback was *ignore*d (→ *action*). The feedback further contains human readable descriptions of the feedback type (→ *rdfs:label* and → *rdfs:comment*). If the feedback affects a single triple, it is included in the feedback using RDF reification [26], *i.e.,* its subject (→ *rdf:subject*), predicate (→ *rdf:predicate*), and object (→ *rdf:object*) are listed.
MissingTriple	A subclass of *FeedbackType* that uses additional properties of the ontology. It lists information about the expected subject (→ *expectedSubject*), the expected predicate (→ *expectedPredicate*), and the expected datatype of the object value (→ *expectedObjectDatatype*) that the missing triple should be composed of.
UnknownSubject	A subclass of *FeedbackType* that uses an additional property of the ontology. It contains a list of triples (→ *triples*) that use the unknown subject.
UnknownTriple	A subclass of *FeedbackType* that uses no additional properties of the ontology.
NonMatchingTriple	A subclass of *FeedbackType* that uses no additional properties of the ontology.
DefaultTripleAdded	A subclass of *FeedbackType* that uses no additional properties of the ontology.
FeedbackAction	Indicates if the feedback resulted in aborting the request or not. The ontology defines two instances of this class named *Abort* and *Ignore*.
FeedbackLevel	Describes the severity of the feedback. The ontology defines four instances of this class named *Info*, *Warn*, *Error*, and *Fatal*. Depending on this severity level a client may react differently to the feedback. For instance, *Fatal* means the request was not executed at all while *Info* means it was successful, but some feedback was generated for information purposes.
FeedbackSource	Indicates if the feedback was generated during an insert or a delete request. The ontology defines two instances of this class named *Insert* and *Delete*.

shows how few lines of code it requires compared to the translation logic in the core layer which consists of more than 8000 lines of Java code.

6.2. Performance

The layered architecture of the OntoAccess platform may suggest a disadvantage in performance compared to other RDB-to-RDF mapping approaches. In this section, we compare the performance of OntoAccess with D2R, Jena SDB, a RDB-backed triple store, and Jena TDB, a native RDF triple store. The benchmark experiment is based on basic Jena API calls for querying, adding, and deleting triples since all evaluated approaches provide support for the Jena framework. We show that OntoAccess delivers comparable or better performance than D2R and Jena SDB.

6.2.1. Experimental Setup

The experiment was conducted on a Apple MacBook Pro notebook with a 2.33GHz Intel Core 2 Duo dual core CPU, 3GB of DDR2 667MHz RAM, a 320GB SATA HDD with 7200rpm running Mac OS X 10.6.3 as the operating system. As Java runtime we used version 1.6.0_17 provided with Mac OS X. As database system, MySQL version 5.1.45 was used for all systems with the default settings except `innodb_buffer_pool_size` which was increased to 64MB via the *my.cnf* configuration file. The benchmark was run with a heap space allocation of 1024MB (*-Xmx1024m*).

We reused the dataset from the Berlin SPARQL Benchmark (BSBM) [28] in sizes equivalent to one million, ten millions, and hundred millions of triples. The datasets were generated with the BSBM data generator as described in the BSBM specification [29]. The mapping for D2R was reused from a prior benchmark experiment conducted by the BSBM team. It is publicly available from their benchmark results website.[23] The mapping for OntoAccess was specifically developed for this evaluation. It is a complete mapping according to Definition 3.

The experiment consists of two parts, a query part and an update part. The query part tests the evaluation performance of single triple pattern queries. There exist eight such patterns including the one containing no variables (*i.e.,* a concrete triple) and the one containing only variables (*i.e.,* resulting in a dump of the database). The times reported below include the translation and evaluation of the queries as well as the retrieval of at

[23] `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/results/`

most fifty result triples. The update part tests the performance of adding and deleting **A** a single triple, **B** a set of eight triples that affect a single table in the RDB, and **C** a set of thirteen triples that affect multiple tables. The results presented below were measured as the average of five benchmark runs after two warmup runs. The query part was executed for all systems under test (SUTs), the update part for OntoAccess, Jena SDB, and Jena TDB as D2R lacks support for data updates.

We used the following releases of the SUTs. OntoAccess in version 0.3,[24] D2R in version 0.7, Jena SDB in version 1.3.0 with the index-based database layout, and Jena TDB in version 0.8.4. All SUTs were used with default settings.

6.2.2. Results

Table 1 depicts the results of the query benchmark for datasets equivalent to one, ten, and hundred millions of triples. The first column names the approach and the dataset size. The remaining eight columns show the benchmark result times in milliseconds for each of the eight possible triple patterns. The triple patterns are depicted as a combination of the letters *s*, *p*, *o* that represent concrete subjects, predicates, objects and the question mark *?* that represents variables. For instance, the triple pattern *(s p ?)* represents a pattern with concrete subject and predicate but variable object.

The results show that for triple patterns with known subject, OntoAccess performs comparable to D2R and better than Jena SDB and Jena TDB. For triple patterns with unknown subject the results are mixed. Compared to D2R, OntoAccess performs comparable for the patterns with known predicate and in general better for the patterns with unknown predicate. The performance of Jena TDB for triple patterns with unknown subject is similar to patterns with known subject and therefore better than OntoAccess. Jena SDB performs better than OntoAccess for the patterns *(? p o)* and *(? ? o)* but worse for the other two. The performance of evaluating patterns with known predicate could be improved in OntoAccess and D2R if a database index is created on the attribute that is mapped to the property *p*. Tests showed that this reduces evaluation times to the levels of triple patterns with known subject. Also note that Jena SDB is only able to evaluate the *(? ? ?)* pattern in the one million triple dataset. It crashes with a `java.lang.OutOfMemoryError` error in larger datasets even if the heap memory allocation is doubled to 2048MB.

Table 2 depicts the results of the update benchmark for datasets equivalent to one, ten, and hundred millions of triples. The first column names again the approach and dataset size. The remaining six columns show the benchmark result times in milliseconds for adding and removing the three different triple sets. **A** represents the single triple, **B** the set of eight triples affecting a single table in the RDB, and **C** the set of thirteen triples affecting multiple tables. We report results for OntoAccess, Jena SDB, and Jena TDB as D2R is limited to read-only queries.

The results show that OntoAccess performs better than Jena SDB in adding and removing triples irrespective of the triple set or dataset size. The performance difference is especially striking in the removal of triples. A closer examination revealed that Jena SDB translates the removing of each individual triple to a SQL statement that needs to perform multiple joins on large tables. OntoAccess, on the other hand, translates the removing of triples to a single, join-less SQL statement for each affected table. Compared to Jena TDB OntoAccess performs better on the two larger data sets and the performance difference increases with the number of triples to add or remove.

6.3. Case Study

In [30], we presented a case study on how OntoAccess can be used to facilitate the transition from legacy systems to Semantic Web-enabled applications in practice. The case study showed how we successfully used OntoAccess to advance our Eclipse-based software evolution analysis framework Evolizer [31] to Sofas [32], a service-oriented, distributed, and collaborative software analysis platform. To motivate our case study, we present use cases that require interoperability between Evolizer and Sofas. These use cases need a bidirectional data exchange, *i.e.,* from Evolizer to Sofas and vice versa. First, Evolizer contains data about the software life-cycle of hundreds of software systems. Re-importing this vast amount of data in Sofas from version control and bug tracking systems would take months, and some of these repositories might not even be available online anymore. Therefore, RDF-based *read access* to the Evolizer database is needed. Second, Evolizer implements importers to import source code and history data from centralized version control systems, such as CVS and SVN. Lately decentralized version control systems, such as Git or Mercurial, gained popularity. Therefore, respective import services were developed for the Sofas platform. The data produced by these importer services is modeled in RDF. It would also be valuable to Evolizer because existing tools could be used to leverage it. This, however, requires RDF-based *write access* to the Evolizer database. Lastly, Sofas implements an extensible framework to compute software metrics on the data. Again, this data is modeled in RDF, but matching relations are available in the Evolizer database schema. RDF-based *write access* to the RDB is needed to make the metrics data available to Evolizer. These use cases indicate that, for making a bridge between Evolizer and Sofas, a RDB-to-RDF mapper such as OntoAccess is needed that provides RDF-based read and write access to RDBs.

7. Limitations

The expressivity of our RDB-to-RDF mapping language R3M is lower than some of the existing, read-only mapping languages (*cf.* [19]). The reason is the additional requirement of enabling RDF-based write access to the RDB while avoiding the view update problem. This difference in expressivity is relevant if the similarity between the RDB schema and the target ontology is low. However, we showed in [30] that R3M can be applied on real world application scenarios where a certain similarity between the RDB schema and the target ontology is given.

[24] available for download at `http://ontoaccess.org/`

Table 2: Result times for query benchmark [ms]

1M	(s p o)	(s p ?)	(s ? o)	(s ? ?)	(? p o)	(? p ?)	(? ? o)	(? ? ?)
OntoAccess	1	1	2	2	5	8	724	39
D2R	2	2	7	5	8	9	267	110
JenaSDB	16	11	6	13	5	71	4	36 427
JenaTDB	12	9	13	16	17	23	13	2
10M	**(s p o)**	**(s p ?)**	**(s ? o)**	**(s ? ?)**	**(? p o)**	**(? p ?)**	**(? ? o)**	**(? ? ?)**
OntoAccess	1	1	3	4	203	224	216	489
D2R	2	2	7	6	202	235	1 051	1 369
JenaSDB	34	21	26	33	25	520	20	–*
JenaTDB	24	24	31	28	93	10	87	2
100M	**(s p o)**	**(s p ?)**	**(s ? o)**	**(s ? ?)**	**(? p o)**	**(? p ?)**	**(? ? o)**	**(? ? ?)**
OntoAccess	2	1	3	4	1 962	2 222	1 952	5 130
D2R	2	2	8	7	1 998	2 341	3 650	13 725
JenaSDB	42	30	45	60	70	5 320	100	–*
JenaTDB	38	32	53	43	316	10	99	3

* *crashed with a* `java.lang.OutOfMemoryError`

Table 3: Result times for update benchmark [ms]

1M	add A	remove A	add B	remove B	add C	remove C
OntoAccess	4	3	5	4	6	8
JenaSDB	11	1 311	42	10 328	60	16 776
JenaTDB	2	3	10	8	12	12
10M	**add A**	**remove A**	**add B**	**remove B**	**add C**	**remove C**
OntoAccess	5	3	5	5	6	8
JenaSDB	11	13 601	44	111 150	78	180 532
JenaTDB	9	4	47	13	98	17
100M	**add A**	**remove A**	**add B**	**remove B**	**add C**	**remove C**
OntoAccess	4	3	4	5	5	7
JenaSDB	30	329 184	198	2 676 356	238	4 356 603
JenaTDB	15	3	77	13	134	16

Note: D2R is missing from this table as it is limited to read-only queries

OntoAccess is currently limited to a single RDB as a data source. It does not incorporate any query or update federation. However, the explicit information provided in the mapping about what kind of data (*i.e.,* classes and properties of an ontology) are stored in each data source could be leveraged to add federation support.

In this paper, we introduced a semantic feedback protocol to bridge the conceptual gap between the relational model and RDF in case of invalid write requests. We provided a proof-of-concept implementation, but a thorough evaluation is needed to show the usability of this feedback approach.

8. Conclusion

In this paper, we presented OntoAccess as an extensible platform for RDF-based read and write access to data stored in existing RDBs. We discussed that there are many different data access approaches in current Semantic Web applications and that a platform-based approach is needed to avoid repeated implementation effort in RDB-to-RDF translation. We identified three basic operations that such a platform has to provide in its core implementation, namely (1) querying for a single triple pattern, (2) adding triples, and (3) removing triples. These basic operations are implemented in the core layer of OntoAccess and we discussed that this architectural decision enables the simple implementation of various data access interfaces in the interface layer.

We introduced a semantic feedback protocol to bridge the conceptual gap between the relational model and RDF. It informs the client about invalid write request in an RDF-based format and provides recommendation on how to change the request for the better. We presented the semantic feedback ontology and the implementation in OntoAccess.

We showed that this platform-based approach performs comparable or better than existing read-only RDB-to-RDF mapping

approaches as well as current triple stores.

We further introduced a formal definition of our RDB-to-RDF mapping and proofs of its bidirectional properties. The rationale of the formal definition and the proofs is to show that our mapping language R3M is bidirectional and therefore not affected by the view update problem.

References

[1] World Wide Web Consortium, W3C Semantic Web Activity, http://www.w3.org/2001/sw/, 2011.

[2] C. Patel, S. Khan, K. Gomadam, TrialX: Using Semantic Technologies to Match Patients to Relevant Clinical Trials Based on Their Personal Health Records, in: Proceedings of the 8th International Semantic Web Conference.

[3] L. Ma, X. Sun, F. Cao, C. Wang, X. Wang, Semantic Enhancement for Enterprise Data Management, in: Proceedings of the 8th International Semantic Web Conference.

[4] A. Langegger, W. Wöss, M. Blöchl, A Semantic Web Middleware for Virtual Data Integration on the Web, in: Proceedings of the 5th European Semantic Web Conference.

[5] E. Prud'hommeaux, A. Seaborne, SPARQL Query Language for RDF, W3C Recommendation. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/, 2008.

[6] A. Seaborne, G. Manjunath, C. Bizer, J. Breslin, S. Das, I. Davis, S. Harris, K. Idehen, O. Corby, K. Kjernsmo, B. Nowack, SPARQL Update – A Language for Updating RDF Graphs, W3C Member Submission. http://www.w3.org/Submission/2008/SUBM-SPARQL-Update-20080715/, 2008.

[7] C. Ogbuji, SPARQL 1.1 Graph Store HTTP Protocol, W3C Working Draft. http://www.w3.org/TR/2011/WD-sparql11-http-rdf-update-20110512/, 2011.

[8] M. Hert, Relational Databases as Semantic Web Endpoints, in: Proceedings of the 6th European Semantic Web Conference.

[9] C. Bizer, A. Seaborne, D2RQ – Treating Non-RDF Databases as Virtual RDF Graphs, in: Proceedings of the 3rd International Semantic Web Conference.

[10] O. Erling, I. Mikhailov, RDF Support in the Virtuoso DBMS, in: Proceedings of the SABRE Conference on Social Semantic Web.

[11] J. Barrasa, O. Corcho, A. Gómez-Pérez, R2O, an Extensible and Semantically Based Database-to-Ontology Mapping Language, in: Proceedings of the 2nd Workshop on Semantic Web and Databases.

[12] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, D. Aumueller, Triplify – Light-Weight Linked Data Publication from Relational Databases, in: Proceedings of the 18th International World Wide Web Conference.

[13] C. Bizer, R. Cyganiak, D2R Server – Publishing Releational Databases on the Semantic Web, in: Proceedings of the 5th International Semantic Web Conference.

[14] S. Das, S. Sundara, R. Cyganiak, R2RML: RDB to RDF Mapping Language, W3C Working Draft. http://www.w3.org/TR/2010/WD-r2rml-20101028/, 2010.

[15] A. Malhotra, W3C RDB2RDF Incubator Group Report, http://www.w3.org/2005/Incubator/rdb2rdf/XGR-rdb2rdf-20090126/, 2009.

[16] H. Halpin, I. Herman, RDB2RDF Working Group Charter, http://www.w3.org/2009/08/rdb2rdf-charter, 2009. Last visited July 2011.

[17] A. Garrote, M. N. M. Garcia, RESTful Writable APIs for the Web of Linked Data Using Relational Storage Solutions, in: Proceedings of the WWW2011 Workshop on Linked Data on the Web.

[18] S. S. Sahoo, W. Halb, S. Hellmann, K. Idehen, T. T. Jr, S. Auer, J. Sequeda, A. Ezzat, A Survey of Current Approaches for Mapping of Relational Databases to RDF, http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf, 2009. Last visited July 2011.

[19] M. Hert, G. Reif, H. C. Gall, A Comparison of RDB-to-RDF Mapping Languages, in: Proceedings of the 7th International Conference on Semantic Systems.

[20] F. Bancilhon, N. Spyratos, Update Semantics of Relational Views, ACM Transactions on Database Systems (1981).

[21] T. Berners-Lee, Relational Databases on the Semantic Web, http://www.w3.org/DesignIssues/RDB-RDF.html, 2009. Last visited July 2011.

[22] C. Fürber, Ontology-Based Data Quality Management: Methodology, Cost, and Benefits, in: Proceedings of the 6th European Semantic Web Conference.

[23] M. Hert, G. Reif, H. C. Gall, Updating Relational Data via SPARQL/Update, in: EDBT Workshop Proceedings.

[24] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, K. Wilkinson, Jena: Implementing the Semantic Web Recommendations, in: Proceedings of the 13th International World Wide Web Conference.

[25] S. Schenk, P. Gearon, A. Passant, SPARQL 1.1 Update, W3C Working Draft. http://www.w3.org/TR/2010/WD-sparql11-update-20101014/, 2010.

[26] F. Manola, E. Miller, RDF Primer, W3C Recommendation. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/, 2004.

[27] D. Beckett, T. Berners-Lee, Turtle – Terse RDF Triple Language, W3C Team Submission http://www.w3.org/TeamSubmission/turtle/, 2011.

[28] C. Bizer, A. Schultz, The Berlin SPARQL Benchmark, International Journal on Semantic Web and Information Systems (2009).

[29] C. Bizer, A. Schultz, Berlin SPARQL Benchmark (BSBM) Specification – V2.0, http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/, 2008.

[30] M. Hert, G. Ghezzi, M. Würsch, H. C. Gall, How to "Make a Bridge to the New Town" using OntoAccess, in: Proceedings of the 10th International Semantic Web Conference.

[31] H. C. Gall, B. Fluri, M. Pinzger, Change Analysis with Evolizer and ChangeDistiller, IEEE Software (2009).

[32] G. Ghezzi, H. C. Gall, SOFAS: A Lightweight Architecture for Software Analysis as a Service, in: Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture.