Bridging the Technology Gap between Industry and Semantic Web: Generating Databases and Server Code from RDF

Markus Schröder[1,2], Michael Schulze[1,2], Christian Jilek[1,2] and Andreas Dengel[1,2]
[1]*Smart Data & Knowledge Services Dept., DFKI GmbH, Kaiserslautern, Germany*
[2]*Computer Science Dept., TU Kaiserslautern, Germany*

Keywords: Database Generator, Resource Description Framework, Relational Database, Create Read Update Delete, Representational State Transfer, Application Programming Interface.

Abstract: Despite great advances in the area of Semantic Web, industry rather seldom adopts Semantic Web technologies and their storage and query approaches. Instead, relational databases (RDB) are often deployed to store business-critical data, which are accessed via REST interfaces. Yet, some enterprises would greatly benefit from Semantic Web related datasets which are usually represented with the Resource Description Framework (RDF). To bridge this technology gap, we propose a fully automatic approach that generates suitable RDB models with REST APIs to access them. In our evaluation, generated databases from different RDF datasets are examined and compared. Our findings show that the databases sufficiently reflect their counterparts while the API is able to reproduce rather simple SPARQL queries. Potentials for improvements are identified, for example, the reduction of data redundancies in generated databases.

1 INTRODUCTION

The Resource Description Framework (RDF) (Raimond and Schreiber, 2014) is a well-established data model in the Semantic Web community. It is used to express facts about resources identified with uniform resource identifiers (URIs) (Berners-Lee et al., 1998) in the form of statements (subject, predicate and object). Ontologies (Gruber, 1993) are used to model domains and to share formally specified conceptualizations which can be expressed by using RDF Schema (RDFS) (Guha and Brickley, 2004). RDF-based data is typically stored in triplestores and queried with the SPARQL Protocol and RDF Query Language (SPARQL) (W3C SPARQL Working Group, 2013).

In our experience, beyond the Semantic Web and especially in industry, such technologies are rather seldom used. A case study for the manufacturing industry also points out this observation (Feilmayr and Wöß, 2016). Instead, relational databases (RDBs) are commonly used to store important and system critical data – information systems which are well-researched over 50 years. By implementing Application Programming Interfaces (APIs), a controlled access on these datasets with Create, Read, Update and Delete (CRUD) operations are provided for system developers. APIs often conform to the Representational State Transfer (REST) software architecture, utilize the Hypertext Transfer Protocol (HTTP) and exchange data in the JavaScript Object Notation (JSON) format.

Table 1: Comparison of storage and query approaches between Semantic Web and Industry.

Approach	Semantic Web	Industry
Storage	Triplestore	SQL Database
Domain Modeling	Terminology in Ontology	Database Schema
Data Modeling	RDF Statement Assertions	Database Records
Identification	URIs	Primary Keys
Query Interface	SPARQL	SQL / REST API
Exchange Format	Result Set / RDF	Result Set / JSON

By examining both sides, we observe distinct solutions regarding storage and query approaches. Table 1 summarizes and compares these findings: while the Semantic Web encourages the use of triplestores loaded with ontologies and assertions, industry prefers databases with defined schemata and records. Resources are identified with URIs on the Semantic Web, while database records use primary keys. Graph-oriented SPARQL queries and their special result set responses are opposed to relational SQL queries. On top of databases, additional web services can provide document-oriented REST APIs returning JSON documents.

Yet, some enterprises would greatly benefit from Semantic Web technologies. This also includes re-

Schröder, M., Schulze, M., Jilek, C. and Dengel, A.
Bridging the Technology Gap between Industry and Semantic Web: Generating Databases and Server Code from RDF.
DOI: 10.5220/0010186005070514
In *Proceedings of the 13th International Conference on Agents and Artificial Intelligence (ICAART 2021) - Volume 2*, pages 507-514
ISBN: 978-989-758-484-8

lated datasets and the way knowledge is modeled. We see a trend that more and more publicly available datasets are modeled and/or published in the RDF format, such as datasets in the Linked Open Data (LOD) cloud[1], DBpedia (Bizer et al., 2009) and Wikidata (Vrandečić and Krötzsch, 2014). Moreover, it has been become popular in data integration services and especially in knowledge services to construct and maintain knowledge graphs (Hogan et al., 2020) for selected use cases, for instance, when building a corporate memory (Maus et al., 2013). To embed these new datasets and technologies in workflows and processes, corporations would have to spend considerable efforts. In order to keep a company's overhead to a minimum, we suggest transforming RDF-based datasets back to storage systems they are more familiar with, namely RDBs. This way, data can be maintained by admins without knowledge about Semantic Web, but with the trade-off of losing the flexibility of Semantic Web technologies. This implies that this transformation should be performed only once and further data changes should be made in the generated database. For integration purpose, we further recommend that enterprises implement CRUD REST APIs to provide access and manipulation layers for their developers. Since such conversions and implementations are quite tedious and costly when executed manually, a fully automatic way to generate the envisioned assets would be helpful.

Since related work did not appropriately address this particular use case, in this paper, we provide a solution to this challenge. A generation procedure is described that accepts an arbitrary RDF(S) dataset and generates an RDB with a CRUD REST API to access and modify it. Doing this, raises the following research questions which are addressed in our experiments:

1. How well do the generated RDBs reflect their RDF dataset counterparts? By using various RDF datasets, we check if any critical data is missing in the databases and how they are structured.
2. How well can the generated CRUD REST API reproduce queries that would have been performed with SPARQL? To answer this, we try to query same information with our API compared with given SPARQL queries.
3. What limitations do the generated databases and interfaces have? In our experiments, we reveal and discuss shortcomings of our approach.

For future research, the source code of our algorithm and the evaluation material is publicly available at GitHub[2].

This paper is structured as follows: in the next section (Sec. 2) we investigate procedures and tools in literature that also transform RDF to RDB. This is followed by our own approach in Section 3. Section 4 presents the evaluation of our method and answers the stated research questions. We close the paper with a conclusion and an outlook in Section 5.

2 RELATED WORK

One can find a lot of papers in literature which are related to the conversion of RDB (and similar formats) to RDF. However, only few works actually investigated the opposite direction (from RDF to RDB).

An early work (Teswanich and Chittayasothorn, 2007) transforms RDF documents to databases to apply Business Intelligence (BI) technologies. RDFS-related information is stored in meta tables, for example, in relations like *class*, *property* and *sub_class_of*. For each class and object property (regardless of its cardinality) a table is created. SQL statements demonstrate how a generated database can be queried.

Similarly, the R2D approach (Ramanujam et al., 2009) generates databases from RDF data to reuse visualization tools. The authors suggest several improvements regarding (Teswanich and Chittayasothorn, 2007): in contrast, their approach still works, albeit no ontological information is available in the input dataset. Moreover, it considers the cardinality of properties to avoid the creation of tables, and it also handles blank nodes.

RDF2RDB[3] is a Python based tool that converts a given RDF/XML document into a MySQL database. The generation approach is also comparable with the one from (Teswanich and Chittayasothorn, 2007), except it does not generate meta tables for RDF schema information.

RETRO (Rachapalli et al., 2011) focuses on query translation with the same motivational arguments as we have about bridging the gap between Semantic Web and industry. However, their approach does not physically transform RDF to an RDB. Instead, they use a fixed schema mapping approach that virtually maps all predicates from the A-Box statements to relational tables having two columns, namely for subject and object.

Although R2D and RDF2RDB are comparable to our approach, we did not find any related work that also takes into account the generation of a suitable REST interface to access the data.

[1]https://lod-cloud.net/

[2]https://github.com/mschroeder-github/rdf-to-rdb-rest-api

[3]https://github.com/michaelbrunnbauer/rdf2rdb

3 APPROACH

Our approach is divided into three phases. First, RDF data is analyzed to receive valuable insights about the nature of the dataset. Secondly, these findings are used to design a suitable database model which consists of tables, columns and data records. Thirdly, based on the database model, Java source code is generated to directly provide developers with usable data classes, a database access layer, a server and its REST interface.

Figure 1 presents a very small example of the approach. A simple RDF dataset about persons reading books (on the left in RDF Turtle syntax) is transformed into three tables (in the middle): two of them represent persons resp. books while the third one models the many-to-many relation between them. On the right side, corresponding source code is generated to automatically implement the REST API. The ready to use server application can be utilized by developers to query a book resource which returns a JSON representation of it.

3.1 Analysis of RDF

Before the RDF model is analyzed, possible Blank Nodes in the dataset are skolemized (Mallea et al., 2011). This means that they are replaced with randomly generated URIs in a consistent manner. That way further analysis and processing is simplified without any semantic change to the RDF model.

Next, for generating the RDB tables and filling them with records later, classes and their instances have to be discovered. They are collected by scanning through the assertion box (A-Box) statements. In this process, instances having more than one type (multi-typed instances) are detected. The instances' properties together with their domains and ranges are further examined since they will be modeled as either table columns or many-to-many tables. We divide the properties into object properties (objects are resources) and data type properties (objects are literals). In case of object properties, the object's type is inspected (range). If the object is not further mentioned in the RDF dataset or has no type, it is classified as a dangling resource. Later, these resources will be stored in the database as textual URIs (instead of numeric IDs) because they are not present in the database as a record. At least, this allows to look up such resources by following the links. Regarding data type properties, a suitable SQL storage class is inferred which can be either text, real, integer or binary large object. If the literal has a language tag, the property is assigned to be a special language string property.

After that, the properties cardinalities are analyzed to decide if a relationship should be modeled as a foreign key column or a many-to-many table. For each property, based on the given data, it is deduced if it has a one-to-one, one-to-many, many-to-one or many-to-many cardinality by scanning through the A-Box statements. The special `rdf:type` property is always assumed to be many-to-many because multiple resources can have multiple types in RDF. Additionally, all the properties' domains and ranges are collected. Note that one predicate can have subjects and objects with various domains and ranges. To retrieve clear mappings from domains to ranges, distinct domain-range pairs are calculated.

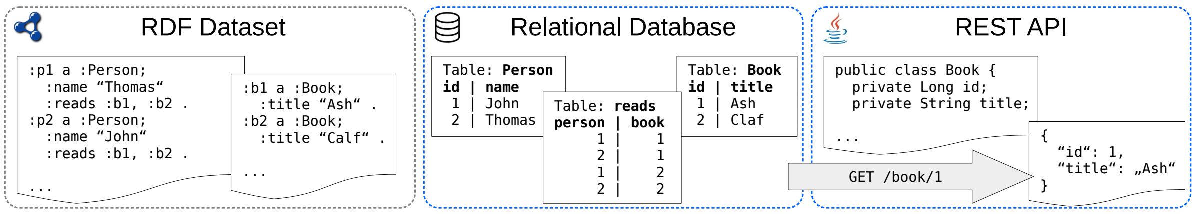
3.2 Conversion to RDB

Our generated RDBs follow the type-store approach mentioned in (Ma et al., 2016). Multi-valued attributes are avoided by using many-to-many tables when cardinalities require it. We assume that the type-based structure is easier to grasp for developers than a horizontal or vertical structure.

Complying with the type-store approach, for each determined type from the previous step, a table is designated that will contain all instances of this type in form of records. We denote such relations as entity tables. They contain mandatory numeric *id*-columns which serve as primary keys. We do not add a *uri*-column since the numeric primary key already serves as an identifier and databases usually do not model another textual identifier, like a URI, for their records.

An entity table is annotated with its representing RDF class. All properties matching the class with their domain are assumed to be a column of this table. However, there are two special cases with respect to the cardinality of the properties. In case of a one-to-many cardinality, the column is placed in the referring table instead. This is a usual step when entity-relationship models (ER-models) are instantiated as relations that should satisfy the third normal form (Kent, 1983). In case of a many-to-many cardinality, no column in an entity table is created. Instead, an extra table is modeled that contains two columns, namely to refer to subject and object. If we have a language string property at hand, another *lang* column is added to store the language tag.

Next, tables are filled with data records. To do that consistently, each resource in the RDF dataset is assigned to a unique numeric ID. A special *_res_id*-relation records for each URI the mapped ID for later lookup. Using this information, records of entity tables obtain distinct IDs for their primary keys. By



```
:p1 a :Person;
  :name “Thomas“
  :reads :b1, :b2 .
:p2 a :Person;
  :name “John“
  :reads :b1, :b2 .
...
```

```
:b1 a :Book;
  :title “Ash“ .
:b2 a :Book;
  :title “Calf“ .
...
```

```
Table: Person
id | name
 1 | John
 2 | Thomas
```

```
Table: reads
person | book
     1 |    1
     2 |    1
     1 |    2
     2 |    2
```

```
Table: Book
id | title
 1 | Ash
 2 | Claf
```

```
public class Book {
  private Long id;
  private String title;
...
```

```
{
  “id“: 1,
  “title“: „Ash“
}
```

Figure 1: Illustrating example of our approach: An RDF dataset about persons reading books (left) is converted to a relational database (middle). The further generated REST API contains corresponding source code (right) and allows to query the database with REST calls (indicated by the gray arrow). The response is a JSON representation of the data.

scanning through the outgoing edges of every resource, record fields are allocated with the acquired objects. For many-to-many tables, all statements with the corresponding predicate are considered. Those statements have the following mandatory condition: the subject's type matches the given domain and the object's type matches the given range. Only those statements are inserted as appropriate records in the table. Last, SQL statements are formulated to populate an SQLite[4] database with the modeled tables and their records.

3.3 Generation of REST API Code

To generate source code and file contents, we utilize the template engine Apache FreeMarker[5]. In this phase, two Apache Maven[6] projects are produced by our approach: an *api* project containing data classes together with the database access layer and a *server* project with the RESTful communication logic.

For the *api* project, every entity table from the previous part is converted into a Plain Old Java Object (POJO)[7]. The tables' columns are turned into attributes of the POJO class. All many-to-many tables that could be joined with the entity table on the first column become attributes of type `java.util.List`. In case of a table that originated from a language string property, a special `LangString` POJO class holding string and language tag is used as the attribute's type.

After the data classes, the database controller class is generated. For each Java class, a corresponding *select*, *insert*, *update* and *delete* method is generated. These methods internally use SQL to communicate with the database. In case of the *select* method, filtering is supported by using the resource query language (RQL). For that we parse the syntax[8] and filter results based on given RQL expression.

The *server* project has a dependency to the *api* project. Thus, it reuses the POJO classes, database access logic and provides the RESTful communication. We use the Spark[9] framework to implement the server application. For each Java class (from the API) a corresponding REST endpoint is generated. The endpoints are able to interpret and perform HTTP GET, POST, PUT, PATCH and DELETE requests by utilizing the database controller from the *api* project. A converter from POJO to JSON (and vice versa) is provided to exchange data.

4 EVALUATION

The evaluation of our approach consists of three parts. First, several diverse and publicly available[10] RDF(S) datasets are transformed into RDBs to check the algorithm's ability to handle different datasets and to analyze the generated databases. Second, we make a comparison between the outcome of a similar tool with our results using the same input dataset. Third, a SPARQL benchmark is used which provides predefined queries and an RDF dataset generator. Our re-engineered database is compared with the benchmark's generated SQL database. We also investigate how well our REST API can reproduce given SPARQL queries. The evaluation section is closed with a general discussion of the results.

4.1 Datasets

The Linked Open Data (LOD) cloud[11] is a hub that refers to all kinds of publicly available RDF-based resources or endpoints. To test our algorithm, we randomly selected six rather small RDF datasets from the LOD cloud. Additionally, a private dataset from an industrial scenario and a generated one from

[4]https://www.sqlite.org/
[5]https://freemarker.apache.org/
[6]https://maven.apache.org/
[7]https://www.martinfowler.com/bliki/POJO.html
[8]https://github.com/jirutka/rsql-parser

[9]http://sparkjava.com/
[10]Except of one private dataset that was obtained from an industrial scenario.
[11]https://lod-cloud.net/

the Berlin SPARQL Benchmark (BSBM) (Bizer and Schultz, 2011) are examined too. Only relatively small sized datasets with triples around a 5-digit number of statements are chosen since we are not interested in testing the time and memory performance of our approach. Instead, we investigate what effects various datasets have on the output of our algorithm. The findings will be discussed at the end of the evaluation section. Characteristics of the eight datasets and the resulting RDBs are presented in Table 2. In the following, we briefly describe each dataset and discuss the results.

TBL-C[12] is the RDF representation of Tim Berners-Lee's electronic business card. Since the instance representing himself has two types (`foaf:Person` and `con:Male`), this record is redundantly stored in two tables with identical columns. Having multiple types causes also the generation of several many-to-many tables with equivalent data.

The Copyright Term Bank **CTB**[13] dataset contains copyright terminology. After the conversion, four entity tables contain data records: *concept*, *lexical_entry*, *lexical_sense* and *sense_definition*. However, *lexical_sense* does not have any functional properties, thus containing no columns (except the mandatory *id* column).

The Edinburgh Associative Thesaurus RDF dataset **EAT**[14] (Hees et al., 2016) contains associations of terms. Despite the large number of statements (1,674,376), it only has two classes which results in two tables, namely *association* with 325,588 records and *term* with 23,218 rows.

Pokedex[15] is an RDF catalog of fictitious monsters of the popular Pokémon franchise[16]. The main table *pokemon* contains all functional properties and has 493 entries. All Pokémon species are listed in the *species* table. Because there is a one-to-many relationship between species (one) and Pokémon (many), the property *pkm:speciesOf* is represented with a column in the *pokemon* table.

Betweenourworlds **BOW**[17] is a dataset about animes which are drawn animations originated from Japan. Besides having the largest number of statements in our selection (4,041,676), it also has the largest number of instances with multiple types (349,195). The main reason is that every anime is typed with at least *dbo:Anime*, *dbo:Cartoon* and *dbo:Work*.

S-IT[18] is a dataset generated from the traveling website about the Salzburg state in Austria in Italian language. The dataset has been built with WordLift[19] (Volpini and Riccitelli, 2015), which is a plugin that annotates website content with linked data. Although, it has the second lowest number of statements (4,477), the generated database contains the most tables. This can be explained by the high number of classes (406) and resources (81) which are instances of multiple classes.

Our private RDF dataset **IndScn** from an industrial scenario consists of meta-data about documents and their revisions. The conversion went straight forward: 16 classes result in the desired 16 tables with appropriate many-to-many tables for the properties.

The **BSBM**[20] (Bizer and Schultz, 2011) dataset has been generated with the provided generator tool (version 0.2). Its domain is about produced, offered and reviewed products. With a product count parameter of 100, 40,177 RDF statements have been generated. After the conversion, all main entity tables were created, namely *product*, *product_feature*, *product_type*, *person*, *producer*, *offer*, *review* and *vendor*.

4.2 Comparison with RDF2RDB

RDF2RDB[21] is an open-source tool that converts RDF to relational databases. The procedure, which is written in Python 2, reads RDF files and fills a MySQL database with tables and records. We use two datasets, namely No. 1 and No. 8 in Table 2, to compare exemplarily the behavior of the procedures.

For demonstration purpose, the developer provides the generated database from Tim Berners-Lee's electronic business card[22]. Since we did also the conversion of the same dataset (No. 1 in Table 2), we can compare the resulting databases: RDF2RDB produced 59 tables, while our procedure made only 18 relations. The main reason for this is that RDF2RDB converts more properties into many-to-many tables. It also generated a *thing* table that sparsely records all `owl:Thing` resources with their properties. Another difference is that RDF2RDB provides a *labels*-table and *uris*-table. In the *labels*-relation, all URIs from the dataset are related to their labels to enable label-based searches. The *uris*-table lists for each resource

[12]http://www.w3.org/People/Berners-Lee/card.rdf

[13]https://lod-cloud.net/dataset/copyrighttermbank

[14]https://lod-cloud.net/dataset/associations

[15]https://lod-cloud.net/dataset/data-incubator-pokedex

[16]https://www.pokemon.com/

[17]https://betweenourworlds.org/ (Release 2020-06)

[18]https://lod-cloud.net/dataset/salzburgerland-com-it

[19]https://wordlift.io/

[20]http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/

[21]https://github.com/michaelbrunnbauer/rdf2rdb

[22]https://www.netestate.de/Download/RDF2RDB/timbl.txt

Table 2: Characteristics of eight RDF datasets and their generated databases by our approach: the number of RDF statements (Stmts), classes (Cls), multi-typed instances (MT) and average number of types per MT instance (avgMT). Further, the number of object property (OP), datatype properties (DP), and properties which have the following cardinality: one-to-one (OO), many-to-one (MO), one-to-many (OM) and many-to-many (MM). Regarding the database, we show the number of generated entity tables (ET), many-to-many tables (MMT) and average number of columns per entity table (avgCol).

No.	Name	Stmts	Cls	MT	avgMT	OP	DP	OO	MO	OM	MM	ET	MMT	avgCol
1	TBL-C	109	5	1	2	26	18	31	19	7	1	5	11	12.2 ± 12.4
2	CTB	10,853	4	0	-	10	5	4	6	1	4	4	7	3.0 ± 1.6
3	EAT	1,674,376	2	0	-	3	3	1	5	0	0	3	1	2.7 ± 2.1
4	Pokedex	26,562	19	0	-	9	29	13	28	5	3	19	40	3.5 ± 6.7
5	BOW	4,041,676	15	349,195	2.9 ± 0.9	7	19	2	9	0	15	15	180	5.3 ± 3.0
6	S-IT	4,477	406	81	14.3 ± 9.0	9	25	18	6	3	7	406	3,056	2.6 ± 1.0
7	IndScn	25,016	16	0	-	24	37	4	34	0	23	16	74	5.7 ± 4.5
8	BSBM	40,177	22	100	2.0 ± 0.0	12	28	16	22	0	2	22	17	13.7 ± 5.5

its class and assigned numeric ID. This table is comparable with our *_res_id*-relation.

We also use the generated BSBM dataset (No. 8 in Table 2) to compare the tool's outcome with our result. While RDF2RDB's database contains 145 tables, our method generated 41 relations. RDF2RDB created for each product type a table but used for its name the product type's label. As before, a lot of many-to-many tables are created by the tool. The entity tables are nearly equal, except that RDF2RDB decided to model some relations as tables instead of columns.

In conclusion, RDF2RDB comply more with the RDF model with the side effect of generating more tables, especially many-to-many tables. Our version is more data-driven, thus more properties are modeled as columns since their observed cardinalities allow that. In both cases the given facts could be stored in the databases. However, some more specific RDF structures, like RDF containers or Blank Node information, is not stored.

4.3 REST Interface Test with BSBM

The Berlin SPARQL Benchmark (BSBM) (Bizer and Schultz, 2011) provides a dataset generator and SPARQL queries to enable performance comparisons of storage systems with a SPARQL endpoint. First, using the dataset generator, we will examine how well our approach re-engineers the benchmark's intended database. Second, we utilize provided queries to check if our API can reproduce them.

BSBM's dataset generator can produce, aside from usual RDF, a SQL description of a database containing equivalent data. In the following, we examine how close our method can re-engineer this database only from the RDF dataset. Our generated database was already described in the previous section (No. 8 in Table 2). The comparison shows that we found all necessary entity and many-to-many tables. Regarding the *offer* table, our version misses the *producer* column. However, this is not a surprise because the dataset does not contain any linkage between offer and producer. Concerning the *product_type*-table, BSBM additionally added a *parent* and *sub_class_of*-column to model the class hierarchy. Another difference is the way dates are stored. While BSBM uses text representations in ISO 8601, our database uses milliseconds since the UNIX epoch. Because of multi-typed instances, our procedure created some unnecessary tables per type as well as some many-to-many tables. In conclusion, despite small differences, our re-engineered database is similarly modeled and contains all intended data.

The benchmark's main purpose is to provide queries to test performance of SPARQL endpoints. With 12 formulated exploration queries[23], endpoints can be queried in various ways by using mainly SELECT queries, a DESCRIBE query and a CONSTRUCT query. Since they contain substitution parameters, they are actually templates that can be instantiated in various ways. In our evaluation, we investigate if our generated REST API can reproduce these queries by testing them with meaningful substitutions. Our expectation is that the REST API can retrieve equal information.

In the following, we present for each BSBM query its REST API call counterpart. For readability and reproducibility reasons they are available online[24]. In some cases more then one call has to be made to join data in the client appropriately. We omitted calls that would have been necessary to retrieve further information about referred resources, like mostly their labels. The resource query language (RQL) is often used to mimic SPARQL's basic graph patterns and filter possibilities. We put the responsibility for or-

[23]http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/ExploreUseCase/index.html

[24]https://github.com/mschroeder-github/rdf-to-rdb-rest-api/blob/master/bsbm-queries.txt

dering the results to the client. The interpretation of optional information (expressed in SPARQL with the OPTIONAL keyword) is also up to the developer.

What follows are short explanations how some queries are reproduced with our API. Regarding Query 3, the label of the optional `ProductFeature2` is checked to be not bound. This is solved in our case by excluding this feature with RQL's `=out=` operator which yields to the same result. In Query 4, the UNION statement is imitated by a logical *or* construction (in RQL a `','`) whether `ProductFeature2` or `ProductFeature3` (or both) are the product's features. Concerning Query 5, the SPARQL query contains filters with arithmetic expressions that can not be emulated by our RQL engine. Thus, in a second call (5b), the client will be responsible for filtering the results to find matching products. Using RQL's `=regex=` operator, we reproduce the regular expression filter in Query 6. Query 7 demonstrates that joining data involves multiple REST calls and has to be done on the client-side. Since data types of literals are not stored in our database, the currency of an offer's price can not be retrieved. With the special `=lang=` operator in RQL, language strings can be filtered as shown in Query 8. Regarding Query 9, a first call determines the reviewer of `ReviewXYZ`, while in a second call (not shown in the code), the reviewer is retrieved by using the `/person` endpoint. In Query 11, also incoming edges of a given `OfferXYZ` are queried. Since our REST API can only retrieve outgoing edges of a resource, this request is only partially reproducible. In case of Query 12, which is a CONSTRUCT query, the actual construction part lies in the responsibility of the client.

4.4 Discussion

At the beginning of the paper, we stated three question that can now be answered based on the evaluation results. When answering the first two questions, we also address limitations of our generated databases and interfaces.

How well do the generated RDBs reflect their RDF dataset counterparts? The observation of the eight generated databases (Table 2) shows that the information content of a dataset's A-Box is sufficiently reflected by its database counterpart, i.e. we did not miss any critical information. However, it is possible to express the same information with different relational models. Compared to RDF2RDB, our procedure created less tables but still more than intended by BSBM. The evaluation reveals that the handling of multi-typed instances poses the main challenge. Since RDF classes directly correspond to RDB tables, a resource having more then one type is redundantly distributed among respective tables. It also causes the generation of more many-to-many tables because such a table relates one certain domain to one particular range. That means that unnecessary and unwanted data redundancies occur.

Another major challenge is the decision for a trade-off that determines whether a property becomes a table or a column. If, on the one extreme, every property has a table counterpart, the number of tables explodes and joins become inevitable which make queries more complex. If, on the other extreme, properties become columns in tables, data redundancy occurs since relations would violate the second normal form. That is why generation procedures should have a meaningful (maybe configurable) trade-off. We decided to infer the cardinality of properties based on existing A-Box statements to minimize the number of many-to-many tables. However, this makes the properties' cardinalities unchangeable in the database model. For example, a one-to-many relationship can not be easily turned into many-to-many relationship once the database model is defined.

Evaluation also points out smaller issues in our generator. Tables having only an *id* column could be removed because they provide no further information. We also noticed that RDF data types are missing in our database model. In conclusion, our generated RDBs indeed reflect their RDF dataset counterparts sufficiently but contain unnecessary redundant data as well.

How well can the generated CRUD REST API reproduce queries that would have been performed with SPARQL? Many SPARQL operations like aggregates, arithmetic expressions, sub-queries and various functions are not supported by our rather simple API. When such queries become more complex, we make the client responsible to send more requests and to process the results accordingly. A major issue is that the API does not provide a join-mechanism. Since each endpoint represents a class and returns the corresponding instances, the join has to be performed by the client. Another limitation is the inability to retrieve a resource's incoming edges since the underlying database model is not designed for that. Yet, by using the BSBM benchmark, we showed that our interface could in almost all cases retrieve the same information as the given SPARQL queries. Hence, our conclusion is that the API can reproduce rather simple and common queries, while more complex ones have to be handled by the client.

5 CONCLUSION AND OUTLOOK

Especially in industry, Semantic Web technologies are rather seldom used. Since corporations would greatly benefit from available and future RDF-based datasets, we suggested to bridge the technology gap by a fully automatic conversion of RDF to RDBs together with CRUD REST APIs. Our experiments suggest that in comparison to related work, our reengineered databases reflect their RDF counterparts with less tables. Moreover, our generated REST APIs are able to reproduce rather simple and common SPARQL queries. We identified as a remaining challenge the generation of unnecessary data redundancies because of multi-typed instances.

Future work should find an appropriate way to model the database to reduce the high number of tables and data redundancies. In this regard, as already pointed out, algorithms should provide a configurable trade-off to decide whether properties become many-to-many tables or simple columns. Thus, the right setting can be dependent on a particular use case. Since RDF datasets can change over time, we also suggest that future procedures provide an update-mechanism in order to avoid rebuilding the whole database (and possibly removing already inserted data). Moreover, to also process larger datasets (like DBpedia), we intend to reduce our algorithm's memory usage.

ACKNOWLEDGEMENTS

This work was funded by the BMBF project SensAI (grant no. 01IW20007).

REFERENCES

Berners-Lee, T., Fielding, R. T., and Masinter, L. (1998). Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396, RFC Editor.

Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., and Hellmann, S. (2009). DBpedia - A crystallization point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154–165.

Bizer, C. and Schultz, A. (2011). The berlin SPARQL benchmark. In *Semantic Services, Interoperability and Web Applications - Emerging Concepts*, pages 81–103. CRC Press.

Feilmayr, C. and Wöß, W. (2016). An analysis of ontologies and their success factors for application to business. *Data Knowl. Eng.*, 101:1–23.

Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5:199–220.

Guha, R. and Brickley, D. (2004). RDF Vocabulary Description Language 1.0: RDF Schema. W3c recommendation, W3C.

Hees, J., Bauer, R., Folz, J., Borth, D., and Dengel, A. (2016). Edinburgh associative thesaurus as RDF and dbpedia mapping. In *The Semantic Web - ESWC 2016 Satellite Events, Heraklion, Crete, Greece, May 29 - June 2, 2016, Revised Selected Papers*, volume 9989 of *LNCS*, pages 17–20.

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Gayo, J. E. L., Kirrane, S., Neumaier, S., Polleres, A., Navigli, R., Ngomo, A.-C. N., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., and Zimmermann, A. (2020). Knowledge graphs. *arXiv 2003.02320.*

Kent, W. (1983). A simple guide to five normal forms in relational database theory. *Commun. ACM*, 26(2):120–125.

Ma, Z., Capretz, M. A. M., and Yan, L. (2016). Storing massive resource description framework (RDF) data: a survey. *Knowl. Eng. Rev.*, 31(4):391–413.

Mallea, A., Arenas, M., Hogan, A., and Polleres, A. (2011). On blank nodes. In *The Semantic Web - ISWC 2011 - 10th Int. Semantic Web Conf., Bonn, Germany, October 23-27, 2011, Proc., Part I*, volume 7031 of *LNCS*, pages 421–437. Springer.

Maus, H., Schwarz, S., and Dengel, A. (2013). *Weaving Personal Knowledge Spaces into Office Applications*, pages 71–82. Springer.

Rachapalli, J., Khadilkar, V., Kantarcioglu, M., and Thuraisingham, B. (2011). Retro: a framework for semantics preserving sql-to-sparql translation. *The University of Texas at Dallas*, 800:75080–3021.

Raimond, Y. and Schreiber, G. (2014). RDF 1.1 Primer. W3c note, W3C.

Ramanujam, S., Gupta, A., Khan, L., Seida, S., and Thuraisingham, B. M. (2009). R2D: A bridge between the semantic web and relational visualization tools. In *Proc. of the 3rd IEEE Int. Conf. on Semantic Computing (ICSC 2009), 14-16 September 2009, Berkeley, CA, USA*, pages 303–311. IEEE Computer Society.

Teswanich, W. and Chittayasothorn, S. (2007). A transformation from rdf documents and schemas to relational databases. In *2007 IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing*, pages 38–41. IEEE.

Volpini, A. and Riccitelli, D. (2015). Wordlift: Meaningful navigation systems and content recommendation for news sites running wordpress. In *Proc. of the ESWC Developers Workshop 2015 co-located with the 12th Extended Semantic Web Conference (ESWC 2015), Portorož, Slovenia, May 31, 2015*, volume 1361 of *CEUR Workshop Proc.*, pages 20–22. CEUR-WS.org.

Vrandečić, D. and Krötzsch, M. (2014). Wikidata: A Free Collaborative Knowledgebase. *Communications of the ACM*, 57(10):78–85.

W3C SPARQL Working Group (2013). SPARQL 1.1 Overview. W3c recommendation, W3C.