AN APPROACH FOR THE INCREMENTAL EXPORT OF RELATIONAL DATABASES INTO RDF GRAPHS

Nikolaos Konstantinou, Dimitrios-Emmanuel Spanos, Dimitris Kouis
Hellenic Academic Libraries Link, National Technical University of Athens
Iroon Polytechniou 9, Zografou, 15780, Athens, Greece
{nkons, dspanos, dimitriskouis}@seab.gr

Nikolas Mitrou
School of Electrical and Computer Engineering, National Technical University of Athens
Iroon Polytechniou 9, Zografou, 15780, Athens, Greece
mitrou@cs.ntua.gr

Several approaches have been proposed in the literature for offering RDF views over databases. In addition to these, a variety of tools exist that allow exporting database contents into RDF graphs. The approaches in the latter category have often been proved demonstrating better performance than the ones in the former. However, when database contents are exported into RDF, it is not always optimal or even necessary to export, or dump as this procedure is often called, the whole database contents every time. This paper investigates the problem of incremental generation and storage of the RDF graph that is the result of exporting relational database contents. In order to express mappings that associate tuples from the source database to triples in the resulting RDF graph, an implementation of the R2RML standard is subject to testing. Next, a methodology is proposed and described that enables incremental generation and storage of the RDF graph that originates from the source relational database contents. The performance of this methodology is assessed, through an extensive set of measurements. The paper concludes with a discussion regarding the authors' most important findings.

Keywords: Linked Open Data; Incremental; RDF; Relational Databases; Mapping.

1. Introduction

The Linked Data movement has lately gained considerable traction and during the last few years, the research and Web user communities have invested some serious effort to make it a reality. Nowadays, RDF data on a variety of domains proliferates at increasing rates towards a Web of interconnected data. Access to government (data.gov.uk), financial (openspending.org), library (theeuropeanlibrary.org) or news data (guardian.co.uk/data), are only some of the example domains where publishing data as RDF increases its value.

Systems that collect, maintain and update RDF data are not always using triplestores at their backend. Data that result in triples are typically exported from other, primary sources into RDF graphs, often relying on systems that have a Relational Database Management System (RDBMS) at their core, and maintained by teams of professionals that trust it for mission-critical tasks.

Moreover, it is understood that experimenting with new technologies – as the Linked Open Data (LOD) world can be perceived by people and industries working on less frequently changing environments – can be a task that requires caution, since it is often difficult to change established methodologies and systems, let alone replace by newer ones. Consider, for instance, the library domain, where a whole living and breathing information ecosystem is buzzing around bibliographic records, authorities records, digital object records, e-books, digital articles etc., where maintenance and update tasks are unremitting. In these situations, changes in the way data is produced, assured for its quality and updated affects people's everyday working activities and therefore, operating newer technologies side-by-side for a period of time before migrating to new technologies seems the only applicable – and sensible – approach.

Therefore, in many cases, the only viable solution is to maintain triplestores as an alternative delivery channel, in addition to production systems, a task that becomes increasingly multifarious and performance-demanding, especially when the primary information is rapidly changing. This way the operation of information systems remains intact, while at the same time they expose seamlessly their data as LOD.

Several mapping techniques between relational databases and RDF graphs have been introduced in the bibliography, among which various tools, languages, and methodologies. Thus, in order to expose relational database contents as LOD, several policy choices have to be made, since several alternative approaches exist in the literature, without any one-size-fits-all approach[1].

When exporting database contents as RDF, one of the most important factors to be considered is whether RDF content generation should take place in real-time or should database contents be dumped into RDF asynchronously[2]. In other words, the question to be answered is whether the RDF view over the relational database contents should be transient or persistent. Both approaches constitute acceptable, viable approaches, each with its own characteristics, its benefits and its drawbacks.

In contexts where data update is not frequent, as is the case in digital repositories, real-time SPARQL-to-SQL conversions are not the optimal approach, despite the presence of database performance improvement techniques (e.g. indexes) that would presumably increase performance compared to plain RDF graphs[2]. When querying, the round-trips to the database pose an additional burden, while RDF dumps perform much faster. The performance difference is even more visible, especially in cases when SPARQL queries involve many triple patterns, which, subsequently translated to SQL queries, usually include numerous "expensive" `JOIN` statements. Additionally, it must be noted that asynchronous RDF dumps of the database, leave current established practices untouched, by doing the additional work in extra steps, without replacing existing steps in the information processing workflow.

The performance optimization problem defined and analyzed in this paper focuses on providing a persistent RDF view over relational database contents and, more specifically, the minimization of the triplestore downtime each time the RDF export is materialized, which corresponds to the time that is required until the triplestore contents are replaced/updated by the new ones, reflecting the changes in the database. In order to do so, an incremental approach is introduced for the generation and storage of the RDF graph, as opposed to fully replacing the graph contents with the latest version each time the RDF dump is materialized. The problem can be further distinguished into two sub-problems:

- Sub-problem #1: *Incremental transformation*. This states that each time the transformation is executed, the entire RDF graph is not created from scratch, but instead only the parts that need to be updated are computed. As we will see, an RDF graph needs to be updated when either the contents of the database or the mapping itself has changed.
- Sub-problem #2: *Incremental storage*. This is a problem that is investigated only when the resulting RDF graph is stored in a persistent RDF store, since file storage inherently does not allow for efficient incremental storage techniques. The problem here, regardless of whether the transformation took place fully or incrementally is about storing (persisting) to the destination RDF graph only the triples that were modified and not the whole graph.

RDF views on relational database contents can be materialized either synchronously (i.e. in real-time), or asynchronously (ad hoc, as it is often mentioned). We note that the notion of real-time is tightly coupled with the concepts of event and response time[3]. An event can be defined as "any occurrence that results in a change in the sequential flow of program execution" and the response time as "the time between the presentation of a set of inputs and the appearance of all the associated outputs". Contrarily, in the ad hoc, asynchronous approach, the user can run the execution command that will dump the relational database contents into an RDF graph at will.

Given the definitions above, the incremental approach can be characterized as ad hoc, and not real-time since transformations are performed asynchronously. Emphasis is given in studying processing times that each transformation step requires towards the RDF graph generation, taking into account parameters such as the output medium, whether the change is incremental or not, the total size of the resulting graph, and the percentage of the triples of the initial graph that were changed.

The paper is structured as follows: Section 2 introduces some necessary preliminary definitions and concepts on the subject relational-to-RDF mapping and relevant technologies, Section 3 provides an overview of related work in the literature, Section 4 introduces and analyzes the proposed approach, Section 5 describes the measurement environment and parameters, presents the performance measurements, Section 6 provides a discussion over the results while Section 7 concludes the paper with our most important observations and future plans to expand on this work.

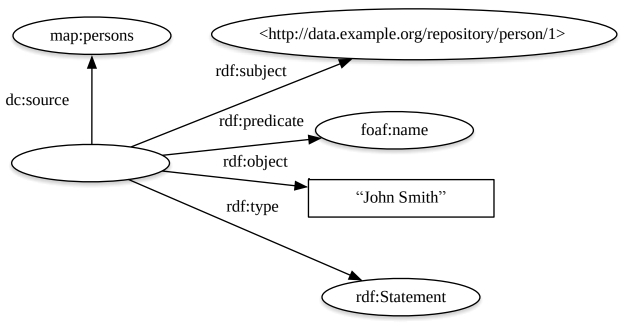
2. Background

In this section we recall some basic facts about reified statements and R2RML (including *triple maps* and *mapping document*), delineating the needed notation and terminology for the paper.

2.1 Reification in RDF

Reification or reified statements[a] in RDF, is the ability to treat an RDF statement as an RDF resource, and hence to make assertions about that statement. In other words apart from "pure" statements that apply for describing Web resources, RDF can also be used to create statements about other RDF statements, known as high-order statements. In order to construct reification statements, four properties are necessary, namely *subject*, *predicate*, *object* and *type*. In Fig. 1, an example is shown of how a standard RDF triple is being reified followed by a provenance note (`dc:source`). In the scope of the hereby-presented work, the term *reified model* is introduced, in analogy to the term *reified statement,* appointing a model that contains only reified statements.

[a] RDF 1.1Concepts and Abstract Syntax, W3C Recommendation 25 February 2014, http://www.w3.org/TR/rdf11-concepts/



```
<http://data.example.org/repository/person/1>
foaf:name "John Smith" .
```

becomes

```
[] a rdf:Statement ;
   rdf:subject
<http://data.example.org/repository/person/1> ;
   rdf:predicate foaf:name ;
   rdf:object "John Smith" ;
   dc:source map:persons .
```

Fig. 1. Annotated reified statement example.

Note that the result of the reification process is a resource that can participate as a subject in another triple. Thus, reification process, in our work aims to store information about every triple, regarding the mapping definition that produced it, meaning adding the proper annotation containing the mapping declaration (`map:persons` in the example) that led to its generation.

2.2 R2RML

R2RML[b] (*RDB 2 RDF Mapping Language*) is a language for expressing customized mappings from relational databases to RDF datasets. Every mapping follows a database schema and vocabulary specification, producing, after its application, an RDF dataset, ready for SPARQL[c] (*SPARQL Protocol and RDF Query Language*) queries. R2RML processors can offer virtual SPARQL interfaces over the mapped relational data, Linked Data endpoints or generate RDF dumps.

To retrieve data from the source database, R2RML mappings use *logical tables*, which can be a base table, a view, or even an arbitrary valid SQL query, known as "R2RML view". Consecutively, each logical table is mapped to RDF using *triple maps*. A *triples map* specifies the rules for translating each row of a *logical table* to zero or more RDF triples[d] and is represented by a resource that references the following other resources: *rr:logicalTable property* (logical table as value), *rr:subjectMap* (rules for generating subjects for each row) and *rr:predicateObjectMap properties* (rules that specify pairs of predicate maps and object maps). Fig. 2 below provides a schematic overview of the basic R2RML concepts.

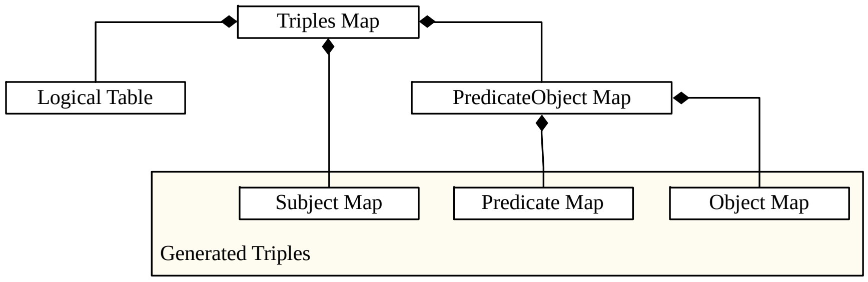


Fig. 2. Simplified R2RML overview.

For example, the triples map `map:persons` (see Fig. 3 below and Fig. 5 for an instance of the database schema), translates every tuple of the `eperson` relation to an instance of the `foaf:Person` class, using the value of the `eperson_id` attribute in the generated URI template. Furthermore, a literal value for the `foaf:name` property is generated, by combining the values of the `firstname` and `lastname` attributes.

```
map:persons
    rr:logicalTable [ rr:tableName '"eperson"'; ];
    rr:subjectMap [
        rr:template 'http://data.example.org/repository/person/{"eperson_id"}';
        rr:class foaf:Person; ];
    rr:predicateObjectMap [
      rr:predicate foaf:name;
      rr:objectMap [ rr:template '{"firstname"} {"lastname"}' ;
                      rr:termType rr:Literal; ] ].
```

[b] R2RML, RDB to RDF Mapping Language : http://www.w3.org/TR/r2rml/

[c] SPARQL Query Language for RDF: http://www.w3.org/TR/rdf-sparql-query/

[d] Definition of a Triples Map: http://www.w3.org/TR/r2rml/#dfn-triples-map

Fig. 3. Triples map example.

Another important concept to our work is the *mapping document*. The *mapping document* is an RDF document, written in Turtle (*Terse RDF Triple Language*[e]) RDF syntax containing a set of *triples maps*, providing instructions about how to convert the source relational database contents into RDF.

In a nutshell, a mapping document can be seen as a set of triples maps: $map = \{triplesMap_i\}$ and each triples map can roughly be viewed as tuple consisting of a logical table, a subject map and a set of predicate-object maps: $triplesMap = \langle table, subjMap, predObjMaps \rangle$. According to the R2RML specification, every logical table has a corresponding effective SQL query, which retrieves the appropriate result sets from a database instance. Therefore, we define the `efsql` function, which maps logical tables to their effective SQL query and we refer to the result set that originates from the execution of such a query $efsql(table)$ over a database instance DB_I as $efsql(table)(DB_I)$.

3. Related Work

Numerous approaches have been proposed in the literature, mainly concerning the creation and maintenance of mappings between RDF and relational databases. Mapping relational databases to RDF graphs and ontologies is a domain where much work has been conducted and several approaches have been proposed[4,5,6,7,8]. Typically, the goal is to describe the relational database contents using an RDF graph (or an ontology) in a way that allows queries submitted to the RDF graph to be answered with data stored in the relational database. Also, for transporting data residing in relational databases into the Semantic Web, many automated or semi-automated mechanisms for ontology instantiation have been created[9,10,11,12].

Related software tools can be broadly divided into two major categories: the ones that allow real-time translation from relational database contents into RDF (transient RDF views) and the ones that allow exports (persistent RDF views).

Many approaches exist where transient RDF views are offered on top of relational databases, putting effort into conceiving efficient algorithms that translate SPARQL queries over the RDF graph in semantically equivalent SQL ones that are executed over the underlying relational database instance[13,14]. Evaluation results show that under certain conditions, some SPARQL-to-SQL rewriting engines (e.g. D2RQ[16] or Virtuoso RDF Views[17,18]) perform faster than native triple stores in query answering, achieving lower query response times[15]. In cases when this happens, it is mostly attributed to two reasons: The first one is the maturity and optimization strategies existing in relational database systems that outperform triple stores, while the second one is more fundamental and lies in the combination of the RDF data model and the structure of the (relatively homogenous) benchmark dataset that was used in the mentioned experiment[15].

Query-driven approaches provide access to an RDF graph that is implied and does not exist in physical form (transient approach). In this case, the RDF graph is virtual and only partially instantiated when some appropriate request is made, usually in the shape of a semantic query. Tools of this category include Virtuoso RDF Views[17], D2RQ and Ontop[11]. In this category of approaches, queries using SPARQL are consequently translated into SQL queries. Such systems are used to publish a so-called transient RDF graph on top of a relational database.

Asynchronous, ad hoc dumps, performed by tools that can materialize an RDF graph based on the contents of a relational database instance can be classified according to specific criteria to a number of categories[20]. Batch-transformation or, equivalently, Extract-Transform-Load (ETL) approaches generate a new RDF graph from a relational database instance[21,22] and store it physically in an external storage medium, such as a triplestore. This approach is called materialized or persistent[2] and does not provide or maintain any means of automatic update of the output (as in the approach by Vavliakis et al.[12]), but requires a mapping file, with the use of which, it is made possible to obtain a snapshot of the relational database contents and export it as an RDF graph. The option of dumping relational database contents into RDF is also supported by D2RQ (alongside its main function as a SPARQL endpoint), Triplify[22], and also the Virtuoso universal server.
The authors' previous work[21] comprises an approach that, in contexts where data is not updated frequently, performs much faster in RDF-izing relational database contents, compared to translating SPARQL queries to SQL in real-time[2].

This approach was further modified and enhanced and is applied in this paper, in order to support incremental RDF dumps for our evaluation through experimentation.

Less work has been conducted as far as it concerns *incremental* RDF generation techniques. Vidal et al.[23] introduce an approach for the incremental, rule-based generation of RDF views over relational data. The paper presents an incremental maintenance strategy, based on rules, for RDF views defined on top of relational data. Unfortunately, an implemented working solution based on the theoretical approach is not currently available, therefore leaving our system as the sole implementation supporting incremental RDF dumps from relational database instances. Finally, the AWETO RDF storage system[24,25] supports both querying and incremental updates, following a hash-based approach in order to perform incremental updates, and constitutes an approach that targets RDF storage and not transformation as in the hereby presented work.

4. Proposed Approach

[e] Turtle, Terse RDF Triple Language: http://www.w3.org/TR/turtle/

We have already noted in Section 3 that, in contexts when the relational dataset is relatively stable and updates are scarce, the application of an ETL-style or persistent RDF generation scheme can be advantageous to the application of dynamic rewriting approaches. In such cases, the performance difference among the two can be attributed to the additional query rewriting time that is needed for the latter category as well as the possibility of translating SPARQL queries into expensive SQL ones involving columns for which appropriate indexes have not been defined.

However, one issue that needs to be taken care of when applying a persistent RDF generation approach is the synchronization of the relational instance with the generated RDF graph. In other words, the RDF generation task has to be combined with an efficient procedure that updates appropriately the already constructed RDF graph in case an update in the contents of the relational database occurs or part of the mapping changes. The presence of such a procedure guarantees that, if not needed, the entire RDF graph will not be recomputed from scratch, involving the execution of several SQL queries on the underlying database.

The basic information flow in the proposed approach has the relational database as a starting point and an RDF graph as the result. The basic components are: the source relational database, the R2RML Parser tool[f], and the resulting RDF graph. Fig. 4 illustrates how this flow in information processing takes place.

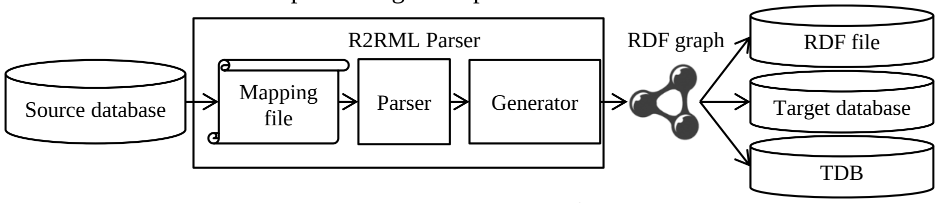


Fig. 4. Overall architectural overview of the approach.

First, database contents are parsed into result sets. Then, according to the mapping file, defined in R2RML, the Parser component generates a set of instructions (i.e. a Java object) for the Generator component. Subsequently, the Generator component, based on this set of instructions, instantiates in-memory the resulting RDF graph. Next, the generated RDF graph is persisted, which can be an RDF file on the hard disk, a target (relational) database, or TDB, Jena's[26] custom implementation of threaded B+ Trees[g].

At this point, it is interesting to describe TDB briefly. The TDB (Tuple Data Base) engine works in tuples, with RDF triples being a special case. Technically, a dataset backed by TDB is stored in a single directory in the file system. A dataset comprises of the node table, triple and quad indexes, and a table with the prefixes. Jena's implementation of B+ Trees only provides for fixed length key and fixed length value, and there is no use of the value part in triple indexes. Because of the custom implementation, it performs faster than a relational database backend, allowing the implementation to scale much more, as it is demonstrated in the performance measurements in Section 5.

In order to produce RDF content incrementally, we can distinguish the following two cases, based on the two problems we identified in Section 1:

A. **Incremental *transformation***: This is possible when the resulting RDF graph is persisted on the hard disk. In this approach, the algorithm that produces the resulting graph does not run over the entire set of mapping document declarations. This is realized by consulting the log file with the output of a previous run of the algorithm, and performing transformations only on the changed data subset. In this case, the resulting RDF graph file is erased and rewritten on the hard disk. The mentioned log file is based on the notion of a *triples map's identity*, which will be introduced shortly.
B. **Incremental *storage***: This is an approach that is only possible in cases when the resulting graph is persisted in an RDF store, in our case using Jena's TDB implementation. Only when the output medium allows additions/deletions/modifications at the level of triples, it is made possible to store the changes without rewriting the whole graph.

The overall generation time can be thought of as the sum of the following time components:

- t1: The mapping document is parsed
- t2: The Jena model is generated in-memory. This is considered to be a discrete step since, at least in theory, upon termination of this step, the model is available to APIs that could as well belong to third party applications.
- t3: The model is dumped to the destination medium.
- t4: The results are logged. In the incremental transformation case, the log file contains the so-called *identities* of the mapping document's triples maps, as well as the *source* of every RDF triple generated.

[f] The R2RML Parser tool: http://www.w3.org/2001/sw/wiki/R2RML_Parser

[g] TDB Architecture: http://jena.apache.org/documentation/tdb/architecture.html

The time measured in the experiments is the sum of t_1, t_2, and t_3. The time needed to log the results, t_4, is not included in the measurements as the output of the incremental generation is available to third party applications immediately after t_3.

In the incremental RDF triple generation, the basic challenge lies in discovering which mapping definitions were added, deleted, and/or modified, and also which database tuples were added, deleted, and/or modified since the last time the incremental RDF generation took place, and perform the mapping only for this altered subset. This means that it is required, for each generated triple, to store annotation information regarding its provenance. This is the core idea in the case of incremental *transformation*.

Ideally, the exact database cell and mapping definition that led to the specific triple generation should be stored. However, using R2RML, the atom of the mapping definition becomes the triples map. Therefore, when annotating a generated triple with the mapping definition that generated it, we can inspect at subsequent executions both the triples map elements (e.g. subject template), as well as the dataset from the database, in order to assert whether the data are changed or not.

Consider, for instance, the triples map `map:persons` (see Fig. 4). In this case, when one of the source tuples change (i.e. the table `eperson` appears to be modified), then the whole triples map definition will be executed. This execution would also be triggered in case the triples map definition had any changes.

In order to detect changes in the source dataset or the mapping definition itself, the proposed approach utilizes hashes for the information of interest. The algorithm that performs the incremental RDF graph generation is presented in Algorithm 1. The hashes were produced using the MD5 cryptographic hash function.

As a result, the hashes that are stored in the log file include: the source logical table SQL `SELECT` query, the respective result set that is retrieved from the source database, and the whole triples map definition itself. These are the components of a triples map's *identity*: for a triples map $triplesMap = \langle table, subjMap, predObjMaps \rangle$, its identity is defined as a tuple $\langle table\_id, triplesMap\_id, resultSet\_id \rangle$, where $table\_id = hash(table)$, $triplesMap\_id = hash(triplesMap)$, $resultSet\_id = hash(efsql(table)(DB_I))$ and hash is a hash string function. We also define the function id, which maps a triples map to its identity. A triples map's identity provides a way to trace for modifications in a triples map, which alter the resulting RDF graph.

In Algorithm 1, we also refer to the function source, which maps every generated RDF triple with the triples map that is responsible for its creation and function generateRDF, which maps a triples map to an RDF graph, following the process illustrated in the R2RML specification.

```
Input: R2RML mapping document map, database instance $DB_I$, input RDF graph graph={$triple_i$}
Output: updated RDF graph graph
for triples_map ∈ map
   if hash(map.table)) != id(map).table_id or
       hash(efsql(map.table)($DB_I$)) != id(map).resultSet_id or
       hash(map) != id(map).triplesMap_id
   then
       graph = graph - {$triple_i$: source($triple_i$)=map}
       graph = graph ∪ generateRDF(map)
   end if
   id(map).table_id =  hash(map.table)
   id(map).resultSet_id = hash(efsql(map.table)($DB_I$)
   id(map).triplesMap_id = hash(map)
end for
```

Algorithm 1. In incremental RDF generation, mapping definitions will be processed only when changes are detected in the queries, their result sets, or the mapping definitions themselves.

In order to create a unique hash over a result set, and subsequently detect whether changes were performed on it or not, Algorithm 2 was devised. It is noteworthy to mention that in order to ensure that the order of the results would be the same, in cases when an `ORDER BY` construct was not present, it was explicitly added programmatically, in order to sort the result set by the first column of the logical table. If the query was ordered beforehand, it was left intact.

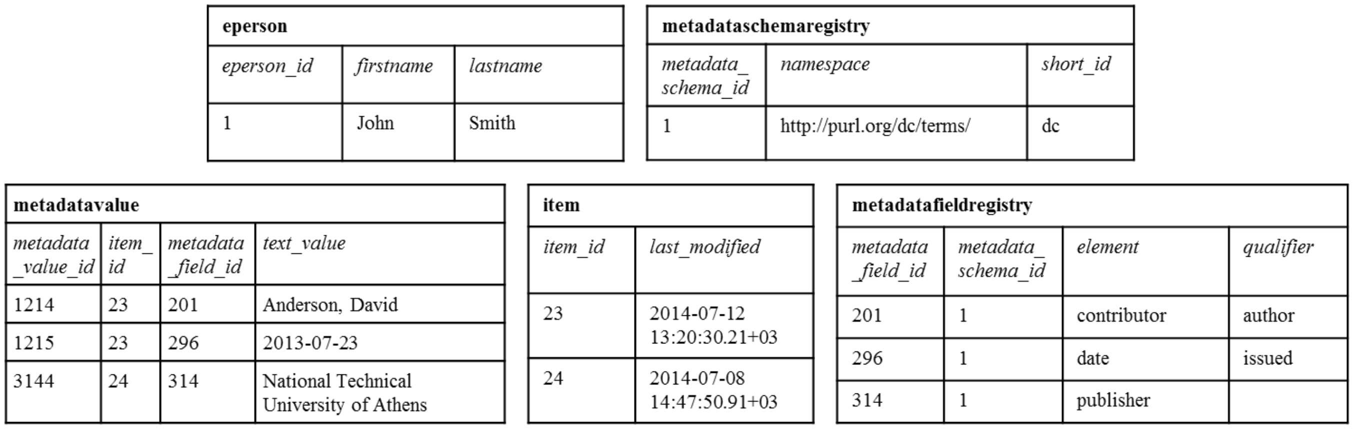
```
Input: a result set result_set
Output: string hash
for row ∈ result_set do
   for column ∈ row do
      hash = concatenate(hash, column as string)
   end
   hash = MD5(hash)
end
```

Algorithm 2. Hash a result set from the source relational database.

Next, in order to verify that no changes were performed on the triples map definitions themselves, they were hashed in order to allow subsequent checks for modifications. For each triples map definition, the input string for the hash contained: the SQL selection query, the subject template, the Class URIs of which the subject was an instance, the predicate-object map templates and/or columns, the predicates, and finally, the parent triples map, if present.

In the case of incremental storage, as the output is persisted at a relational database-backed triplestore or at a Jena TDB, no hashes are needed. Instead, the resulting RDF graph is generated and updates to the existing RDF graph are translated into commands to the dataset (such as SQL `DELETE` and `INSERT`). For convenience, an RDF graph can be viewed as a set of *triples*. Algorithm 3 describes the algorithm used in case the final output is persisted in an RDF store. The goal of this algorithm is to compare the RDF graph that has already been materialized in an earlier point in time with the RDF graph that corresponds to the current database contents and update accordingly the former in order to contain the updates of the latter. We use the $G_1 \setminus G_2$ and $G_1 \cup G_2$ to denote the difference and union of two graphs respectively. After the execution of Algorithm 3, the existing graph will be updated accordingly in order to become equal to the newly computed graph.

In order to give a concrete example of application of our approach, we consider a real scenario based on the DSpace software, a popular open source solution for institutional repositories. A part of the database instance of a DSpace installation is shown in Fig. 5.



eperson		
eperson_id	*firstname*	*lastname*
1	John	Smith

metadataschemaregistry		
metadata_schema_id	*namespace*	*short_id*
1	http://purl.org/dc/terms/	dc

metadatavalue			
metadata_value_id	*item_id*	*metadata_field_id*	*text_value*
1214	23	201	Anderson, David
1215	23	296	2013-07-23
3144	24	314	National Technical University of Athens

item	
item_id	*last_modified*
23	2014-07-12 13:20:30.21+03
24	2014-07-08 14:47:50.91+03

metadatafieldregistry			
metadata_field_id	*metadata_schema_id*	*element*	*qualifier*
201	1	contributor	author
296	1	date	issued
314	1	publisher	

Fig. 5. An example database instance.

We also define an R2RML mapping that comprises a set of TriplesMaps: one of them is shown in Fig. 3 and references just the `eperson` table, while the rest of the triples maps (`map:dc-contributor-author`, `map:dc-date-issued` and `map:dc-publisher` as shown in the Appendix) combine data from the other 4 relations. When the materialization of the RDF graph implied by the relational instance and the R2RML mapping takes place for the first time, the following RDF triples are generated:

```
(T1a) <http://data.example.org/repository/person/1> a foaf:Person;
(T1b)  foaf:name "John Smith".

(T2a) <http://data.example.org/repository/item/23> a dcterms:BibliographicResource;
(T2b)  dc:contributor "Anderson, David";
(T2c)  dcterms:issued "2013-07-23".

(T3a) <http://data.example.org/repository/item/24> a dcterms:BibliographicResource;
(T3b)  dc:publisher "National Technical University of Athens".
```

Fig. 6. RDF triples generated from the relational instance of Fig. 5.

Furthermore, a reified graph is also produced that annotates every generated RDF triple with the source triples map that is responsible for its production. It should be noted here that, alternatively, the source of a triple could be denoted as the graph element of a quad. However, such a choice would incur possible creation of several triples in more than one graph: the one specified in the R2RML mapping document and the one that is implied by the source of the triple. This would, in turn, have implications in the implementation when removing a triple (i.e. it would have to be removed from all graphs it is part of), it would increase considerably the size needed for storage and also, it would pollute the generated dataset with graphs that exist purely for administrative purposes, in this case, for the efficient update of an RDF dataset.

Returning to our example, triples T1a and T1b have as source the `map:persons` triples map (see Fig. 3), T2a has both `map:dc-contributor-author` and `map:dc-date-issued` as sources, T2b the `map:dc-contributor-author` triples map and so on. Knowledge of the source of generated triples is necessary for finding out the triples that should be substituted when re-applying a specific triples map to a relational instance.

Suppose now that the record (2, Susan, Johnson) is added to the `eperson` relation which, according to the `map:persons` triples map, will give rise to the following additional RDF statements:

```
(T1c) <http://data.example.org/repository/person/2> a foaf:Person;
(T1d)  foaf:name  "Susan Johnson".
```

Fig. 7. Additional RDF triples after insertion in the relational instance.

Following Algorithm 1, the modification in the relational instance is detected when comparing the updated result set that corresponds to the `map:persons` triples map (see Fig. 3) with the respective result set that held during the first execution of the R2RML mapping. Since no other modification is detected in either the relational instance or the R2RML mapping, Algorithm 1 only considers the `map:persons` triples map for the incremental update of the materialized RDF graph. All RDF triples that have originated from the `map:persons` triples map are dropped, given that there is no information available on the type of modification occurred on the relational instance. Therefore, triples T1a and T1b are generated again during the update of the RDF graph, along with new triples T1c and T1d.

The same procedure takes place when there is a deletion or modification in the relational instance. The triples maps that are affected by the change are the ones that are going to be solely taken into account during the incremental update of the materialized RDF graph.

The previously described procedure also applies in cases of modification of the R2RML mapping. As expected, when a new triples map is added to the R2RML mapping, it suffices to simply examine the newly added triples map and add the RDF triples that are derived from it to the materialized RDF graph. In case an existing triples map is modified or deleted, then the RDF triples that have emanated from that triples map are deleted and new triples that are derived from the modified triples map are added. Suppose, for example, that the `map:persons` triples map is substituted by the `map:persons-new` one, which is shown in the Appendix. Initially, triples T1a-T1d will be removed from the RDF graph and triples T1a'-T1f' will be added.

```
(T1a') <http://data.example.org/repository/person/1> a foaf:Person;
(T1b') foaf:firstName "John";
(T1c')  foaf:lastName "Smith".
(T1d') <http://data.example.org/repository/person/2> a foaf:Person;
(T1e') foaf:firstName "Susan";
(T1f')  foaf:lastName "Johnson".
```

Fig. 8. Additional RDF triples after modification of the R2RML mapping.

For the interested reader, the source code of the implementation that served as the basis for our experiments is available online at http://www.github.com/nkons/r2rml-parser.

5. Measurements

This Section provides information regarding the environment setup, the evaluation results and a discussion over our findings.

5.1 Measurements Setup

Using the popular open-source institutional repository software solution DSpace (dspace.org), seven installations were made and their relational database backends were populated with synthetic data, comprising 1k, 5k, 10k, 50k, 100k, 500k, 1m items (the metadata of each stored as a row in the `item` table), respectively. The data was created using a random string generator in Java: each randomly generated item was set to contain between 5 and 30 metadata fields from the Dublin Core (DC) vocabulary, with random text values ranging from 2 to 50 text characters. Next, several mapping files were considered for our tests.

```
Input: RDF graph new_graph, RDF graph existing_graph
Output: An updated existing_graph

statements_to_remove ← existing_graph\new_graph
statements_to_add ← new_graph\existing_graph

existing_graph ← existing_graph\statements_to_remove
existing_graph ← existing_graph ∪ statements_to_add

return existing_graph
```

Algorithm 3. Incrementally dump the resulting model to a Jena TDB backend.

The first set of mapping definitions, targeting the contents of the repository, comprised "complicated" SQL queries (including `JOIN`s among 4 tables), as the one presented below:

```
Q1: SELECT i.item_id AS item_id, mv.text_value AS text_value
FROM item AS i, metadatavalue AS mv, metadataschemaregistry
AS msr, metadatafieldregistry AS mfr WHERE
msr.metadata_schema_id=mfr.metadata_schema_id AND
mfr.metadata_field_id=mv.metadata_field_id AND
mv.text_value is not null AND
i.item_id=mv.item_id AND
msr.namespace='http://dublincore.org/documents/dcmi-terms/'
AND mfr.element='coverage'
AND mfr.qualifier='spatial'
```

Inspection of the execution plan of Q1 reveals 3 inner joins and 4 filter conditions that do not involve any indexed columns, leading up to a total cost of 28.32[h], according to PostgreSQL's query engine. Q1 was later simplified, by removing one of the `JOIN` conditions and two of the `WHERE` conditions, thus reducing its complexity and becoming:

```
Q2: SELECT i.item_id AS item_id, mv.text_value AS text_value
FROM item AS i, metadatavalue AS mv,
metadatafieldregistry AS mfr WHERE
mfr.metadata_field_id=mv.metadata_field_id AND
i.item_id=mv.item_id AND
mfr.element='coverage' AND
mfr.qualifier='spatial'
```

Indeed, the cost of Q2 is found to be 21.29, the most costly operation being the `metadatavalue` and `metadatafieldregistry` join. Another, even simpler mapping definition was also considered, comprising a simple query without any `JOIN` conditions, as follows:

```
Q3: SELECT "language", "netid", "phone", "sub_frequency","last_active", "self_registered",
"require_certificate", "can_log_in", "lastname", "firstname", "digest_algorithm", "salt",
"password", "email", "eperson_id"
FROM "eperson" ORDER BY "language"
```

Q3 proves to be the simplest query of all, with a cost of 12.52. Measurements were performed on machine with a dual-core 2GHz processor, with 4096MB RAM and 40GB hard disk. The machine was running Ubuntu Linux, Oracle Java 1.7.0_45 and PostgreSQL version 9.1.9. We note that several experiments were conducted initially using the OpenJDK 64-Bit Java version, which led, however, to frequent Out of Memory issues, forcing us to switch to Oracle's implementation. It also has to be noted, that, in order to deal with database caching effects, the queries were run several times, in order to get the system "warmed up", prior to performing our measurements.

5.2 Measurements Results

In Figures 9 to 15, the y-axis measures execution time in seconds, while the letters in the x-axis correspond to the following cases: n: non-incremental mapping transformation, a: incremental, for the initial time, b, c, d, e, f, g: incremental transformation, where the ratio of the changed triples over the total amount of triples in the resulting graph is 0, 1/12, 1/4, 1/2, 3/4, and 1, respectively.

5.2.1 Exporting to an RDF File

The first set of performance measurements was realized using an output file. In Fig. 9, the triples maps contained simple SQL queries (Q3 in Section 5.1). We note that the time needed for the initial export of the database contents into an RDF graph increases as the number of the triples in the resulting model increases. Incremental dumps take more time than non-incremental, since the reified model also has to be created and stored, as analyzed in Section 4, a sacrifice in performance that pays off in subsequent executions. Similar results were obtained for mappings containing simple SQL queries resulting in graphs containing up to 300k triples. Similar results are observed in Fig. 10, regarding the more complicated SQL queries (Q1 in Section 5.1), the difference however is not as great as in the previous case.

Next, in Fig. 11, the same tests were performed in the complicated SQL query case, showing that the results here were very good. As Fig. 11 shows, the time required to dump database contents non-incrementally was less than the initial incremental dump. Consecutive dumps, however, took less than the initial time, practically no time when no changes were

[h] PostgreSQL's planner estimates the cost of queries in arbitrary units that roughly correspond to disk page fetches.

detected in the source database, and performing faster even in the case when the percentage of the initial mapping that changed reached to 75%. Only when the vast majority of the mapping was changed, did the incremental dump take longer than the non-incremental one.

Next, we considered the effect that the query simplification would have on the result. In Fig. 12, we demonstrate a set of measurements with all parameters being the same, except for the query itself, which was similar to Q2 in Section 5.1. Query simplification did have an impact on the resulting time, but a rather small one.

In order to verify how the number of triples of the resulting RDF graph affects the performance, we added 6 triples map definitions, similar to the existing ones, thus producing 6 times as much triples. This allowed us to increase the size of the produced graph up to 3 million triples, the behavior however remained the same. This behavior is demonstrated in Fig. 13: At all cases, the initial, non-incremental dump time took less than every attempt to dump the relational database contents incrementally.

5.2.2 Exporting to a Relational Database

The next set of measurements was performed using the relational database as the output medium. Our experiments revealed that database behavior compared to hard disk behavior gave dissatisfactory performance times at all cases, regardless to the number of the triples in the resulting graph. Comparative performance is illustrated in Figures 14 and 15, showing that the relational database as a backend performed more poorly than the hard disk or TDB, which justifies Jena developers' decision to drop support for a relational database in favor of TDB as an RDF backend.

5.2.3 Exporting to Jena TDB

The next set of experiments included incremental transformation and storage using TDB. As the experiments showed, its behavior is similar to the one of the database backend, but with a much faster performance, at all sizes of triples in the resulting graph. For instance, as Fig. 14 shows, the results for the complicated mappings, in cases when each of the initial SQL queries returns 10k rows using TDB is much faster than using a relational database. Still, however, storing on the hard disk is the fastest solution. The next figure, Fig. 15, compares results in the database and results in TDB. In this case, however, it was not possible to obtain result output on the hard disk, or on the relational database in cases where changes in the initial model had taken place, because of out-of-memory errors.

The TDB behavior in the case of simple mappings is similar to the one with the database backend, only the performance is much faster. The same holds in the case of more resulting triples, as illustrated in Fig. 16, in which the resulting triples are 150k and 300k. The values of the horizontal axis in Fig. 16 correspond to the following cases: n: non incremental, a': incremental – initial, b': 0% change, c': 33% change, d': 66% change, e': 100% change.

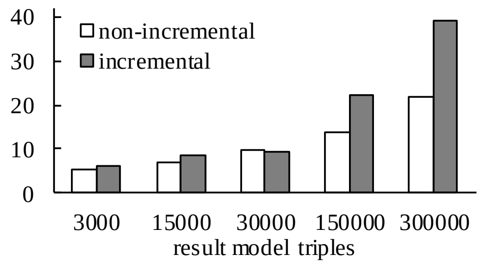


Fig. 9. Time needed for the initial export of database contents into an RDF file using simple queries in mappings.

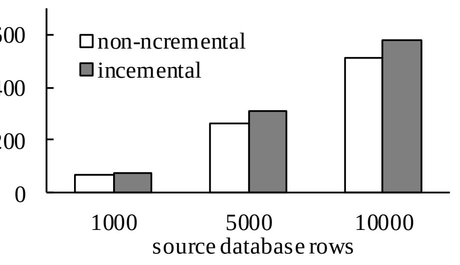


Fig. 10. Time needed for the initial export of database contents into an RDF file, using complicated queries in mappings.

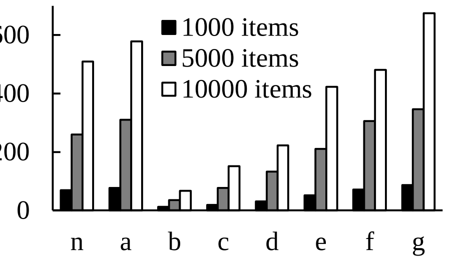


Fig. 11. Incremental dumps in the case of complicated queries outperform non-incremental dumps for changes on approximately 3/4 of the mapping.

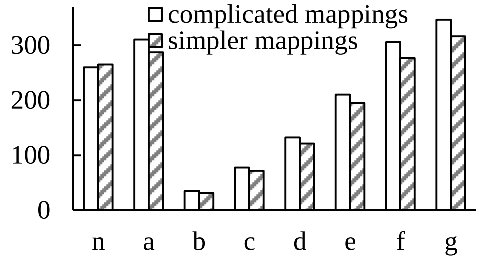


Fig. 12. Incremental dumps it the case of complicated mapping queries (`JOIN`s among 4 tables) and simplified queries (`JOIN`s among 3 tables) with the same results.

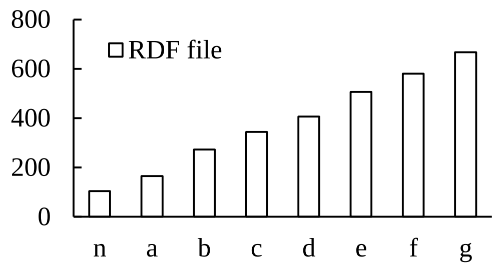


Fig. 13. Consecutive, incremental dumps in the case of simple queries.

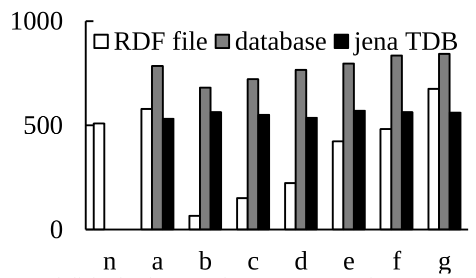


Fig. 14. Hard disk, database, and Jena's TDB as the output medium. Complicated mappings, 10k items in the source database, approx. 180k triples in the resulting model.

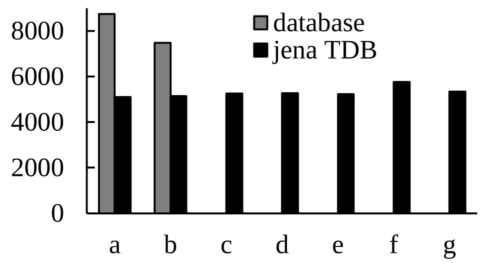


Fig. 15. Database vs TDB behavior: Complicated mappings, 100k items in the source database, approximately 1.8 million triples in the resulting model.

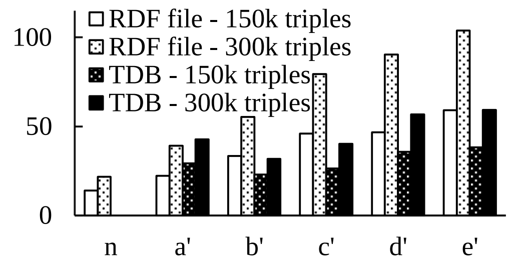


Fig. 16. TDB behavior: Simple mappings.

6. Discussion

First, it has to be noted that logging the provenance of each triple in case of file output has a severe impact on the size required to store the logged output. Logging, which takes five times as much space as the resulting RDF model, improves performance for small to average datasets up to several hundreds of thousands of triples. The upper bound that will be defined by the system's RAM will have to contain the logged reified model. Therefore, models that will be produced will not be able to scale as much. For larger models, however, TDB is preferred, mostly because of the more relaxed constraints in terms of memory, due to the absence of the logging mechanism. Thus, while persisting the final model into TDB takes longer than outputting it to a file, this does not raise memory usage to the level of throwing out-of-memory exceptions, allowing the result to scale more. In our experiments, the TDB-backed triplestore scaled up to 1.8 million triples. Of course, TDB scales more, but for our measurements no more tests were needed to demonstrate the approach's behavior according to the storage mechanism.

The fact that consecutive executions take less time than the initial time is largely due to the fact that the unchanged result graph subsets are not re-generated; only the respective SQL queries are executed but the queries' results are not iterated upon in order to produce triples. Moreover, relying upon the logged hashes of previous executions allows the result to be compared to the previous one (previous dump) without having to load both models in memory, enabling further scaling of the approach.

Notably, the RDF serialization format affected performance. Several serializations, such as RDF/XML or TTL try to pretty-print the result, consuming extra resources. Because of this, the serialization format that was adopted in our tests, in the cases when the resulting RDF had to be output to hard disk – also in creating the reified model – the "N-TRIPLES" syntax was used.

An alternative to the hereby proposed approach would be to replace reified triples with named graphs, each one containing a statement, and allowing properties to be asserted on it. Such an implementation could support the use cases of reification without requiring a data transformation.

Also, since there are various tools that allow direct, synchronous RDB2RDF mappings, one could argue whether the whole approach is too complex and not worth the implementation overhead. However, the task of exposing database contents as RDF could be considered similar to the task of maintaining search indexes next to text content: data freshness can be sacrificed in order to obtain much faster results[2].

Although this approach could be followed in many domains, it finds ideal application in cases when data is not often (e.g. daily) updated to a significant amount, for instance in the bibliographic domain[21]. In these cases, when freshness is not crucial and/or selection queries over the contents are more frequent than the updates, what our approach succeeds in, is a lower downtime of the system that hosts the generated RDF graph and serves query-answering.

Once the RDF dump is exported, a software system could operate completely and reason based on the semantically enriched information. This approach could be materialized using for instance Jena's Fuseki SPARQL server, the Sesame framework (www.openrdf.org), Virtuoso, or even native RDF support that can be found in modern RDBMS's such as Oracle. Adoption of such approaches enables the custom implementation of systems that allow accessing and operating on the RDF graph, after hosting it on a triplestore server. Materialization of the RDF dumps could in general be part of larger systems using a push or a pull approach to propagate the changes from the database to the triplestore.

Also noteworthy is the fact that still, exporting data as RDF covers half of the requirements that have to be met before publishing datasets as RDF. The other half of the problem concerns the semantics that are infused in the data through the use of terms from RDFS or OWL ontologies: the hereby introduced methodology guarantees only the syntactic validity of the result, without any checks on the semantic matching among the input database instance and the target RDF graph. Meanwhile, it has to be underlined that caution is required in producing de-referenceable URIs. Mistakes can easily go unnoticed, even if syntactically everything is correct.

7. Conclusions and Future Work

Overall, this paper's contribution is a study of the incremental RDF generation and storage problem, in addition to proposing an approach and assessing its performance after modifying several problem parameters. Exporting relational database contents as RDF graphs, and thus bringing their contents into the Semantic Web, opens a world of opportunities, non-existent in relational databases. These include reasoning, integration with third party datasets, allowing more intelligent queries (using SPARQL instead of SQL), to name a few. As such, the proposed approach can be considered as an enabler for AI applications from the Semantic Web domain to consume relational database data.

Our measurements indicate that the proposed approach performs optimally when the triples mappings contain complicated SQL queries, and the resulting RDF graph is persisted on a file. In these cases, despite the increase in the time required for the initial (incremental) dump, subsequent dumps are performed much faster, especially in cases when the changes affect less than the half of the mapping. However, for graphs containing millions of triples, storing in TDB is the optimal solution, since storing in the hard disk was limited by the physical memory. During our tests we were struggling with out-of-memory errors, since, in incrementally generating RDF, both the resulting model and the reified had to be loaded in-memory before outputting to the target medium.

Several directions exist towards which this work could be expanded. These could include the investigation of two-way updates: in the same manner in which updates in the database are reflected on the triplestore, updates on the triplestore could be sent back to the database. This could be made possible by keeping the source mapping definition at the target, as annotation on the reified statement. Furthermore, we intend to extend our algorithms in order to take into account scenarios when reasoning is applied on the target RDF graph. In such a case, the issue of updating the materialized graph becomes more involved, as triples that are inferred from other ones need to be updated accordingly as well. Future work could also include further studying the impact of the complexity of the SQL query on generating the result. In order to study this, the total time needed to execute the SQL queries that retrieve the data from the source database could be measured. Last, statement annotations to include other annotations as well, e.g. geo-tagging, timestamps, or even database connection parameters instead of only mapping definition provenance, thus allowing incremental RDF generation from multiple database schemas in a variety of scenarios, in the same manner.

Acknowledgements

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF).

References

1. Villazon-Terrazas, B., Vila-Suero, D., Garijo, D., Vilches-Blazquez, L.M., Poveda-Villalon, M., Mora, J., Corcho, O., Gomez-Perez, A. 2012. Publishing Linked Data - There is no One-Size-Fits-All Formula. In *Proceedings of the European Data Forum*
2. Konstantinou, N, Spanos, D.E., Mitrou, N. 2013. Transient and Persistent RDF Views over Relational Databases in the Context of Digital Repositories. In *Proceedings of the 7th Metadata and Semantics Research Conference (MTSR'13)*, Thessaloniki, Greece
3. Dougherty, E., Laplante, P. 1995. *Introduction to Real-Time Imaging*, chapter What is Real-Time Processing?, pp. 1-9. Wiley-IEEE Press
4. Sahoo, S., Halb, W., Hellmann, S., Idehen, K., Thibodeau, T., Auer, S., Sequeda, J., Ezzat, A. 2009. A Survey of Current Approaches for Mapping of Relational Databases to RDF. Available at http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf, accessed on February 28th 2014
5. Konstantinou, N., Spanos, D.E., Mitrou, N. 2008. Ontology and Database Mapping: A Survey of Current Implementations and Future Directions. In *J. Web Engineering*, 7, 1, 1–24
6. Sequeda, J.F., Tirmizi, S.H., Corcho, O., Miranker, D.P. 2009. Direct Mapping SQL Databases to the Semantic Web: A Survey (Technical Report TR-09-04), University of Texas, Austin, Department of Computer Sciences
7. Hellmann, S., Unbehauen, J., Zaveri, A., Lehmann, J., Auer, S., Tramp, S., Williams, H., Erling, O., Thibodeau Jr, T., Idehen, K., Blumauer, A., Nagy, H. 2011. Report on Knowledge Extraction from Structured Sources, LOD2 Project, Deliverable 3.1.1, available at: http://static.lod2.eu/Deliverables/deliverable-3.1.1.pdf
8. Michel, F., Montagnat, J. & Faron-Zucker, C. 2013. A survey of RDB to RDF translation approaches and tools, Technical Report, Universite Nice, Sophia Antipolis, available at: http://hal.inria.fr/hal-00903568
9. Arenas, M., Bertails, A., Prud'hommeaux, E., Sequeda, J. 2012. A Direct Mapping of Relational Data to RDF. W3C Recommendation. Available online at http://www.w3.org/TR/rdb-direct-mapping/
10. Sequeda, J., Arenas, M., Miranker, D. 2012. On Directly Mapping Relational Databases to RDF and OWL. In *Proceedings of the 21st World Wide Web Conference*, Lyon, France
11. Rodriguez-Muro, M., Kontchakov, R., Zakharyaschev, M. 2013. Ontology-Based Data Access: Ontop of Databases. In *Proceedings of the 12th International Semantic Web Conference (ISWC'13)*, Sydney, Australia
12. Vavliakis, K. N., Grollios, T. K., Mitkas, P. A. 2013. RDOTE – Publishing Relational Databases into the Semantic Web. In *Journal of Systems and Software*, 86, 1, 89–99
13. Chebotko, A., Lu, S., and Fotouhi, F. 2009. Semantics Preserving SPARQL-to-SQL Translation. In *Data & Knowledge Engineering*, 68, 10, 973–1000
14. Cyganiak, R. 2005. A Relational Algebra for SPARQL, Technical Report HPL 2005-170
15. Bizer, C., Schultz, A. 2009. The Berlin SPARQL Benchmark. In *International Journal On Semantic Web and Information Systems*, 5, 2, 1–24
16. Bizer, C., Cyganiak, R. 2006. D2R Server - Publishing Relational Databases on the Semantic Web. In *Proceedings of the 5th International Semantic Web Conference*
17. Erling, O., Mikhailov, I. 2007. RDF support in the Virtuoso DBMS. *Proceedings of the 1st Conference on Social Semantic Web*, Leipzig, Germany, 59–68
18. Blakeley, C. 2007. Virtuoso RDF Views Getting Started Guide. Available online at http://www.openlinksw.co.uk/virtuoso/Whitepapers/pdf/Virtuoso_SQL_to_RDF_Mapping.pdf accessed on February 28th 2014
19. Bizer, C., Seaborne, A. 2004. D2RQ—Treating non-RDF databases as virtual RDF graphs. In *Proceedings of the 3rd International Semantic Web Conference*, Hiroshima, Japan
20. Spanos, D.-E., Stavrou, P., Mitrou, N. 2012. Bringing Relational Databases into the Semantic Web: A Survey. In *Semantic Web Journal*, 3, 2, 169-209
21. Konstantinou, N., Spanos, D.-E., Houssos, N., Mitrou, N. 2014. Exposing Scholarly Information as Linked Open Data: RDFizing DSpace contents. In *The Electronic Library*, 32, 6, 834-851
22. Auer, S, Dietzold, S., Lehmann, J., Hellmann, S., Aumueller, D. 2009. Triplify – Light-Weight Linked Data Publication from Relational Databases. In *Proceedings of the 18th international conference on World Wide Web*, New York, NY, USA, 621–630
23. Vidal, V. M. P., Casanova, M. A., Cardoso, D. S. 2013. Incremental Maintenance of RDF Views of Relational Data. In *On the Move to Meaningful Internet Systems (OTM 2013 Conferences)*, Lecture Notes in Computer Science, Volume 8185, 572–587
24. Pu, X., Wang, J., Song, Z., Luo, P., Wang, M. 2014. Efficient incremental update and querying in AWETO RDF storage system. In Data & Knowledge Engineering, 89, 55-75
25. Pu, X., Wang, J., Luo, P., Wang, M. 2011. AWETO: Efficient Incremental Update and Querying in RDF Storage System. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM '11)*, New York, New York, USA

26. Carroll, J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K. 2004. Jena: Implementing the Semantic Web Recommendations. In *Proceedings of the 13th World Wide Web Conference*, New York, New York, USA

Appendix

The following R2RML triples maps definitions re used in the running example of Section 4, Figures 5 to 7. The first three (`map:dc-contributor-author`, `map:dc-date-issued`, and `map:dc-publisher`) follow the same general pattern, each one for a distinct metadata field. The fourth triples map (`map:persons-new`) is used in order to showcase the example in Fig. **8**. Those triples maps are defined as follows:

```
map:dc-contributor-author
    rr:logicalTable [
       rr:sqlQuery """
    SELECT i.item_id AS item_id, mv.text_value AS text_value
    FROM item AS i, metadatavalue AS mv, metadataschemaregistry AS msr, metadatafieldregistry
AS mfr WHERE
    msr.metadata_schema_id=mfr.metadata_schema_id AND
    mfr.metadata_field_id=mv.metadata_field_id AND
    mv.text_value is not null AND
    i.item_id=mv.item_id AND
    msr.namespace='http://purl.org/dc/terms/' AND
    mfr.element='contributor' AND
    mfr.qualifier='author'
    """. ];
    rr:subjectMap [
        rr:template 'http://data.example.org/repository/item/{"item_id"}';
        rr:class dcterms:BibliographicResource; ];
    rr:predicateObjectMap [
        rr:predicate dc:creator;
        rr:objectMap [ rr:column '"text_value"' ]; ].

map:dc-date-issued
    rr:logicalTable [
       rr:sqlQuery """
    SELECT i.item_id AS item_id, mv.text_value AS text_value
    FROM item AS i, metadatavalue AS mv, metadataschemaregistry AS msr, metadatafieldregistry
AS mfr WHERE
    msr.metadata_schema_id=mfr.metadata_schema_id AND
    mfr.metadata_field_id=mv.metadata_field_id AND
    mv.text_value is not null AND
    i.item_id=mv.item_id AND
    msr.namespace='http://purl.org/dc/terms/' AND
    mfr.element='date' AND
    mfr.qualifier='issued'
    """. ];
    rr:subjectMap [
        rr:template 'http://data.example.org/repository/item/{"item_id"}';
        rr:class dcterms:BibliographicResource; ];
    rr:predicateObjectMap [
        rr:predicate dcterms:issued;
        rr:objectMap [ rr:column '"text_value"' ]; ].

map:dc-publisher
    rr:logicalTable [
       rr:sqlQuery """
    SELECT i.item_id AS item_id, mv.text_value AS text_value
    FROM item AS i, metadatavalue AS mv, metadataschemaregistry AS msr, metadatafieldregistry
AS mfr WHERE
    msr.metadata_schema_id=mfr.metadata_schema_id AND
    mfr.metadata_field_id=mv.metadata_field_id AND
    mv.text_value is not null AND
    i.item_id=mv.item_id AND
    msr.namespace='http://purl.org/dc/terms/' AND
    mfr.element='publisher' AND
    mfr.qualifier=''
    """. ];
    rr:subjectMap [
        rr:template 'http://data.example.org/repository/item/{"item_id"}';
        rr:class dcterms:BibliographicResource; ];
    rr:predicateObjectMap [
        rr:predicate dc:publisher;
        rr:objectMap [ rr:column '"text_value"' ]; ].

map:persons-new
    rr:logicalTable [ rr:tableName '"eperson"'; ];
    rr:subjectMap [
        rr:template 'http://data.example.org/repository/person/{"eperson_id"}';
        rr:class foaf:Person; ];
    rr:predicateObjectMap [
      rr:predicate foaf:firstName;
      rr:objectMap [ rr:column "firstname" ;
                     rr:termType rr:Literal; ] ];
    rr:predicateObjectMap [
      rr:predicate foaf:lastName;
      rr:objectMap [ rr:column "lastname" ;
                     rr:termType rr:Literal; ] ];
```