Incremental Export of Relational Database Contents into RDF Graphs

Nikolaos Konstantinou
Hellenic Academic Libraries Link
National Technical University of Athens
Iroon Polytechniou 9, Zografou
15780, Athens, Greece
+30 210 772 4483
nkons@seab.gr

Dimitris Kouis
Hellenic Academic Libraries Link
National Technical University of Athens
Iroon Polytechniou 9, Zografou
15780, Athens, Greece
+30 210 772 4487
dimitriskouis@seab.gr

Nikolas Mitrou
School of Electrical and Computer Engineering, National Technical University of Athens
Iroon Polytechniou 9, Zografou
15780, Athens, Greece
+30 210 772 1639
mitrou@cs.ntua.gr

ABSTRACT
In addition to tools offering RDF views over databases, a variety of tools exist that allow exporting database contents into RDF graphs; tools proven that in many cases demonstrate better performance than the former. However, in cases when database contents are exported into RDF, it is not always optimal or even necessary to dump the whole database contents every time. In this paper, the problem of incremental generation and storage of the resulting RDF graph is investigated. An implementation of the R2RML standard is used in order to express mappings that associate tuples from the source database to triples in the resulting RDF graph. Next, a methodology is proposed that enables incremental generation and storage of an RDF graph based on a source relational database, and it is evaluated through a set of performance measurements. Finally, a discussion is presented regarding the authors' most important findings and conclusions.

Categories and Subject Descriptors
E.2 [**Data Storage Representations**]: Linked representations, H.2.4 [**Systems**] Relational databases

General Terms
Algorithms, Measurement, Performance, Experimentation.

Keywords
Linked Open Data, Incremental, RDF, Relational Databases, Mapping, R2RML.

1. INTRODUCTION
Systems that collect, maintain and update information 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 are produced, assured for their 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.

To this direction, 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 discussed in [2]: Should RDF content generation take place in real-time or should database contents be dumped into RDF asynchronously? 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.

As argued in [2], 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. 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,

WIMS '14, June 02 - 04 2014, Thessaloniki, Greece

http://dx.doi.org/10.1145/2611040.2611082

especially in cases when SPARQL queries involve many triple patterns, which, subsequently translated to SQL queries, 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, not all of the initial information that lies in the database should be transformed into RDF, but only the one that changed.

- Sub-problem #2: *Incremental storage*. This is a problem that is investigated only when the resulting RDF graph is stored in a relational database or a Jena TDB (to be briefly analyzed in Section 3) model. The problem here, regardless to 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, according to [3], the notion of real-time is tightly coupled with the concepts of event and response time. 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 steps 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 provides an overview of the related works in the literature, Section 3 introduces and analyzes the proposed approach, Section 4 describes the measurement environment and parameters, presents the performance measurements and a discussion over the results while Section 5 concludes the paper with our most important observations and future plans to expand on this work.

2. RELATED WORK

Numerous approaches have been proposed in the bibliography, mainly concerning the creation and maintenance of mappings between RDF and databases. Mapping relational databases to RDF or graph databases is a domain where much work has been conducted and several approaches have been proposed [4, 5, 6, 7]. 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 schema 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 schemes instantiation have been created [8,9,10, 11].

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 [12, 13]. Evaluation results, such as the ones presented in [14], show that under certain conditions, some SPARQL-to-SQL rewriting engines (e.g. D2RQ, a solution introduced in [15] or Virtuoso RDF Views [16, 17]) perform faster than native triple stores in query answering, achieving lower query response times. 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 for providing the results [14].

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 (i.e. partially instantiated) and is only considered when some appropriate request is made, usually in the shape of a semantic query. Tools of this category include Virtuoso [16] and D2RQ. In this category of approaches, queries using SPARQL are consequently translated into SQL queries. These systems include D2RQ [13] and the Virtuoso universal server [16], both of which are approaches that support RDF Views over relational database contents making it possible to publish a transient RDF graphs on top of relational databases.

Asynchronous, ad hoc dumps, performed by a category of tools that can generate an RDF dump based on a relational database can be classified according to a number of criteria to a number of categories [19]. Batch-transformation, or Extract-Transform-Load (ETL) approaches generate a new RDF graph from a relational database instance (e.g. [20, 21]) and store physically in an external storage medium, such as a triplestore. This approach is called materialized, or persistent [2]. This approach does not provide or maintain any means of mappings between the source contents and the output (as in [11]), 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 an RDF server), Triplify [21], and also the Virtuoso universal server.

The authors' previous work, introduced in [20], comprises an approach that, in contexts where the data are 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 used in this paper and was

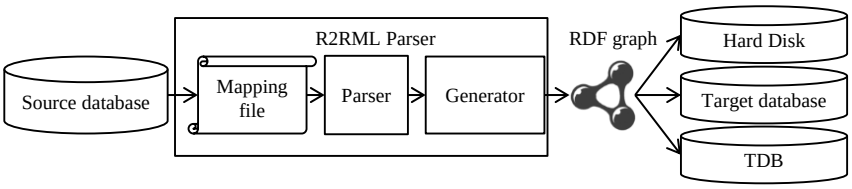

Figure 1: Overall architectural overview of the approach.

further modified and enhanced, 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. In [22], an approach is introduced 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. A comparison with our system could not currently be made as the work presented in [22] is not currently followed by an implementation that could be evaluated/compared to ours, it is mentioned however in the authors' future plans. Finally, in [23] the AWETO RDF storage system is introduced, where both querying and incremental updates are supported, following a hash-based approach in order to perform incremental updates, an approach that targets RDF storage and not transformation as in the hereby presented work.

3. PROPOSED APPROACH

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[1] , and the resulting RDF graph. Figure 1 illustrates how this flow in information processing takes place. First, database contents are parsed into result sets. Then, according to the mapping file, defined in R2RML [24], 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, another relational database, tailored to serve RDF data, or TDB, Jena's [25] custom implementation of threaded B+Trees[2].

While the first two need no further comments, it is interesting to describe TDB briefly. The TDB (Tuple Data Base) engine works in tuples, of which triples are a 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 the 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:

- 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 whole 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.
- Incremental *storage*: This is an approach that is only possible in cases when the resulting graph is persisted in a relational database or 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.

In the course of the experiments, time in each execution is divided in the following consecutive time parts. The time measured in the experiments is the sum of t_1, t_2, and t_3:

- 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 RDF generation case, the log file includes writing the "reified model"
- t5: The program is terminated

In the scope of the hereby presented work, the term *reified model* is introduced, in analogy to the term *reified statement*. *Reification* in RDF is the ability to treat an RDF statement as an RDF resource, and hence to make assertions about that statement. The term *reified model* denotes a model whose statements are all reified, i.e. a model that contains only reified statements. This was made in order to store information about every triple regarding the mapping definition that produced it.

In Figure 2, a materialized example is shown of how standard RDF triples are logged as reified statements, followed by a provenance note. The reified statement in the example is annotated with the mapping declaration that led to its generation, i.e. the triples map definition `map:persons`.

The term *triples map* is also a key term to this work: A *triples map* specifies a rule for translating each row of a *logical table* to zero or more RDF triples[3]. A *logical table* is a tabular SQL query result that is to be mapped to RDF triples. Hence, execution of a *triples map* generates the triples that originate from a specific dataset, i.e. a *logical table*. Figure 3 illustrates an example of a triples map that is assigned the qname `map:persons`.

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

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

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

```
<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 .
```

Figure 2: Annotated reified statement example.

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. Thus 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.

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

Figure 3. Triples maps are the atoms of the mapping document.

Consider, for instance, the triples map in Figure 3. 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.

The basic concepts involved here are the *mapping document*, the *triples map*, and the *logical table*. A *mapping document* is an RDF document in R2ML containing *triples maps*, containing instructions about how to convert the source relational database contents into RDF. Each of these *triples maps*, similar to the one in Figure 3, contains a *logical table* that, in its turn, contains an SQL *select query*, which has a respective *select query result set*, with the tuples retrieved from the database. As a result, the hashes that are stored in the log file include: the source logical table SQL `SELECT` query, the respective dataset that is retrieved from the source database, and the whole triples map definition itself.

```
Input: RDF mapping document in R2RML
Output: RDF triples
for triples map ∈ mapping document
   if select query hash != logged select query
hash or
      resultset hash != logged resultset hash or
      triples map hash != logged triples map hash
   then
      produce triples, by executing mapping
instructions for triples map
   end if
end for
write MD5(select query) to log
write MD5(select query resultset) to log
write MD5(triples map) to log
```

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 added programmatically, according to the first field in the 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`). The algorithm in this case is different, as shown in Algorithm 3.

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

4. MEASUREMENTS

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

4.1 Measurements Setup

Using the popular open-source institutional repository software solution DSpace (dspace.org), seven repositories were constructed and their relational database backends were populated with synthetic data, comprising 1k, 5k, 10k, 50k, 100k, 500k, 1m

items (i.e. rows in the item table), respectively. Several mapping files were considered for our tests. The first set of mapping

```
Input: RDF model new model, RDF model existing
model
Output: An updated RDF model
for triple ∈ new model
   if triple ∉ existing model then
      add triple to list of statements
      to remove
   end
end

for triple ∉ existing model
   add triple to list of statements to add
end

remove list of statements to remove from existing
model
add list of statements to add to existing model
return existing model
```

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

definitions, targeting the contents of the repository, comprised "complicated" SQL queries (including JOINs among 4 tables), as the one presented below:

```
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/dcm
i-terms/'
AND mfr.element='coverage'
AND mfr.qualifier='spatial'
```

This query was later simplified, by removing one of the JOIN conditions and two of the WHERE conditions, thus reducing its complexity and becoming:

```
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'
```

Another, even simpler mapping definition was also considered, comprising a simple query without any JOIN conditions, as follows:

```
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"
```

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, Postgresql version 9.1.9, and Mysql x86_64 version 5.5.32. 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 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.

4.2 Measurements Results

In Figures 4 to 9, the y-axis measures execution time in seconds, while the letters in the x-axis correspond to the following cases: a: non-incremental mapping transformation, b: incremental, for the initial time, c, d, e, f, g, h: 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.

4.2.1 Exporting on the hard disk

The first set of performance measurements was realized using the hard disk as the output storage medium. In Figure 4, the triples maps contained simple SQL queries (the third query in Section 4.1). 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 3, a sacrifice in performance that pays off in subsequent executions. Similar results were obtained for mappings containing simple SQL queries resulting graphs containing up to 300k triples. Similar results are observed in Figure 5, regarding the more complicated SQL queries (the first query in Section 4.1), the difference however is not as great as in the previous case.

Next, in Figure 6, the same tests were performed in the complicated SQL query case, showing that the results here were very good. As Figure 6 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 detected in the source database, and performing faster even in the case when the percentage of the initial data that were changed reached to 75%. Only when the vast majority of the data 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 Figure 7, we demonstrate a set of measurements with all parameters the same, except for the query itself, which was similar to the second query in Section 4.1. 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 statements, similar to the existing ones, thus producing 6 times as much triples. This allowed us to scale up to 3 million triples, the behavior however remained the same.

4.2.2 Exporting on 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 Figure 8 and Figure 9, 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.

4.2.3 Exporting on 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 Figure 8 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, Figure 9, 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.

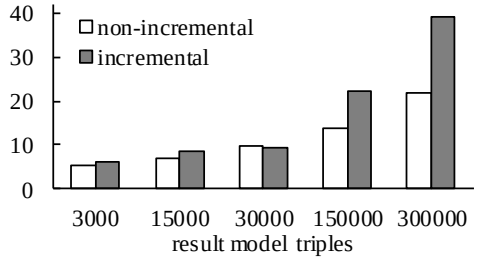

Figure 4. Time needed for the initial export of database contents into an RDF file using simple queries in mappings.

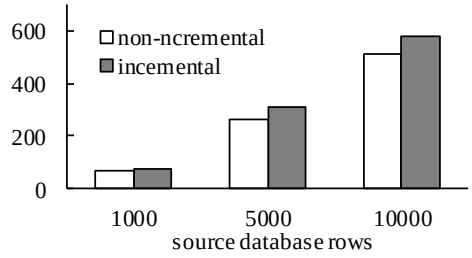

Figure 5. Time needed for the initial export of database contents into an RDF file, using complicated queries in mappings.

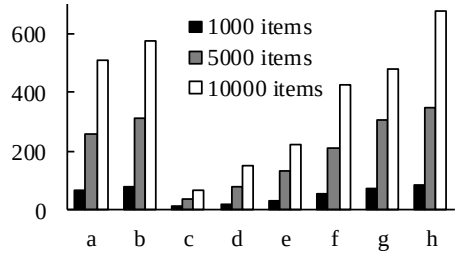

Figure 6. Incremental dumps in the case of complicated queries outperform non-incremental dumps for changes on approximately 3/4 of the data.

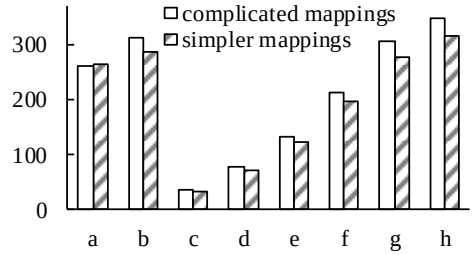

Figure 7. Incremental dumps it the case of complicated mapping queries (`JOIN`s among 4 tables) and simplified queries (`JOIN`s among 3 tables) with the same results.

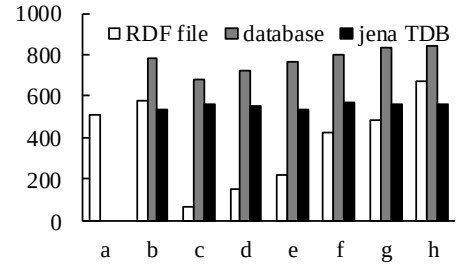

Figure 8. Hard disk, database, and Jena's TDB as the output medium. Complicated mappings, 10k items in the source database, approx. 180k triples in the resulting model.

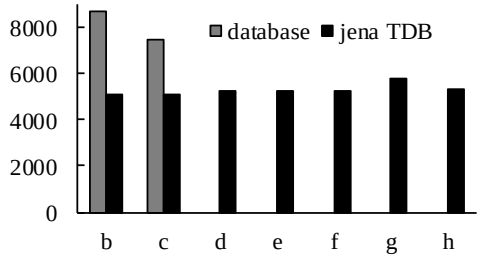

Figure 9. Database vs TDB behavior: Complicated mappings, 100k items in the source database, approx. 1.8 million triples in the resulting model.

4.3 Discussion

First, it has to be noted that logging the provenance of each triple on the hard disk has a great impact on the size required to store the logged output. Logging, which takes five times as much space as the result 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 constraints in terms of memory, imposed by the logging mechanism. Thus, while persisting the final model into a database or TDB takes longer than outputting it to the hard disk, 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 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 the 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.

It has to be noted that, given the fact that reification is being reconsidered in RDF 1.1 semantics, as named graphs could support the use cases of reification without requiring a data transform, 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.

Also, since there are various tools that allow direct, synchronous RDB2RF 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 are not updated to a significant amount often (e.g. daily), for instance in the bibliographic domain [20]. 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 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: The hereby introduced methodology guarantees only the syntactic validity of the result, without any checks on the semantics of 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 all is correct.

5. 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. 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 the hard disk. 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 initial content. 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 and storing in the relational database performed much worse. 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. Additional steps that could be followed in order to expand this work could include consideration of RDF (1.1-compatible) datasets in the mapping result, each one originating from a different triples map definition. Future work could also include further studying the impact of the SQL complexity 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, etc., and not only mapping definition provenance, thus allowing incremental RDF generation in other scenarios, in the same manner.

6. ACKNOWLEDGMENTS

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).

7. 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] Bakkas, J., Bahaj, M. 2013. Generating of RDF graph from a relational database using Jena API. *International Journal of Engineering and Technology*, 5, 2, 1970–1975

[7] De Virgilio, R., Maccioni, A., Torlone, R. 2013. Converting Relational to Graph Databases. In *First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*, pp. 1–6, New York, New York, USA, ACM Press

[8] 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/

[9] 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

[10] 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

[11] 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

[12] Chebotko, A., Lu, S., and Fotouhi, F. 2009. Semantics Preserving SPARQL-to-SQL Translation. In *Data & Knowledge Engineering*, 68, 10, 973–1000

[13] Cyganiak, R. 2005. A Relational Algebra for SPARQL, Technical Report HPL 2005-170

[14] Bizer, C., Schultz, A. 2009. The Berlin SPARQL Benchmark. In *International Journal On Semantic Web and Information Systems*, 5, 2, 1–24

[15] Bizer, C., Cyganiak, R. 2006. D2R Server - Publishing Relational Databases on the Semantic Web. In *Proceedings of the 5th International Semantic Web Conference*

[16] Erling, O., Mikhailov, I. 2007. RDF support in the Virtuoso DBMS. *Proceedings of the 1st Conference on Social Semantic Web*, Leipzig, Germany, 59–68

[17] 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

[18] 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

[19] 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

[20] Konstantinou, N., Spanos, D.-E., Houssos, N., Mitrou, N. 2014. Exposing Scholarly Information as Linked Open Data: RDFizing DSpace contents. In *The Electronic Library*, in press

[21] 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

[22] 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

[23] 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

[24] Das, S., Sundara, S., Cyganiak, R. 2012. R2RML: RDB to RDF Mapping Language. W3C Recommendation. Available online at http://www.w3.org/TR/r2rml/, accessed on February 28th 2014

[25] 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