Kamil Bajda-Pawlikowski

kbajda@cs.yale.edu

Querying RDF data stored in DBMS: SPARQL to SQL Conversion

Yale University technical report #1409

ABSTRACT

This paper discusses the design and implementation of a tool for SPARQL to SQL conversion. The converter receives a SPARQL query which is then parsed and analyzed. All three current approaches to storing RDF data in RDMS (triple store, property tables, and vertical partitioning) are supported in this paper. Depending on the tool's configuration one of these methods is chosen and a proper SQL query is generated. After executing the SQL query and getting rows back from the database, the tool converts the result into SPARQL XML format. In the end this result is returned as an answer to the initial SPARQL query.

The converter was evaluated using UniProt data (16.6 million triples) and real world SPARQL queries. The performance study included both the comparison of all RDF storing approaches and the analysis of the overhead of the conversion between SPARQL and SQL.

1. INTRODUCTION

RDF [1] data are usually stored in RDBMS and there are several approaches to improving the efficiency of querying these data. Users who want to retrieve RDF data should not be burdened by storage implementation details and by the necessity of constructing proper SQL queries. SPARQL [2], the language which became officially the W3C recommendation in January 2008, is well suited to querying RDF data because it reflects the concepts of the Semantic Web. Hence, a SPARQL to SQL converting tool is needed. The conversion ought to take place behind the scenes in order to hide the complexity of the process from the user.

A SPARQL to SQL conversion tool should support all three current RDF to DB mappings: single triples table, property tables, and vertical partitioning (one predicate per table) [3]. To increase the ease of use and deployment, the converter should be customized by changes in its configuration files. I envision the SPARQL to SQL converter running on a server and processing requests coming from the Internet via web form or webservices. Thus, I would opt for using technologies that can be easily integrated with the web environment.

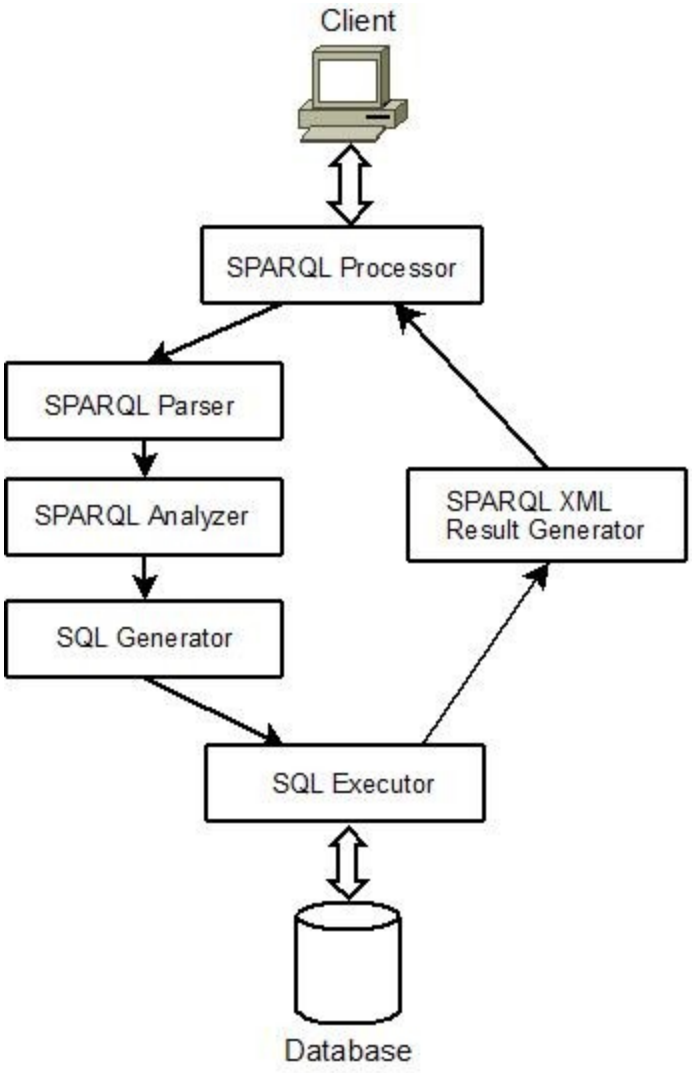
To evaluate the converter one needs to use real SPARQL queries over a large RDF dataset. The evaluation consists of assessing the overhead of SPARQL to SQL conversion and of comparing the three approaches to storing RDF data.

The remainder of this paper is organized as follows. In section 2, I describe the design and implementation of the SPARQL to SQL converter. In section 3, I evaluate the tool using real world

RDF data and a few SPARQL queries. Finally, in section 4, I conclude and propose further development and research related to the discussed conversion.

2. DESIGN AND IMPLEMENTATION

The overall architecture of the system is shown on the diagram below.



Now, I will describe each component in detail and present the way it was implemented. My analysis will follow the way a SPARQL query is processed.

2.1. SPARQL Processor

This component follows the Facade design pattern. It hides the complexity of the whole conversion process by providing an *execute()* method that accepts a SPARQL query in string format as an argument and returns an XML result (also in string format). The SPARQL Processor passes on the query to the next component.

2.2. SPARQL Parser

This component is responsible for parsing the SPARQL query. The component delegates this task to the ARQ module [4] which belongs to the Jena Framework [5]. Although Jena and ARQ constitute a complete system supporting RDF storage and SPARQL execution, I use only their SPARQL parsing function.

2.3. SPARQL Analyzer

This component takes the SPARQL query parsed earlier and traverses its object representation using the Visitor design pattern. During this process a new data structure is built to represent the query. Only those elements of the SPARQL syntax which I currently support are reflected in the new data structure. These elements include SELECT, WHERE (triples patterns only, no FILTER nor OPTIONAL), ORDER, OFFSET and LIMIT. The new query representation is better suited to translation into SQL than the original one coming from the SPARQL Parser and ARQ.

2.4. SQL Generator

This component is responsible for producing a SQL query based on the SPARQL query and on selected RDF to DB mapping. The Factory design pattern is used here. During the initialization of the converter the appropriate implementation (specified in the configuration) of the SQL Generator is loaded and run. Each of the three available implementations has its own configuration file which describes the details of the tables containing RDF data. This solution allows quick deployment and changes without modifying/recompiling the source code.

For example, the TripleSQLGenerator reads the following configuration file:

```
<simple_triple>
      <triple_table>uniprot_triples</triple_table>
      <subject_field>subject</subject_field>
      <predicate_field>predicate</predicate_field>
      <object_field>object</object_field>
      <table_alias_prefix>t</table_alias_prefix>
</simple_triple>
```

The implementation of vertical partitioning requires mapping between predicates and database tables.

```
[...]
<table predicate="http://purl.uniprot.org/core/organism">uniprot_v_organism</table>
<table predicate="http://purl.uniprot.org/core/name">uniprot_v_name</table>
<table predicate="http://purl.uniprot.org/core/mnemonic">uniprot_v_mnemonic</table>
<table predicate="http://purl.uniprot.org/core/existence">uniprot_v_existence</table>
[...]
```

The implementation of property tables requires more detailed information about the tables:

```
[...]
<property_table name="uniprot_p_protein" subject_field="subject">
      <field predicate="http://purl.uniprot.org/core/organism">organism</field>
      <field predicate="http://purl.uniprot.org/core/name">name</field>
      <field predicate="http://purl.uniprot.org/core/mnemonic">mnemonic</field>
      <field predicate="http://purl.uniprot.org/core/existence">existence</field>
</property_table>
```

```
[...]
```

The result of the SQL Generator's execution is a string containing a valid SQL query.

2.5. SQL Executor

This component takes the generated SQL query and executes it against a database. The connection credentials are stored in a configuration file.

```
<db>
      <driver>org.postgresql.Driver</driver>
      <url>jdbc:postgresql://localhost:5432/sparql</url>
      <user>sparql</user>
      <password>sparql</password>
</db>
```

If the query returns a result set, rows are fetched from the database and added as objects to a result list.

2.6. SPARQL XML Result Generator

This component builds an XML tree from the result list according to the SPARQL XML Result standard. The XML tree is then serialized to string format. The SPARQL Processor *execute()* method returns the result. This ends the processing of the SPARQL query.

2.7. Tools and libraries

The converter was developed with Java [6] 1.5 technology using open source tools and libraries such as Eclipse [7], Apache Ant [8] and Log4j [9]. The XML configuration files are parsed by JDOM [10]. The SPARQL XML results are created using Apache Xerces [11]. Jena's ARQ module is also a Java-based project – a fact which significantly simplified integrating it into my codebase. PostgreSQL [12] 8.3 was chosen as the database server.

3. EVALUATION

3.1. RDF Data

In the evaluation of my converter I used UniProt [13] data. The original file consisted of over 18.8 million RDF triples (2.7 GB raw text data) 0.4% of which was skipped (because of particularly long string literals). Next, the elimination of duplicates reduced the dataset to 16.6 million triples. Uniprot data contained 64 unique predicates of which 31 were multi-valued (they appear more than once for some subjects).

3.2. SPARQL queries

I used five SPARQL queries developed by an expert. In some cases the queries were slightly changed in order to eliminate currently unsupported elements of SPARQL's syntax (i.e. FILTER, REGEX, CONSTRUCT). The queries below are shown in an abbreviated form (all URIs are stripped out).

Query1

Find in the Human organism all proteins the existence of which is not certain.

```
SELECT  ?prot ?name ?mnem
WHERE
  { ?prot  <organism>  <9606> ;
           <name>  ?name ;
           <mnemonic>  ?mnem ;
           <existence>  <Uncertain_Existence> .
  }
```

Query 2

Return all items labeled "Pheochromocytoma" with their types.

```
SELECT  ?class ?type
WHERE
  { ?class  <type>  ?type ;
            <label>  "Pheochromocytoma" .
  }
```

Query 3

Find all alternative products for Tyrosine Kinases with their masses.

```
SELECT  ?prot ?name ?alt_prod ?mass
WHERE
  { ?prot     <classifiedWith>  <829> ;
              <name>  ?name ;
              <annotation>  ?alt_prod_anno .
    ?alt_prod_anno <type> <Alternative_Splicing_Annotation> ;
                   <sequence>  ?alt_prod .
    ?alt_prod  <mass>  ?mass .
  }
```

Query 4

Find all proteins (and their clusters) that are associated with diseases.

```
SELECT  ?dis ?cluster
WHERE
  { ?prot  <memberOf>  ?cluster ;
           <annotation>  ?dis .
    ?dis   <type>  <Disease_Annotation> .
  }
```

Query 5

Find all proteins that interact with protein Endofin.

```
SELECT  ?assoc_prot
WHERE
  { <Q7Z3T8> <interaction>  ?interaction .
    ?interaction  <participant>  ?assoc_prot .
  }
```

3.3. SYSTEM SPECIFICATION

All the tests were performed on a machine with Intel Core 2 Duo 2.2 GHz (4MB L2 cache), running

Microsoft Windows Vista, with 3 GB of RAM and a 100 GB hard drive (7200 rpm, 8 MB cache).

3.4. STORES IMPLEMENTATION

Triple Store

In this case all RDF data are stored in a single table in which each row represents one triple.

```
CREATE TABLE uniprot_triples (
subject VARCHAR(255),
predicate VARCHAR(255),
object VARCHAR(255));
```

There are three B+tree indices on the table: one clustered on (subject, predicate, object) and two unclustered on (predicate, object, subject) and (object, subject, predicate).

Vertically Partitioned Store

Here, triples with the same predicate are grouped into a separate table. Consequently, one table is created per predicate. Every one of these tables stores one subject-object pair per row. A definition of one vertical table is shown below (other differ only in their names).

```
CREATE TABLE uniprot_v_name (
subject VARCHAR(255),
object VARCHAR(255));
```

There is a clustered index on the subject column and an unclustered one on the object column.

Property Table Store

In this case, subjects described with a similar set of predicates are grouped together to form one table. Every one of these tables contains a subject attribute and a number of attributes that represent different predicates. The values of some predicates may be unknown, and then NULLs are stored in their place (there are no corresponding triples in RDF data). One of my property tables looks as follows:

```
CREATE TABLE uniprot_p_protein (
subject VARCHAR(255),
organism VARCHAR(255),
name VARCHAR(255),
mnemonic VARCHAR(255),
existence VARCHAR(255));
```

There exists a clustered index on the subject column and unclustered indices on all other columns.

3.5. Test procedures

Each of the five SPARQL queries was executed three times against each of the three implementations. After each run the database server was restarted and large files were copied in order to reset any internal

and disk buffers; in some cases a reboot of the system was required. Establishing a database connection via JDBC and reading the converter configuration did not affect time measurements because these operations were done during the application startup. All time measurements were taken during program execution by calling `System.currentTimeMillis()`. All distinct query processing steps (parsing the SPARQL query, analyzing it, generating the SQL query, executing it against DB and creating an XML result) were timed separately during each run.

While comparing the different stores I took the average of three SQL query execution times. Executing a query against the DB includes fetching result rows, converting them into object representations, and adding these objects to the result list. The times of all other operations did not differ between the implementations.

3.6. Comparison results

The results (in milliseconds) of executing all five SPARQL queries against the three stores are presented in the table below.

	Triple patterns	Result rows	Triple store	Vertical store	Property table
Query 1	4	235	2,466	1,466	531
Query 2	2	37	1,641	1,360	463
Query 3	6	253	31,645	9,975	15,381
Query 4	3	16,687	126,978	84,050	85,252
Query 5	2	4	439	213	-

The table includes information about the number of triple patterns in each query and the number of rows returned as a result. Irrespective of the RDF storage method chosen, both of these factors influence the length of time needed to process a query. If the number of patterns is *n*, *n-1* joins are required in triple store and vertical partitioning. The number of joins is usually smaller for property tables since one property table can store more than one predicate value per row. Self-joins over a huge triples table are more expensive than joins between smaller tables; joins on subjects are faster than joins on subject-object since the former are clustered in each table.

The fact that triple store performance is always the worst is immediately apparent. A careful analysis reveals that for some queries the ideally designed property table can give a 3-5x speedup (see Query 1 and 2). The key advantage here is that no joins are required because all triple patterns match the same subject. This means that a single property table is sufficient to answer a particular query.

The results of Query 3 show the power of vertical partitioning. It performs over 3 times faster than triple store solely because of faster joins (and 5 join operations are required here). Even though using property tables requires fewer joins than using vertical partitioning, the former method is adversely affected by the complexity of the query. The matching of two subject-object patterns precludes the creation of one property table to answer Query 3 (unlike in the case of Queries 1 and 2). Moreover, since multi-valued

attributes are represented as multiple rows, the size (and consequently the processing time) of the property tables increases significantly. In fact, the property table implementation of Query 3 needs to perform a few joins with the leftover triples table (the size of which is very large). Still, the presence of fewer joins than in triple store makes the property table approach twice as fast.

In Query 4, the considerable length of execution time (in all three approaches) is caused by the large number of result rows (over 16.6 thousands). Both vertical partitioning and property tables are about 50% faster than triple store.

Vertical partitioning wins with triple store in Query 5 simply because the vertical tables needed here are small (9K and 18K of rows), so the join between them can be done much faster than self-join on the huge triples table. The property table result for Query 5 was omitted because it would be either identical with vertical partitioning or with triple store, depending on how the property table was implemented (a property table with one predicate is equivalent to a vertical table; a leftovers triples table is equivalent to a regular triples table).

3.7. Conversion overhead

In this paragraph I will discuss how much time is spent on SPARQL to SQL conversion. Times obtained during benchmarking the implementations of different stores suggested that this overhead may be quite significant (especially for easy queries, e.g. Query 5) and can dominate query processing. However, comparing the stores required the restarting of the converter each time a query was executed, which meant that the time measurements obtained included the cost of initializing all components and third party libraries (although I loaded the converter configuration and established a JDBC connection before executing a query). Thus, to assess the conversion overhead in a warmed-up system (which reflects normal working conditions), I performed another test in which all five queries were run one after another without restarting the converter. I measured the times in a few different sequences of queries, always omitting the result of the first query which included the initializing cost.

Parsing any of the five SPARQL queries by the ARQ module lasts 4 ms (the first time cost is about 400 ms). The typical time required to analyze a SPARQL query and to generate a SQL query does not exceed 2 ms (whereas the initializing query falls into the 30-70 ms range). The time needed to create a SPARQL XML result is correlated with the number of result rows and can vary from 1-2 ms (Query 2 and 5), through about 20 ms (Query 1 and 3), to almost a second (935 ms for Query 4). A first time run adds about 50 ms to the generation of an XML result. All in all, these results show that the conversion overhead during normal operation is only a small fraction (typically below 1-2% and less than 5% in the worst case) of the cost of executing the SQL query against a database.

4. CONCLUSION AND FUTURE WORK

The performance results clearly show that the simplest approach to storing RDF data (i.e. using a single triples table) is the worst. Although sometimes the efficiency of the property table method may be impressive, this happens only when the property table is perfectly suited to a given query. Generally,

property tables are not easy to design. Some property tables cannot coexist because they contain the same predicates. Property tables are not efficient if queries are complicated and when multi-valued predicates are ubiquitous. The overlapping of predicate semantics (using the same predicate URIs to express similar properties, e.g. <type>, <label>, for different classes of objects) may prevent the creation of effective property tables in a real world system (UniProt data would suffer from this limitation).

Vertical partitioning is not subject to the problems mentioned above. Its design is straightforward and the more complicated the query, the more visible the performance gains. Therefore, I advocate using vertical partitioning instead of triple store.

The SPARQL to SQL conversion overhead proved to be very low and because of this I recommend my tool as a useful extension to any system that keeps RDF data in RDBMS and wants to provide a SPARQL querying feature. Obviously, some enhancements and tuning, especially for multithreading processing, should be introduced to address real world requirements in terms of scalability.

In order to handle more sophisticated SPARQL queries the converter needs to support more elements of SPARQL's syntax. I consider FILTER (including REGEX) and OPTIONAL the most important of these elements. The former provides the possibility of filtering results using logical expressions and pattern matching. The latter lets users add triple pattern conditions that are not mandatory.

The next version of the SPARQL to SQL converter should also add dictionary string encoding to all approaches to storing RDF data in DBMS. String encoding saves space required to store triples because integer identifiers replace string values. It also reduces query execution time despite the fact that additional joins are required to retrieve strings from a dictionary table. Taking into account the fact that efficient REGEX implementation requires pattern matching at the SQL level, special text-oriented indices should be introduced (a regular b+tree index is useless in this case and a sequential scan of the entire table is performed).

Although most RDF data are URIs and string literals which map onto SQL's varchar datatype, there are some predicates the values of which would be better served if stored as date, integer or other non-string datatypes. This enhancement would be beneficial for ordering and filtering operations. Supporting various datatypes seems to be particularly well suited to property tables and vertical partitioning. Unlike strings, other datatypes should not be dictionary encoded but rather stored explicitly in tables.

Another interesting thing to investigate would be the possibility of mixing all three approaches to storing RDF data. During the conversion from SPARQL to SQL the best option for a particular query would be chosen. In some cases all three methods may be complementary (e.g. the best matching property table is taken together with a few vertical tables). If the query asks for all properties of a given subject, a simple triples table may still be the best choice.

REFERENCES

[1] Resource Description Framework, *http://www.w3.org/RDF/*

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

[3] D. Abadi, A. Marcus, S. Madden, K. Hollenbach, *Scalable Semantic Web Data Management Using Vertical Partitioning*, In VLDB 2007

[4] ARQ - A SPARQL Processor for Jena, *http://jena.sourceforge.net/ARQ/*

[5] Jena – A Semantic Web Framework for Java, *http://jena.sourceforge.net/*

[6] The Source for Java Developers, *http://java.sun.com/*

[7] Eclipse - an open development platform, *http://www.eclipse.org/*

[8] The Apache Ant project, *http://ant.apache.org/*

[9] Apache log4j, *http://logging.apache.org/log4j/*

[10] The JDOM project, *http://www.jdom.org/*

[11] Apache Xerces, *http://xerces.apache.org/*

[12] PostgreSQL, *http://www.postgresql.org/*

[13] Universal Protein Resource, *http://www.uniprot.org/*

APPENDIX 1

SPARQL XML Result generated for Query 1 (only one result row included for brevity)

```
<?xml version="1.0" encoding="UTF-8"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
    <head>
        <variable name="prot"/>
        <variable name="name"/>
        <variable name="mnem"/>
    </head>
    <results distinct="false" ordered="false">
        <result>
            <binding name="prot">
                <literal
datatype="string">http://purl.uniprot.org/uniprot/A6NGE7</literal>
            </binding>
            <binding name="name">
                <literal datatype="string">EC 4.-.-.-</literal>
            </binding>
            <binding name="mnem">
                <literal datatype="string">URAD_HUMAN</literal>
            </binding>
        </result>

<!-- MORE RESULTS HERE -->

    </results>
</sparql>
```

APPENDIX 2

SQL queries generated by SPARQL to SQL converter for Query 1 for all three implementations

Triple store

```
SELECT t0.subject AS prot, t1.object AS name, t2.object AS mnem
FROM uniprot_triples AS t0, uniprot_triples AS t1, uniprot_triples AS t2,
uniprot_triples AS t3
WHERE 1=1
 AND t0.subject = t1.subject
 AND t1.subject = t2.subject
 AND t2.subject = t3.subject
 AND t2.predicate = 'http://purl.uniprot.org/core/mnemonic'
 AND t3.object = 'http://purl.uniprot.org/core/Uncertain_Existence'
 AND t3.predicate = 'http://purl.uniprot.org/core/existence'
 AND t1.predicate = 'http://purl.uniprot.org/core/name'
 AND t0.object = 'http://purl.uniprot.org/taxonomy/9606'
 AND t0.predicate = 'http://purl.uniprot.org/core/organism'
```

Vertically Partitioned Store

```
SELECT t0.subject AS prot, t1.object AS name, t2.object AS mnem
FROM uniprot_v_organism AS t0, uniprot_v_name AS t1, uniprot_v_mnemonic AS t2,
uniprot_v_existence AS t3
WHERE 1=1
 AND t0.subject = t1.subject
 AND t1.subject = t2.subject
 AND t2.subject = t3.subject
 AND t3.object = 'http://purl.uniprot.org/core/Uncertain_Existence'
 AND t0.object = 'http://purl.uniprot.org/taxonomy/9606'
```

Property Table Store

```
SELECT t0.subject AS prot, t0.name AS name, t0.mnemonic AS mnem
FROM uniprot_p_protein AS t0
WHERE 1=1
 AND t0.existence = 'http://purl.uniprot.org/core/Uncertain_Existence'
 AND t0.organism = 'http://purl.uniprot.org/taxonomy/9606'
```