

# Scalable Long-term Preservation of Relational Data through SPARQL queries

**Editor(s):** Name Surname, University, Country

**Solicited review(s):** Name Surname, University, Country

**Open review(s):** Name Surname, University, Country

Silvia Stefanova<sup>\*</sup> and Tore Risch

*Department of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden*

**Abstract.** We present an approach for scalable long-term archival of data stored in relational databases (RDBs) as RDF, implemented in the SAQ (Semantic Archive and Query) system. The proposed approach is suitable for archiving scientific data used in scientific publications where it is desirable to preserve only parts of an RDB, e.g. only data about a specific set of experimental artefacts in the database. With the approach, long-term preservation as RDF of selected parts of a database is specified as an *archival query* in an extended SPARQL dialect, *A-SPARQL*. The query processing is based on automatically generating an RDF view of a relational database to archive, called the *RD-view*. A-SPARQL provides flexible selection of data to be archived in terms of a SPARQL-like query to the RD-view. The result of an archival query is a *data archive* file containing the RDF-triples representing the relational data content to be preserved. The system also generates a *schema archive* file where sufficient meta-data are saved to allow the archived database to be fully reconstructed. An archival query usually selects both properties and their values for sets of subjects, which makes the property  $p$  in some triple patterns unknown. We call such queries where properties are unknown *unbound-property queries*. To achieve scalable data preservation and recreation, we propose some query transformation strategies suitable for optimizing unbound-property queries. These query rewriting strategies were implemented and evaluated in a new benchmark for archival queries called *ABench*. *ABench* is defined as set of typical A-SPARQL queries archiving selected parts of databases generated by the Berlin benchmark data generator. In experiments, the SAQ optimization strategies were evaluated by measuring the performance of A-SPARQL queries selecting triples for archival queries in *ABench*. The performance of equivalent SPARQL queries for related systems was also measured. The results showed that the proposed optimizations substantially improve the query execution time for archival queries.

**Keywords:** Database archival, benchmark, SPARQL optimization, SPARQL views of relational databases, unbound-property queries.

## 1. Introduction

The importance of digital preservation research has been growing for the past ten-fifteen years. Many papers and books [9][13][33] describing problems, tools and techniques for digital preservation, have been written, and standards providing preservation models have been published [29][36]. However, most of this work focuses on preservation of file-based digital objects like documents, images, and web pages [9]. Much less work has focused on the preservation of databases and scientific data, where there is a

recognized need to preserve scientific data [12][28][37][38][39]. Furthermore, preserving scientific data together with scientific publications would contribute to documenting the origin and lineage of scientific achievements [12]. For this a concept of ‘Scientific Publication Packages (SPPs)’ was introduced in [12]. The SPPs were described as composite digital objects linking experimental raw data, associated with metadata, ‘derived information’, and knowledge, including associated publications.

Scientific data, i.e. experimental and observational data, as well as data generated by instruments and

---

<sup>\*</sup> Corresponding author. E-mail: editorial@iospress.nl. Check if the checkbox in menu *Tools/Options/Compatibility/Lay out footnotes like Word 6.x/95/97* is selected if you make a footnote for the corresponding author.

sensors, reside in large datasets that are often stored in relational databases. During the research process the scientists need to select subsets of these databases (i.e. specific tables, rows, columns) to be analysed and processed in order to create a scientific model. Once the model is validated the research results are documented and published. By preserving the selected subsets of both data and publications within one digital object, future reuse, verification, and heritage [33] of the published scientific results can be guaranteed.

Selective data preservation is needed also for example in cases of medical data preservation where, in order to protect the privacy of patients, sensitive data like zip code, dates of birth, salaries, etc. should be excluded from archiving [22]. Other examples are when selecting representative geospatial data for preservation [24] or preserving web resources based on some criteria [13].

For long-term preservation of data, it is desirable for the contents of a database to be saved in a neutral format, so that it can be reconstructed and used after a very long time using current technologies for data representation, which are continuously evolving. Furthermore, preserved representations must include sufficient meta-data to retrieve, explain, reproduce, and disseminate the experiments. We propose RDF and RDF-Schema (RDFS) based neutral format as a database technology-independent format for long-term preservation of data, which provides standard meta-data representation for describing all kinds of data, including relational databases [11].

In this paper we present an approach for scalable long-term archival of selected data stored in relational databases (RDBs) as RDF, implemented in the SAQ (Semantic Archive and Query) system. The proposed approach is suitable for archiving scientific data used in scientific publications where it is desirable to preserve only selected parts of an RDB, e.g. only data about a specific set of artefacts in the database related to some publications. For this SAQ provides selective archival of user-specified parts of an RDB using an extended SPARQL query language, A-SPARQL.

To map a relational database into RDF, SAQ automatically generates an RDF view of the relational database to be archived called the *RD-view*. The RD-view is defined in terms of an RDFS ontology for describing RDB schemas in general.

To select the parts of an RDB to archive, a SAQ user defines an *archival query* to the RD-view in A-SPARQL. For example, the classes representing the tables named *product* and *offer* in the RDB *Products*

of the Berlin Benchmark dataset [3][5] are archived with the archival query:

```
ARCHIVE AS 'data.nt', 'schema.nt'
FROM <Products>
TRIPLES {?subject ?property ?value}
WHERE   {?subject rdf:type db:product }
        UNION
TRIPLES {?subject ?property ?value}
WHERE   {?subject rdf:type db:offer  }
```

In the query the result triples are stored in the *data archive* file '*data.nt*'. While executing the archival query, the system simultaneously produces sufficient meta-data to enable reconstruction of the selected parts of the archived RDB. These meta-data are stored in a *schema archive* file, '*schema.nt*'.

When an archived RDB content is to be recreated, SAQ reads the schema archive to automatically recreate the RDB schema in another RDBMS. The RDB thus created is then populated by reading the data archive and converting the read data into table rows according to the schema. This allows migration from one RDBMS to another, perhaps from different vendors. If only selected parts of the RDB are archived, a corresponding partial RDB is recreated containing only the relevant parts of the schema and data. For migrating data from RDBs to RDF repositories, the contents of the schema and data archive files can be directly loaded into an RDF repository system, e.g. [43][44] [45].

For processing an archival query in A-SPARQL SAQ internally generates a corresponding SPARQL query to select the triples of the database to archive. The archival queries are straight-forward to translate into CONSTRUCT queries. As in the example, unions of sets of triples are often archived, e.g. for different classes and properties, which makes the generated SPARQL queries to become UNION CONSTRUCT queries.

Archival queries typically select sets of attributes of tables to archive. This corresponds to selecting sets of RDF properties in the RD-view of the database to be archived. In the example all properties of the classes representing the tables *product* and *offer* are selected for archival. Therefore, in the generated queries the property *p* in one or several triple patterns (*s,p,o*) is a variable. We call such triple patterns (*TPs*) *unbound-property triple patterns (UPTP)*, and the queries having such triple patterns *unbound-property queries* [30]. To achieve scalable data preservation and recreation, we developed some special query rewriting optimizations for optimizing unbound-property queries. Archival queries can also

contain conventional TPs where the properties are URIs representing RDF properties in the RD-view, which we call *bound-property triple patterns (BPTP)*. Queries having only BPTPs are called *bound-property queries*, which are processed using known methods [15][16][18].

To evaluate the performance of typical archival queries a new benchmark called *ABench* was developed. *ABench* is defined as set of typical archival queries, specified in A-SPARQL, that archive selected parts of databases generated by the Berlin benchmark data generator [4]. A new benchmark was developed since the archival queries generate CONSTRUCT unbound-property queries with UNION clauses, which is not covered by any existing benchmark.

In the experiments, the SAQ optimization strategies were evaluated using *ABench*. The experiments showed that the proposed query rewriting optimizations substantially improve the query execution time for unbound-property queries selecting RDB contents to archive. We also compared the performance of our approach with other systems processing SPARQL queries over views of RDBs and found that the proposed optimizations improve query scalability compared with the approaches used in those systems.

The rest of this paper is organized as follows. Section 2 presents a motivating example for selective preservation of a relational database as RDF, Section 3 presents the SAQ system and the A-SPARQL language, and Section 4 the archival benchmark *ABench*. Section 5 describes the RD-view and the SAQ query processing steps, along with the SAQ rewriting optimizations. Section 6 evaluates the performance of the query optimizations using *ABench*, Section 7 describes related work, and Section 8 provides a summary.

## 2. Motivating example

A user, who has worked on analysing products with different properties, wants to preserve together with the analysis result data about analysed products having some special properties. In the example, data about products produced in Sweden and having a property *pNum1* > 348 have to be preserved. Furthermore, to the preserved data it is needed to add sufficient meta-data in order to allow later reconstruction of the preserved data [9].

The products data resides in a RDB. Fig. 1 shows a small RDB called *Products*, which is part of the

relational Berlin benchmark dataset. The database has four tables, *product*, *productfeature*, *productfeatureproduct* and *producer*, populated with some data. The columns *pnr*, *pfnr* and *prodnr* are primary keys in the tables *product*, *productfeature*, and *producer*. The column *producer* in the table *product* references the column *prodnr* in the table *producer* as foreign key. The table *productfeatureproduct* is a many-to-many link table between the tables *product* and *productfeature*.

Table *product*

<u>pnr</u>	label	pNum1	producer
1	cicatrices	100	2
2	emulsifying	450	3

Table *productfeature*

<u>pfnr</u>	publishDate
3	2000-06-22
4	2000-07-08

Table *productfeatureproduct*

<u>product</u>	<u>productFeature</u>
1	3
2	3
2	4

Table *producer*

<u>prodnr</u>	country
2	DE
3	SE

Fig. 1. RDB Products.

To archive as RDF the selected products along with their properties and values, we define in SAQ the following archival query:

```

ARCHIVE AS ('productD.nt', 'productS.nt')
FROM <Products>
TRIPLES { ?product ?property ?value }
WHERE {
?product rdf:type db:product .
?product db:product_producer ?producer .
?producer db:producer_country 'SE' .
?product db:product_pNum1 ?pn1 .
FILTER (?pn1 > 348)
}

```

Execution of the archival query produces two N-Triples files, *'productD.nt'* to store the archived products from the RDB and *'productS.nt'* to store the schema archive required for recreating the parts of the RDB schema representing the archived products. The RDB reconstructed from the archival query is shown in Fig. 2. It contains only the tables, attributes and rows required to reconstruct the data archived by the query. After the reconstruction, SAQ can process

SPARQL queries to the RD view of the reconstructed database.

Table *R\_product*

<u>pnr</u>	label	pNum1	producer
2	emulsifying	450	3

Table *R\_productfeature*

<u>pfnr</u>
3
4

Table *R\_producer*

<u>prodnr</u>
3

Table *R\_productfeatureproduct*

<u>product</u>	<u>productFeature</u>
2	3
2	4

Fig. 2. Reconstructed RDB.

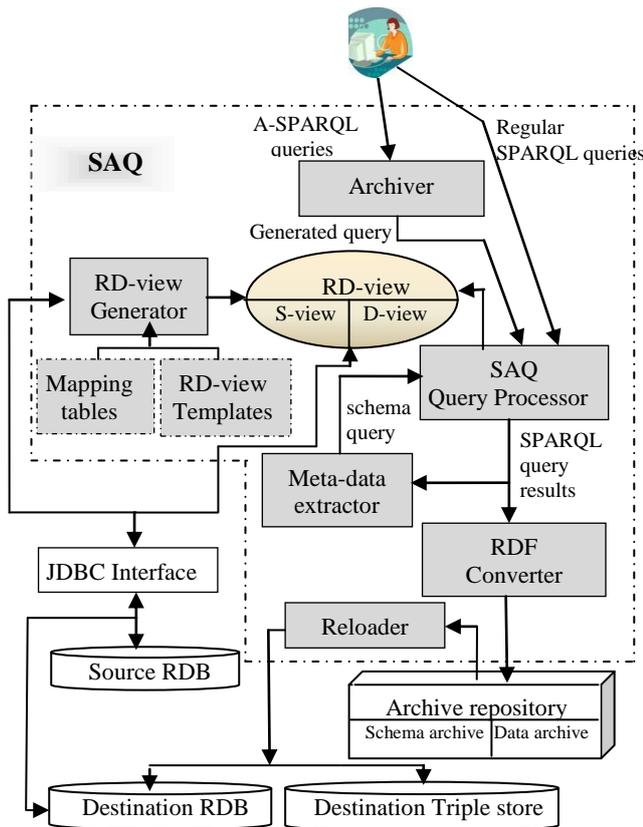


Fig. 3. SAQ.

### 3. The Semantic Archive and Query system

The developed SAQ system for long-term preservation of relational databases follows conceptually the OAIS reference model [36]. The OAIS model is composed by four functional units: Ingest, Archival Storage, Data Management, and Access. The Ingest unit accepts Submission Information Packages (SIPs) and generates Archival Information Packages (AIPs) for storage and management. The Archival Storage unit receives AIPs from Ingest and adds them to permanent storage. The Data Management unit provides functions for populating, maintaining, and accessing variety of meta-data stored in the repository. The Access unit provides an interface between the archive and the consumer.

An OAIS' AIP contains Content Information, i.e. the archived data and the representational metadata, together with a Preservation Description Information (PDI). The PDI contains reference information, context information, provenance information, etc.

SAQ provides functionality for the Ingest component, in particular on generating the content information in the AIPs when preserving relational database contents as RDF.

In order to preserve both schema and data from an RDB, it is important to represent not only the contents of the RDB as RDF, but also the schema. Therefore the RD-view is defined as a union of a *schema view* (the S-view), representing the RDB schema, and a *data view* (the D-view), representing the RDB contents. To allow for interoperability with other systems mapping RDBs to RDF, e.g. [7][8][26], the data view mappings conform to the *direct mapping* recommendations by W3C Recommendation [23].

The architecture of the SAQ system is presented in Fig. 3. The *source RDB* is the underlying RDB, which can be queried by SPARQL and preserved by A-SPARQL queries.

The *RD-view generator* automatically generates one *RD-view* over each *source RDB* by reading the database schema through a *JDBC interface*. The *RD-view templates* thereby provide general prototypes for the structure of the RD-view for any relational database, and the contents of the *mapping tables* provide RDB-to-RDF mappings for specific relational meta-data into RDF. An archival query is processed by the *archiver* and translated into a corresponding *generated query* in SPARQL, which is sent to the *SAQ query processor*. The generated query retrieves the data to archive from the RDB. *Regular*

non-archiving SPARQL queries are sent directly to the SPARQL query processor.

The SAQ query processor executes the SPARQL queries to the RD-view by accessing the source RDB through the JDBC interface.

### 3.1. Archival queries

Archival queries have the following syntax:

```
ARCHIVE AS data_archive_file,
          schema_archive_file
FROM URI
archive_specification
[UNION
archive_specification]...
```

where an *archive specification* is defined as:

```
archive_specification :=
TRIPLES {archived_triple_patterns}
[WHERE {archive_restrictions}]
```

An archival query is specified by an *ARCHIVE* statement where a *data archive file* and the *schema archive file* are specified. SAQ will create these files using the N-triples format [34]. The *FROM* clause specifies the *URI* representing the RDB to archive. The URI is assigned by the user once for each RDB.

The body of an archival query is specified by a (union of) *archive specifications*, which select the triples to archive specified as *archived triple patterns* in a TRIPLES clause and an optional *archive restriction* in a WHERE clause. An archive restriction restricts the triples to archive. It consists of a graph pattern and may include SPARQL functions. A union of several archive specifications can be defined to archive several different sub-graphs from the RD-view.

The following are examples of some of the ABench queries:

```
A1 ARCHIVE AS 'data1.nt', 'schema1.nt'
FROM <Products>
TRIPLES {?subject ?property ?value }

A2 ARCHIVE AS 'data2.nt', 'schema2.nt'
FROM <Products>
TRIPLES {?subject ?property ?value }
WHERE{?subject rdf:type db:product .
      db:product rdf:type rdfs:Class }
UNION
TRIPLES {?subject ?property ?value }
WHERE{?subject rdf:type db:offer .
      db:offer rdf:type rdfs:Class }

A3 ARCHIVE AS 'data3.nt', 'schema3.nt'
FROM <Products>
```

```
TRIPLES {?subject1 db:product_label
          ?value1}
UNION
TRIPLES {?subject2 db:offer_price
          ?value2}
UNION
TRIPLES {?subject3 db:offer_webpage
          ?value3}

A4 ARCHIVE AS 'data4.nt', 'schema4.nt'
FROM <Products>
TRIPLES {?subject1 db:product_pNum1
          ?value1}
WHERE { FILTER (?value1 > 214 ) }
UNION
TRIPLES {?subject2 db:product_pNum3
          ?value2}
WHERE {FILTER (?value2 < 348 ) }
UNION
TRIPLES {?subject3 db:review_text ?value3}
WHERE {
  ?subject3 db:review_rating4 ?value4 .
  FILTER REGEX(?value3, 'time' ) .
  FILTER (?value4 > 8 )
}
```

Query A1 archives the entire database and stores it as a data archive file *'data1.nt'* and a schema archive file *'schema1.nt'*. Query A2 archives only the RDFS classes having the URI *<db:product>* and *<db:offer>*, along with all their properties. Query A3 archives only the RDF properties having URIs *<db:product\_label>*, *<db:offer\_price>*, and *<db:offer\_webpage>*. Finally, query A4 archives the property *product\_pNum1* for values > 214, the property *product\_pNum3* for values < 348, and the property *review\_text* for values matching the string *'time'* if the property *review\_rating4* > 8.

An archival query is straight-forward to translate into a CONSTRUCT SPARQL query. It is usually a CONSTRUCT-UNION query where unions of sets of triples are archived. For example, A3 is translated into the following generated SPARQL query:

```
Q3: CONSTRUCT
{ ?subject1 db:product_label ?value1 .
  ?subject2 db:product_price ?value2 .
  ?subject3 db:offer_webpage ?value3 }
FROM <Products>
WHERE
{{?subject1 db:product_label ?value1 }
  UNION
 {?subject2 db:offer_price ?value2 }
  UNION
 {?subject3 db:offer_webpage ?value3 } }
```

Since archival queries always select a sub-graph from the RDF graph of the RD-view to archive, all archived triple patterns in a generated query appear both in the CONSTRUCT clause and the WHERE clause.

The translation rules from A-SPARQL to the generated SPARQL are the following:

- 1) The CONSTRUCT clause of the translated SPARQL query consists of all the archived triple patterns in the TRIPLES clauses of the archive specifications.
- 2) The WHERE clause of the generated SPARQL query is a UNION of one SPARQL query fragment per archive specification consisting of its archived triple patterns and optional archive restrictions.

### 3.2. Generating the schema archive

During the execution of the generated SPARQL query, the property URIs of the triples to be archived are collected by the *meta-data extractor* while iterating over the result stream. When all data to archive have been processed, the meta-data extractor joins the collected properties with the S-view by issuing a *schema query*.

The archived content retrieved by the generated query and the corresponding meta-data retrieved by the schema query are written by the *RDF converter* into two N-triple files in the *archive repository*.

### 3.3. Restoring a database

Later on, when a preserved database is to be restored, the *reloader* reads from the archive repository the two archive files and makes the database live again by populating it into a *destination RDB* or alternatively a *destination triple store*. When an RDB is restored, the reloader first reads the schema archive in order to generate the RDB schema and then populates the destination RDB by reading the data archive. After the destination RDB is restored, it can be queried or re-archived with A-SPARQL using SAQ. When the destination DBMS is an RDF triple store system, both the schema and data archive files are loaded directly into the triple store and can there be queried with SPARQL.

## 4. The archival benchmark ABench

The archival benchmark ABench consists of archival queries that select subsets of a relational database to archive as RDF, i.e. selecting specific tables, columns, and rows for archival using A-SPARQL. The relational database is generated by the Berlin benchmark dataset generator.

Table 1, Table 2, and Table 3 list the archival queries of ABench, together with the corresponding generated queries by SAQ. The archival queries are denoted  $A_i$  and the generated SPARQL queries are denoted  $Q_i$ .

Query A1 archives the entire database. The generated SPARQL query Q1 is an unbound-property query.

Query A2 archives the entire classes *product* and *offer* representing the entire RDB tables *product* and *offer*. The generated SPARQL query Q2 is an unbound-property UNION query of properties of the classes to archive.

Query A3 archives all values of some explicitly specified properties. Here the generated SPARQL query Q3 is a bound-property UNION query of three triple patterns, where the properties are known URIs in the WHERE clause.

Query A4 is similar to A3, i.e. it archives values of explicitly defined properties, but there are also conditions on the values of these properties. The generated SPARQL query Q4 becomes a bound-property UNION query of known property triple patterns.

Query A5 is similar to A2, but it constrains the properties of class *product* to archive. It retrieves the rows from table *product* for all attributes except those represented by the specified properties. The generated Q5 is an unbound-property query.

Query A6 archives all classes whose URIs match a defined string. The generated query Q6 is an unbound-property query. It should be executed by sending to the underlying RDB SQL queries selecting rows only from the tables represented by URIs matching the defined string.

Query A7 archives data for classes having properties whose URIs match a defined string. The generated query Q7 is an unbound-property query. It should be executed by sending to the underlying RDB SQL queries selecting rows only from the tables having attributes represented by properties that match the defined string.

Query A8 archives all properties and their values of a number of selected subjects. The generated SPARQL query Q8 is an unbound-property query with joins.

Query A9 archives all classes whose property values are literals containing a specific string. The generated query Q9 is an unbound-property query. It should be executed by sending SQL LIKE conditions on only such table attributes whose values are *not* represented by URIs.

Query A10 archives all properties of subjects related through a property to another given subject. The

relationship is represented by a foreign key in the underlying RDB. The generated query Q10 is an unbound-property query. It should be executed by sending SQL queries only to tables owning a foreign key for the table represented by the given subject.

## 5. Query processing in SAQ

In this section, first the structure of the RD-view is presented. Then an overview of the query processing steps in SAQ is presented. Finally the SAQ query

Table 1. ABench queries

Archival query	Generated CONSTRUCT query
<b>Query A1:</b> Archive the entire database	<b>Query Q1:</b>
<pre> ARCHIVE AS 'data1.nt', 'schema1.nt' FROM &lt;Products&gt; TRIPLES {?subject ?property ?value }; </pre>	<pre> CONSTRUCT { ?subject ?property ?value } FROM &lt;Products&gt; WHERE {?subject ?property ?value } </pre>
<b>Query A2:</b> Archive the entire classes <i>&lt;db:product&gt;</i> and <i>&lt;db:offer&gt;</i>	<b>Query Q2:</b>
<pre> ARCHIVE AS 'data2.nt', 'schema2.nt' FROM &lt;Products&gt; TRIPLES {?subject ?property ?value } WHERE{?subject rdf:type db:product .       db:product rdf:type rdfs:Class } UNION TRIPLES {?subject ?property ?value } WHERE{?subject rdf:type db:offer .       db:offer rdf:type rdfs:Class } </pre>	<pre> CONSTRUCT { ?subject ?property ?value } FROM &lt;Products&gt; WHERE {{ ?subject ?property ?value .         db:product rdf:type rdfs:Class .         ?subject rdf:type db:product }       UNION         { ?subject ?property ?value .         db:offer rdf:type rdfs:Class .         ?subject rdf:type db:offer }} </pre>
<b>Query A3:</b> Archive all values of the specific properties <i>&lt;db:product_label&gt;</i> , <i>&lt;db:offer_price&gt;</i> , <i>&lt;db:offer_webpage&gt;</i>	<b>Query Q3:</b>
<pre> ARCHIVE AS 'data3.nt', 'schema3.nt' FROM &lt;Products&gt; TRIPLES {?subject1 db:product_label ?value1 } UNION TRIPLES {?subject2 db:offer_price ?value2 } UNION TRIPLES {?subject3 db:offer_webpage ?value3 } </pre>	<pre> CONSTRUCT { ?subject1 db:product_label ?value1 .   ?subject2 db:product_price ?value2 .   ?subject3 db:offer_webpage ?value3 } FROM &lt;Products&gt; WHERE {{{?subject1 db:product_label ?value1 .   UNION   {?subject2 db:offer_price ?value2 }   UNION   {?subject3 db:offer_webpage ?value3 } </pre>
<b>Query A4:</b> Archive <i>products</i> property <i>pNum1</i> for values > 214 and property <i>pNum3</i> for values < 348, and <i>reviews</i> property <i>text</i> for values matching the string 'time' if the <i>reviews</i> have <i>rating4</i> > 8 .	<b>Query Q4:</b>
<pre> ARCHIVE AS 'data4.nt', 'schema4.nt' FROM &lt;Products&gt; TRIPLES {?subject1 db:product_pNum1 ?value1} WHERE { FILTER (?value1 &gt; 214 )} UNION TRIPLES {?subject2 db:product_pNum3 ?value2} WHERE {FILTER (?value2 &lt; 348 ) } UNION TRIPLES {?subject3 db:review_text ?value3} WHERE {   ?subject3 db:review_rating4 ?value4 .   FILTER REGEX(?value3, 'time' ) .   FILTER (?value4 &gt; 8 ) } </pre>	<pre> CONSTRUCT { ?subject1 db:product_pNum1 ?value1 .   ?subject2 db:product_pNum3 ?value2 .   ?subject3 db:review_text ?value3 } FROM &lt;Products&gt; WHERE {{{?subject1 db:product_pNum1 ?value1 .   FILTER (?value1 &gt; 214 ) }   UNION   {?subject2 db:product_pNum3 ?value2 .   FILTER (?value2 &lt; 348 ) }   UNION   {?subject3 db:review_text ?value3 .   ?subject3 db:review_rating4 ?value4 .   FILTER REGEX(?value3, 'time' ) .   FILTER (?value4 &gt; 8 ) } </pre>

Table 2. ABench queries

Archival query	Generated CONSTRUCT query
<b>Query A5:</b> Archive the class <code>&lt;db:product&gt;</code> except the RDF properties <code>&lt;db:product_label&gt;</code> and <code>&lt;db:product_pNum1&gt;</code>	<b>Query Q5:</b>
<pre> ARCHIVE AS 'data5.nt', 'schema5.nt' FROM &lt;Products&gt; TRIPLES { ?subject ?property ?value } WHERE { ?subject rdf:type db:product .   db:product rdf:type rdfs:Class .   FILTER (?property != db:product_label ).   FILTER (?property != db:product_pNum1 ) }; </pre>	<pre> CONSTRUCT { ?subject ?property ?value } FROM &lt;Products&gt; WHERE { ?subject ?property ?value .   ?subject rdf:type db:product .   db:product rdf:type rdfs:Class .   FILTER (?property != db:product_label ).   FILTER (?property != db:product_pNum1 ) } </pre>
<b>Query A6:</b> Archive entire classes whose URIs match the string <i>'product'</i>	<b>Query Q6:</b>
<pre> ARCHIVE AS 'data6.nt', 'schema6.nt' FROM &lt;Products&gt; TRIPLES { ?subject ?property ?value } WHERE { ?class rdf:type rdfs:Class .   ?subject rdf:type ?class .   FILTER REGEX (str(?class), 'product' ) }; </pre>	<pre> CONSTRUCT { ?subject ?property ?value } FROM &lt;Products&gt; WHERE { ?subject ?property ?value .   ?class rdf:type rdfs:Class .   ?subject rdf:type ?class .   FILTER REGEX (str(?class), 'product' ) } </pre>
<b>Query A7:</b> Archive all subjects having a property with URI matching the string <i>'homepage'</i> .	<b>Query Q7:</b>
<pre> ARCHIVE AS 'data7.nt', 'schema7.nt' FROM &lt;Products&gt; TRIPLES { ?subject ?property ?value } WHERE { ?subject ?property1 ?value1 .   FILTER REGEX (str(?property1), 'homepage')} </pre>	<pre> CONSTRUCT { ?subject ?property ?value } FROM &lt;Products&gt; WHERE { ?subject ?property ?value .   ?subject ?property1 ?value1 .   FILTER REGEX (str(?property1), 'homepage')} </pre>
<b>Query A8:</b> Archive all properties of the subjects from class <i>product</i> having <i>productFeature</i> 3 and 4, and <i>pNum1</i> > 348	<b>Query Q8:</b>
<pre> ARCHIVE AS 'data8.nt', 'schema8.nt' FROM &lt;Products&gt; TRIPLES { ?product ?property ?value } WHERE { ?product rdf:type db:product . ?product db:product_label ?label . ?product rdf:type db:product . ?product db:productFeature db:productFeature/_3 . ?product db:productFeature db:productFeature/_4 . ?product db:product_pNum1 ?pn1 . FILTER (?pn1 &gt; 348) } </pre>	<pre> CONSTRUCT { ?product ?property ?value } FROM &lt;Products&gt; WHERE { ?product ?property ?value . ?product rdf:type db:product . ?product db:product_label ?label . ?product db:productFeature db:productFeature/_3 . ?product db:productFeature db:productFeature/_4 . ?product db:product_pNum1 ?pn1 . FILTER (?pn1 &gt; 348) } </pre>
<b>Query A9:</b> Archive all classes whose literal property values contain the specific string <i>'symbols'</i>	<b>Query Q9:</b>
<pre> ARCHIVE AS 'data9.nt', 'schema9.nt' FROM &lt;Products&gt; TRIPLES { ?subject ?property ?value } WHERE { FILTER REGEX (?value, 'symbols') } </pre>	<pre> CONSTRUCT { ?subject ?property ?value } FROM &lt;Products&gt; WHERE { ?subject ?property ?value .   FILTER REGEX (?value, 'symbols')} </pre>

Table 3. ABench queries

Archival query	Generated CONSTRUCT query
Query <b>A10</b> : Archive all subjects related by a property to another subject identified by the URI <code>&lt;db:product/_2549 &gt;</code>	Query <b>Q10</b> :
<pre> ARCHIVE AS 'data10.nt', 'schema10.nt' FROM &lt;Products&gt; TRIPLES {?subject      ?property ?value } WHERE {db:product/_2549 ?relation ?subject} </pre>	<pre> CONSTRUCT {?subject ?property ?value } FROM &lt;Products&gt; WHERE {?subject      ?property ?value . db:product/_2549 ?relation ?subject } </pre>

rewrite optimizations are described.

### 5.1. The RD-view

The RD-view is defined in SAQ in a Datalog dialect [32]. A specialized RD-view for each given RDB is automatically generated by accessing the RDB catalogue. The RDB to RDF mapping in SAQ conforms to the direct mapping recommended by W3C [23], and more particularly to the augmented direct mapping proposed in [19], which is proven to guarantee information preservation.

We define a unique RDFS class for each relational table, except for *link tables* representing set-valued properties as many-to-many relationships. In addition, RDF properties are defined for each column in a table.

The RD-view is defined as a union of an S-view, representing the schema of the relational database, and a D-view, representing the data stored in the relational database.

The S-view represents all mappings between schema elements of the RDB and the corresponding RD-view classes and properties. It is defined in terms of six *mapping tables* that map relational schema elements to RDFS concepts. The system automatically generates default mappings in the mapping tables by accessing the RDB catalogue. The user can change the contents of the mapping tables to override default mappings in order to match some ontology or to limit data access. In order to guarantee unambiguous preservation the system requires unique URIs for classes and properties to be preserved.

In the used Datalog notation uppercase letters are used to denote constants while lowercase letters are used to denote variables.

The six mapping tables are the following:

- The *class table*,  $cMap(T, cid)$  maps relational table names  $T$  to RDFS class URIs  $cid$ .

- The *property table*,  $pMap(T, A, pid)$  maps relational column names  $A$  in table  $T$  to RDF property URIs  $pid$ .
- The *foreign key table*,  $fkMap(T, f, T', fkid)$  maps foreign keys  $f$  in table  $T$  referencing table  $T'$  to corresponding RDF property URIs  $fkid$ .
- The *many-to-many table*  $mmMap(L, T', T'', mmid)$  maps link tables  $L$  between tables  $T'$  and  $T''$  to corresponding property RDF property URIs  $mmid$ .
- The *type table*,  $typeMap(T, A, xsd)$  maps relational data types of relational attributes  $A$  in table  $T$  to corresponding XML Schema data types  $xsd$ .

The S-view definition itself is the same for any relational database and only the contents of the mapping tables are different. The S-view is defined as a large union of unions of sub-views representing relational schema concepts about tables, columns, types, primary keys, foreign keys, other constraints, and indexes. Since the S-view is complex but contains little data and its extent changes only when the database schema is altered, the S-view is materialized in main memory in SAQ.

Based on the S-view, i.e. on the imported RDB schema information, the system generates a D-view for each specific relational database. We opted to generate a D-view for each concrete database instead of defining a generic D-view, since this enables substantial query reduction at run time via specialization of the view definitions [16].

The D-view is defined in terms of *source predicates* representing the contents of relational tables, the above mapping tables, *URI-construct predicates*, for constructing URIs identifying rows in tables, and *literal-construct predicates* for constructing typed RDF literals. The D-view for an RDB is defined as a union of sub-views:

- For each non-foreign-key attribute, one *column view*  $C_{T,A}$  is generated. It represents as typed lit-

erals the values  $a$  of a column named  $A$  in table  $T$ :

$C_{T,A}(s,p,v) :-$		(1)
$R_T(a_1, \dots, a_k, \dots, a, \dots, a_r)$	AND	
$cMap(T, cid)$	AND	
$pMap(T, A, p)$	AND	
$rowid(cid, (a_1, \dots, a_k), s)$	AND	
$valueid(a, xsd, v)$	AND	
$typeMap(T, A, xsd)$		

$R_T$  in (1) is the source predicate representing the relational table  $T$ , and  $(a_1, \dots, a_k, \dots, a, \dots, a_r)$  is a tuple representing a row in  $T$ . The primary key of  $T$  is represented by the tuple  $(a_1, \dots, a_k)$ . Furthermore,  $rowid$  in (1) is the URI-construct predicate that creates a unique URI  $s$  representing a row identifier in  $T$  by concatenating the class associated with  $T$ , i.e.  $cid$ , and the primary key of a row, i.e.  $(a_1, \dots, a_k)$ . Finally  $valueid$  is the literal-construct predicate that creates a typed literal by concatenating the value  $a$  of the attribute  $A$  with the corresponding XML schema type  $xsd$ . Triples are not generated for NULL values in a RDB as in the direct mappings in [19].

- One *foreign key view*  $FK_F$  is generated for each foreign key relationship  $F$  for table  $T$  with foreign key attribute values  $(a_1, \dots, a_j)$  that references table  $T'$ . It represents foreign key values by URIs constructed by URI-construct predicates:

$FK_F(s,p,v) :-$		(2)
$R_T(a_1, \dots, a_k, a_1, \dots, a_j, \dots, a_r)$	AND	
$cMap(T, cid)$	AND	
$rowid(cid, (a_1, \dots, a_k), s)$	AND	
$fkMap(T, (a_1, \dots, a_j), T', p)$	AND	
$cMap(T', cid')$	AND	
$rowid(cid', (a_1, \dots, a_j), v)$		

- One *many-to-many relationship view*  $MM_L$  is generated for each link table  $L$  linking two tables  $T'$  and  $T''$ . It represents the values in link tables as URIs:

$MM_L(s,p,v) :-$		(3)
$cMap(T', cid')$	AND	
$R_L(a', a'')$	AND	
$rowid(cid', (a'), s)$	AND	
$mmMap(L, T', T'', p)$	AND	
$cMap(T'', cid'')$	AND	
$rowid(cid'', (a''), v)$		

- One *row class view*  $RC_T$  is generated for each non-link table  $T$  to represent the classes of its row identifiers:

$RC_T(s,p,v) :-$		(4)
$R_T(a_1, \dots, a_k, \dots, a_r)$	AND	
$cMap(T, cid)$	AND	

$rowid(cid, (a_1, \dots, a_k), s)$	AND	
$p = <rdf:type>$	AND	
$v = cid$		

A complete generated data view D-view in SAQ has the following structure:

$D\text{-view}(s,p,v) :-$		(5)
$OR_{T,A} C_{T,A}(s,p,v)$	OR	
$OR_{F} FK_F(s,p,v)$	OR	
$OR_{L} MM_L(s,p,v)$	OR	
$OR_{T} RC_T(s,p,v)$		

where  $OR_{T,A}$  denotes a disjunction over all attributes  $T.A$  in all tables  $T$  in the database,  $OR_F$  denotes a disjunction (union) over all foreign key relationships  $F$  in the database,  $OR_L$  denotes a disjunction over all link tables  $L$  in the database, and  $OR_T$  denotes a disjunction over all tables  $T$  in the database.

The D-view generated by SAQ for the ABench database contains the following sub-views:

- 67 column views
- 7 foreign key views
- 2 many-to-many relationship views
- 8 row class views.

## 5.2. Query processing steps in SAQ

The main steps of the query processing in SAQ are illustrated in Fig. 4. The *SPARQL parser* transforms the SPARQL query into a Datalog expression where each triple pattern (TP) in the query becomes a reference to the RD-view. The *view expander* recursively expands each RD-view reference in the query into a disjunctive expanded RD-view. The *view specializer* then enables a transformation called *view specialization* [16]. It looks up the mapping tables in each sub-view of the D-view at query processing time to replace variables in the expanded RD-view with corresponding URIs or literals. We call such a sub-view in the D-view, where the mapping tables have been looked up, a *specialized sub-view*. Then, since the RD-view is defined as a union of the S-view and the D-view, each TP in the query becomes a disjunction of the materialized S-view and the specialized sub-views in the D-view.

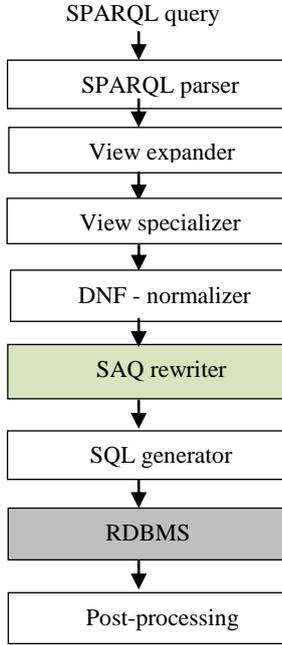


Fig. 4. SAQ Query processing.

The view specialization substantially reduces the disjunction for a TP depending on the TP type based on the following observations:

- i. The disjunction for an expanded bound-property triple pattern (BPTP) with the structure  $(?s P_i ?v)$  where  $P_i$  is unique is reduced into a single *property conjunction* for  $P_i$  representing the single sub-view in the D-view having the property  $p = P_i$ .
- ii. The disjunction for an expanded unbound-property triple pattern (UPTP) with the structure  $(S ?p ?v)$ , where  $S$  is a URI identifying a row in a table  $T$ , is reduced to a disjunction having those specialized sub-views in the D-view where the *subject*  $s$  is associated with  $T$ , i.e.  $s$  is mapped by a URI-construct predicate to rows in  $T$ .
- iii. The disjunction for an expanded UPTP structure  $(?s ?p ?v)$  cannot be reduced and remains a disjunction of the materialized S-view and the specialized sub-views in the D-view.

Later on the query is further simplified by eliminating common sub-expressions by unifying terms [10].

The *DNF-normalizer* transforms the simplified Datalog query into a disjunctive normal form (DNF) predicate. The DNF-normalized query has the following structure:

- a) A join between two BPTPs becomes a conjunction of the property conjunctions of the BPTPs.
- b) A join between a BPTP and a UPTP becomes several disjuncts in the DNF-predicate. The disjuncts are conjunctions between the property conjunction of the BPTP and each disjunct of the expanded UPTP.
- c) A join of two UPTPs becomes several disjuncts that combine the disjuncts of the two UPTPs.

For UNION queries, after normalization the UNION of its TPs becomes a DNF predicate containing the disjuncts of its DNF-normalized expanded TPs.

The *SPARQL rewriter* applies on the DNF-normalized and simplified query a number of query transformations that simplify the queries and improve the execution time. In particular, the *GCT rule* [30] transforms the DNF predicate into a more efficient Datalog representation by grouping those common terms in different disjuncts of the DNF predicate that can be translated to SQL. The query transformation rules are presented and evaluated below using the ABench benchmark.

Finally, the *SQL generator* generates an execution plan in SAQ that contains operators calling SQL. At execution time these SQL statements are sent to the RDB for execution. The generated plan also contains *post-processing* of such expressions that are not processed by the SQL engine, for example constructing URI objects, converting data types, and making union-all of sub-queries. All processing in the system is streamed so that no large intermediate collections are generated.

### 5.3. SAQ query transformations

The query rewriting optimizations for SPARQL queries selecting database parts to archive for different kinds of archival queries are described below. Since these queries often select sets of properties to archive they are mostly unbound-property queries, and therefore the query transformation optimizations for unbound-property queries are elaborated here. The processing and optimization of regular bound-property queries to an RD-view uses the techniques

described in [15][16][18] and is outside the scope of this paper.

All transformations are made on the DNF normalized SAQ predicate.

To describe the SAQ rewrite transformations, we use the following terminology:

- In a SPARQL query with a TP ( $?s ?p ?o$ ) we call the variable  $s$  a *subject variable*,  $p$  a *predicate variable*, and  $o$  an *object variable*.
- In a query, if the same variable is an object variable in one TP, e.g.  $s1$  in ( $?s ?p ?s1$ ), and a subject variable in another TP, e.g.  $s1$  in ( $?s1 ?p1 ?o1$ ), we call the variable  $s1$  a *subject-object join variable*. A subject-object join variable cannot be a literal, since subjects are always URIs.

Table 4 shows which of the query transformations below improve the execution times of queries in ABench.

Table 4. Rewrite transformations for SPARQL queries generated by ABench queries

<i>Rewrite Query</i>	<i>GCT</i>	<i>Eliminate S-view</i>	<i>is-literal</i>	<i>type-match</i>	<i>FKR</i>
<b>Q1</b>	X				
<b>Q2</b>	X	X			
<b>Q3</b>	X				
<b>Q4</b>					
<b>Q5</b>	X	X			
<b>Q6</b>	X	X			
<b>Q7</b>	X	X			
<b>Q8</b>	X	X			
<b>Q9</b>			X	X	
<b>Q10</b>	X	X			X

### 5.3.1. The GCT transformation

The *group common terms (GCT)* query transformation algorithm optimizes SPARQL queries in such a way that the RDB is accessed row-by-row instead of column-by-column. The *GCT* rule is applicable on queries selecting several attributes per table, in particular unbound-property queries. For example, *GCT* improves the performance of queries Q1, Q2, Q3, Q5, Q6, Q7, Q8, and Q10, since they all retrieve several table attributes with the same selection condition. The *GCT* is not applicable on queries Q4 and Q9, since they retrieve single table attributes with a single selection condition on each.

The *GCT* transformation is applied on a SPARQL query after DNF normalization. It factors out from the DNF predicate's disjuncts those conjunctions of common terms that can be translated to SQL queries. After *GCT*, the DNF predicate becomes a disjunction of conjunctions between terms that can be translated to SQL and disjunctions of the remaining terms with the translatable terms removed. The remaining terms cannot be expressed in SQL and must be post-processed.

In general, the steps of the *GCT* rewrite algorithm applied on a DNF predicate are the following:

- In a pre-step, normalize the variable names of the disjuncts in the DNF predicate so that the same variable names are used in equivalent predicate positions.
- Allocate a hash table that, for each extracted conjunction, maintains mappings to the disjuncts from which its terms have been extracted.
- For each disjunct in the DNF predicate, extract conjunctions of terms that can be translated

to SQL and put them in the hash table with the entire extracted conjunction as key along with a pointer to the rest of the disjunct as value.

- After the entire DNF predicate is scanned, go through the hash table and form for each key (extracted conjunction)  $c$  a conjunction between the SQL translatable predicate  $c$  and the post-processed remaining terms in the disjuncts from where  $c$  was extracted. Finally, form a disjunction of all the formed conjunctions.

The pseudo code of *GCT* algorithm is the following.

Function  $GCT(P, gf) \rightarrow GP$

*Input:*  $P$  - a DNF predicate with normalized variable names

$gf$  - a function that extracts a conjunction of specific terms, e.g.  $R_T$ , SQL comparisons from a conjunction

*Output:*  $GP$ :  $P$  grouped on the common terms

1. Allocate a hash table  $Ht$  for the common terms in disjuncts

2.  $GP := \text{null}$

3. **for each** disjunct  $D$  in  $P$  **do**

4. **if**  $D$  is an atom **then**  $GP := \text{orify}(GP, D)$

5. **else if**  $D$  is not a conjunction **then**

$GP := \text{orify}(D, GP)$

6. **else if**  $D$  has only one term **then**

$GP := \text{orify}(D, GP)$

7. **else**  $CT := gf(D)$

/\* $CT$  is a list of common terms\*/

8. **if**  $CT = \text{null}$  **then**  $GP := \text{orify}(D, GP)$

9. **else** put in  $Ht(\text{key}=CT) :=$

$\text{orify}(D \text{ with } CT \text{ removed}),$

(existing value for  $CT$  in  $Ht$ ))

10. **for each** ( $CT$  and  $\text{value}CT$ ) in  $Ht$  **do**

11.  $GP := \text{orify}(\text{andify}(CT, \text{value}CT), GP)$

14. **return**  $GP$

The function  $\text{orify}(x, y)$  forms a disjunction between predicates  $x$  and  $y$ , and  $\text{andify}(x, y)$  forms a conjunction.

Note that the processing is done in one pass and is therefore  $O(N)$ , where  $N$  is the number of disjuncts in the DNF predicate.

### 5.3.2. The is-literal reduction

The *is-literal* rule reduces SPARQL queries in such a way that SQL LIKE conditions are not issued on table attributes whose values are represented by URIs in the RD-view. This rule is applicable in queries where the type of an object variable in the query is restricted by some FILTER or other predicate to be a literal. For example, Q4, Q8, and Q9 restrict object variables to be literals by FILTER comparison predicates.

If an object variable is restricted to be a literal it cannot be bound to a URI by a URI-construct predicate. Therefore the *is-literal* rule eliminates those disjuncts from the expanded DNF normalized query where the object variable represents foreign keys or many-to-many relationships. This eliminates SQL code to access foreign keys and links, which reduces the number of generated SQL queries.

### 5.3.3. The type-match reduction

The *type-match* rule reduces SPARQL unbound-property queries so that SQL comparison conditions are issued only on attributes of correct literal types. For example, the LIKE predicate must be used on textual attributes of type (VARCHAR, TEXT, etc.), and arithmetic comparisons must be over numerical attributes (INT, DECIMAL, etc.). The rule reduces queries where the type of an object variable is restricted by some predicate to be of a specific literal type. For example, in Q9 the object variable *value* must be a literal string, which is inferred by the REGEX filter.

If an object variable is inferred to be of a specific literal type, it cannot be bound to a literal of another type by the literal-construct predicate. Therefore the *type-match* rule eliminates those disjuncts from the expanded DNF normalized query where the object variable represents relational column values of non-matching types. Thus SQL code to access those columns is not generated.

For example, the attribute *pNum1* in table *product* is a number while in Q9 the variable *value* must be a string, and therefore the SQL code generated will not access *pNum1*. The generated query for Q9 contains SQL LIKE conditions only for textual attributes (i.e. of type VARCHAR, TEXT, etc.). SQL LIKE conditions for other types of attributes are not generated.

### 5.3.4. Foreign key relationship (FKR) reduction

The *FKR* rule reduces SPARQL unbound-property queries where a subject-object join variable is shared between two UPTPs, which requires a foreign-key constraint.

The *FKR* rule eliminates those disjuncts from the expanded DNF normalized query where a join subject-object variable represents values that are not foreign keys in the underlying RDB. This reduces the number of SQL queries generated. SQL queries are generated only where there is a foreign key relationship between the tables referenced by the joined UPTPs.

For example, for Q10 *FKR* restricts the SQL generator to SQL queries only to the tables *producer*, *producttype* and *productfeature*, which possess foreign keys for the table *product* represented by *product:\_2549*.

### 5.3.5. Eliminate S-view reduction

The *eliminate S-view* rule reduces unbound-property queries so that an S-view subject is never joined with a subject constructed by the URI-

construct predicate. This rule assumes that user-overridden URIs in the mapping tables are not present in the D-view. This is enforced by the system.

The eliminate S-view is not needed for bound-property queries, because there all binding patterns are of form  $(s, P, o)$ , where  $P$  is a URI constant representing an attribute of a relational table. This URI is not allowed to be in the S-view. Therefore the S-view is always removed from BPTPs by the view specialization.

In contrast, the S-view will remain in UPTPs after specialization. In this case the *eliminate S-view* reduction is applicable when the subject variable of S-view is matched by a URI-construct predicate in a conjunction of the D-view, in which case the conjunction is eliminated. This occurs for queries where an UPTP is joined with another BPTP or UPTP on the subject or object variables. This rule is applicable on queries Q2, Q5, Q6, Q7, Q8 and Q10.

For example, Q2 is a SPARQL UNION unbound-property query where each UNION clause contains a join between the UPTP  $(?subject ?property ?value)$  and a BPTP on the variable  $?subject$ . Both Q7 and Q10 are unbound-property queries with a join between two UPTPs on a subject variable, i.e. the variable  $?subject$ .

## 6. Performance of archival queries

We evaluated the impact of the SAQ query rewrite optimizations for the generated SPARQL queries in ABench. We compared the performance of SAQ with Virtuoso RDF Views [26] and D2RQ [7], all systems accessing the same back-end MS SQL Server database. The experiment configuration was the following:

- a) The measurements were made on a PC Intel(R) Core(TM), 2Quad CPU Q9400 with 2.67 GHz and 8 GB RAM running 64-bits Windows 7 Professional.
- b) The DBMS was MS SQL server 2008 R2 running on a separate machine with Intel(R) Core(TM), i5 CPU 750 with 2.67 GHz and 8 GB RAM running 64-bits Windows 7 Professional. The SQL server was configured with 6 GB for the min and max server memory.
- c) The RDB data sets were generated by the Berlin benchmark data generator and loaded into the MS SQL Server. Table 5 summarizes the RDB sizes for the experimental data sets, together with the corresponding number of tri-

ples in the SAQ RD-view and the number of query result triples for Q1-Q10.

- d) Non-clustered, non-unique indexes were put on the columns *propertyNum1* and *propertyNum3* in the table *product*, and on the column *rating4* in the table *review* to speed up queries Q4 and Q8.
- e) For Virtuoso RDF Views, the RDF view to the underlying relational database was generated on the Virtuoso server (ver. 06.04.3132, Windows-64) using the Virtuoso Conductor tool. The SPARQL queries to this RDF view were run from a Java program, implementing a

	RDB1	RDB2	RDB3
<b>Phys. size</b>	184 MB	1.8 GB	9 GB
<b>Triples</b>	4.28 M	42.48 M	211.4 M
Q1	4.28 M	42.48 M	211.4 M
Q2	2.79 M	27.99 M	139.89 M
Q3	399.83 K	4.002 M	20.01 M
Q4	12.85 K	127.67 K	640.719 K
Q5	373.9 K	3.79 M	18.89 M
Q6	459.31 K	4.29 M	20.48 M
Q7	2.488 K	23.86 K	119.18 K
Q8	3.749 K	31.284 K	129.45 K
Q9	166	1.7 K	8.3 K
Q10	152	159	123

Jena Provider [42], which allows users to query Virtuoso RDF views from Java. Virtuoso was configured with the parameter *NumberOfBuffers* set to 340000 and the Java heap size was set to 4 GB.

- f) For D2RQ (v.08.1), the RDF view of the underlying RDBMS was generated by the D2RQ auto-generated mapping script [6]. In the generated script, we inserted the option `'d2rq:useAllOptimizations true'` to guarantee that full optimization would be used in D2RQ. The SPARQL queries were run from a Java program calling the D2RQ Engine through Jena2 [6]. The Java heap size was set to 4 GB.
- g) The default mappings of the analysed systems SAQ, Virtuoso RDF Views and D2RQ were used.

The following notation is used in the performance diagrams:

- **Virtuoso**: Virtuoso RDF Views configured with the system default mappings.

- **D2RQ**: D2RQ configured with the system default mappings.
- **SAQ-naive**: SAQ without any rewrites.
- **SAQ-ES**: SAQ with the *eliminate S-view* transformation.
- **SAQ-GCT**: SAQ with *GCT*.
- **SAQ-isLiteral**: SAQ with the *is-literal* transformation.
- **SAQ-Type**: SAQ with the *is-literal* and *type-match* transformations.
- **SAQ-FKR**: SAQ with *FKR*.
- **SAQ-FKR-ES**: SAQ with *FKR* and *eliminate S-view*.
- **SAQ-FKR-ES-GCT**: SAQ with *FKR*, *eliminate S-view* and *GCT*.

In all cases, the time spent in executing the query by the relational database followed by post-processing was measured, thus not including the time for preparing the SPARQL query by the respective system. The measured times did not include the backend DBMS query optimization time by excluding a first warm-up execution. The actual measurements were made five times and the mean values plotted. The standard deviation was less than 10% in all measurements.

### 6.1. Discussion of SAQ query performance

The performance of SAQ for the SPARQL queries generated by the archival queries in Abench is described below. Fig. 5 - Fig. 7 show the execution times for Q1-Q10 in seconds for different database sizes, SAQ strategies and other systems compared.

Table 6 summarizes the speed-up of the different rewrite optimizations in SAQ compared with **SAQ-naive** for the queries Q1-Q10. The speed-up is presented in the table as the improvement factor relative to the execution time of SAQ-naive. Table 6 also shows the number of SQL queries sent to the RDB for the different approaches.

#### 6.1.1. Impact of GCT

The performance **SAQ-GCT** for unbound-property queries was better than that of all other systems compared. *GCT* always improves performance substantially, by 55-70%, for queries scanning whole

RDB tables such as Q1 and Q2. Queries Q5 and Q6 are also unbound-property queries but they scan only few columns from different RDB tables and the improvement of *GCT* is lower (35-40%). However, for the very selective unbound-property queries Q7, Q8 and Q10, the improvement of *GCT* is much better, 100-200% for Q7 and Q8, and almost 300% for Q10. The reason is that without *GCT*, more SQL queries are sent to the RDB and the communication overhead dominates when the server time is insignificant.

The *GCT* optimization also somewhat improves bound-property queries selecting RDF properties that represent attributes in the same table, such as Q3. With *GCT* the properties are retrieved by a single SQL query per table, rather than one query per property without *GCT*. Thus for Q3 the number of SQL queries is reduced from 3 to 2.

#### 6.1.2. Impact of eliminate S-view

The *eliminate S-view* reduction (**SAQ-ES**) slightly improves (by 1-3%) the performance for unbound-property queries with one UPTP, i.e. Q2, Q5, Q6 and Q8, by reducing the number SQL queries. In contrast, *eliminate S-view* significantly improves the performance for unbound-property queries with more than one UPTP, where other reductions are not applicable. Thus, it improves the performance for Q7 very substantially, by 970-1400%, and the performance for Q10 substantially, by 70%.

#### 6.1.3. Impact of is-literal, type-match, and FKR

The improvement by the *is-literal* reduction (**SAQ-isLiteral**) for Q9 is 100-200%. The reason is that without *is-literal*, an additional nine SQL queries selecting foreign key values are generated.

The *type-match* reduction (**SAQ-Type**) further improves the performance for Q9 by 200-300%. The improvement is because without *type-match*, 40 unnecessary SQL queries selecting relational columns of type different than VARCHAR are generated and sent to the RDB.

The *FKR* reduction (**SAQ-FKR**) enormously improves the performance of Q10, by 60770-580900%, by eliminating 1344 SQL queries not joining on foreign keys.

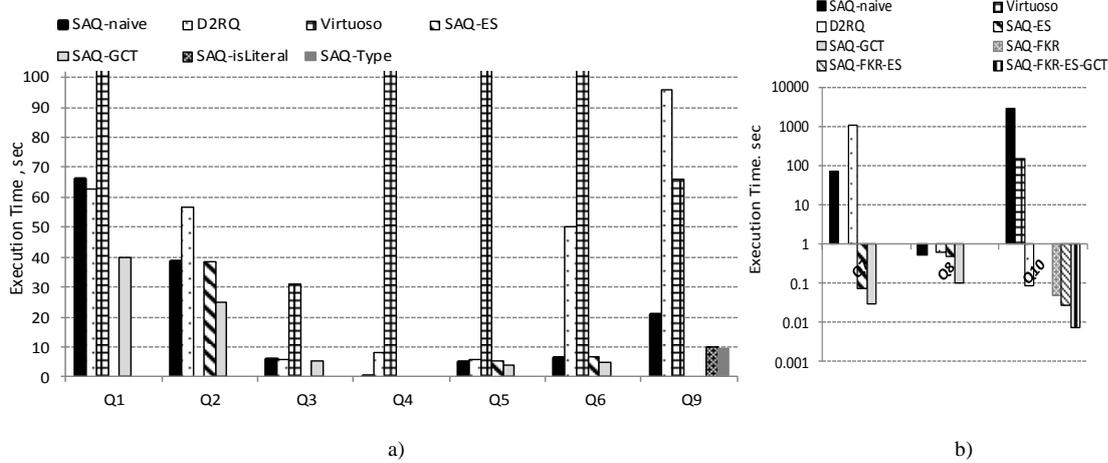


Fig. 5. Query Performance for Q1 – Q10, RDB1=184 MB.

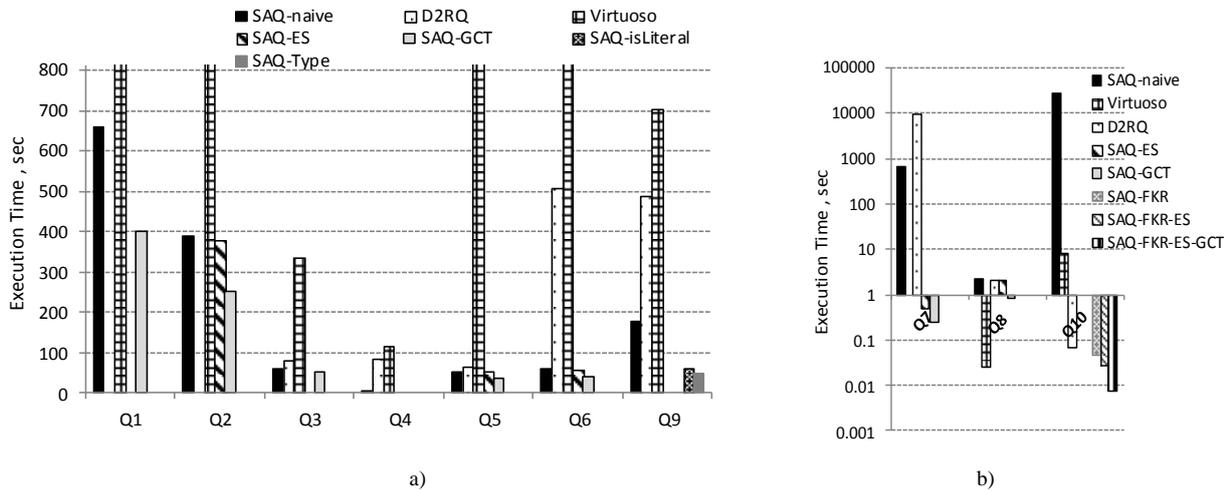


Fig. 6. Query Performance for Q1 – Q10, RDB2=1.8 GB.

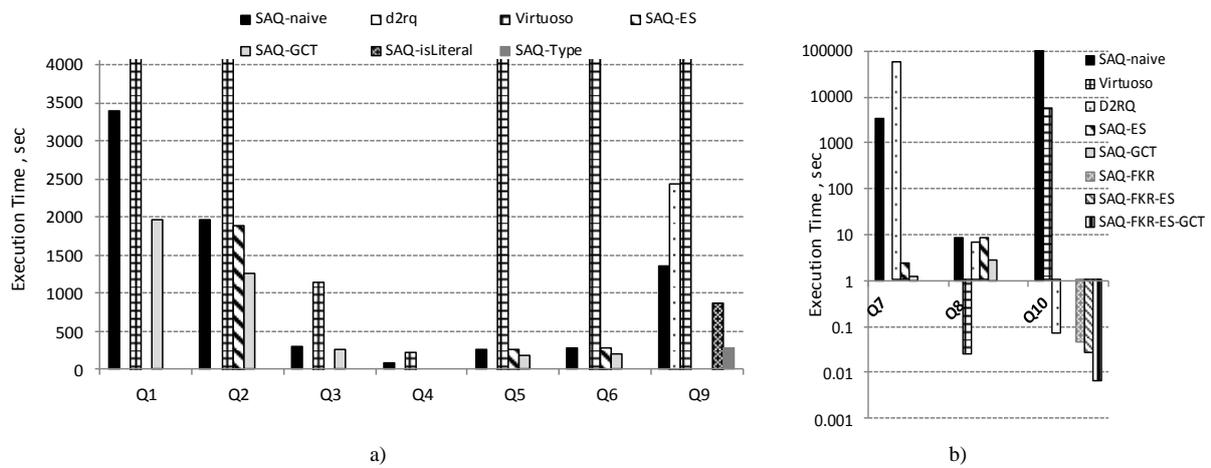


Fig. 7. Query Performance for Q1 – Q10, RDB3=9 GB.

### 6.1.1. Bound-property queries

The bound-property queries Q3 and Q4 are processed by *SAQ-naive* by specializing all the BPTPs in a conjunction into a single SQL query. Thus both Q3 and Q4 are processed by sending three SQL queries, each selecting a single relational attribute in a union.

### 6.2.1. Query performance of D2RQ

For *D2RQ* we used the profiling tool of MS SQL Server 2008 R2 to obtain the SQL queries sent to the DBMS.

Normally for bound-property queries such as Q3 and Q4, and for the unbound-property queries with one UPTP and no filter such as Q2, Q5 and Q8, *D2RQ* extracts RDB data column-wise as *SAQ-naive*. Thus, an optimization similar to *GCT* is not used by

Table 6. Speed-up (in times) for the SAQ rewrite optimizations and number of SQL queries sent to the RDB compared with SAQ-naive

<i>Rewrite</i>	<i>SAQ-naive</i>	<i>GCT</i>	<i>ES</i>	<i>is-literal</i>	<i>type-match</i>	<i>FKR</i>
<b>Query</b>						
Q1- speed up	1	1.63-1.72				
Q1-SQLqueries	84	10				
Q2- speed up	1	1.55	1.01 - 1.03			
Q2-SQLqueries	35	4	33			
Q3- speed up	1	1.15				
Q3-SQLqueries	3	2				
Q4 - speed up	1					
Q4-SQLqueries	3					
Q5- speed up	1	1.35 - 1.37	1.01 - 1.02			
Q5-SQLqueries	20	3	19			
Q6- speed up		1.37 - 1.39	1.015			
Q6-SQLqueries	37	5	34			
Q7- speed up	1	2-2.4	974 - 1450			
Q7-SQLqueries	102	2	16			
Q8- speed up	1	2-3	1.04 - 1.05			
Q8-SQLqueries	22	3	21			
Q9- speed up	1			2 - 3	3 - 4	
Q9-SQLqueries	76			67	27	
Q10- speed up	1	3.8	1.7			60775-580912
Q10-SQLqueries	1385	4 (SAQ-FKR-ES-GCT)	22 (SAQ-FKR-ES)			41 (SAQ-FKR)

### 6.2. Query performance of other systems

To analyse how the other systems process ABench queries, we measured their performance and, in addition, inspected what SQL queries were sent to the relational database.

For *D2RQ*, some measurements caused Java exception *GC overhead limit exceeded* and they are therefore not presented in Fig. 6 and Fig. 7. Similarly, *Virtuoso* failed in Fig. 5 with the exception message “*Query too large, more than 65000 variables in state*” for query Q2 and Q8 with the 184 MB dataset.

Since both *D2RQ* and *Virtuoso* don’t generate for their default mapping a triple with the form (*subject rdf:type rdfs:Class*), this triple was excluded from the definitions of queries Q2, Q5 and Q6 for these systems.

*D2RQ*, which explains why *SAQ-GCT* is faster than *D2RQ* for unbound-property queries.

For Q2, Q5 and Q8, despite *D2RQ* sending to the RDB the same number of SQL queries as *SAQ-naive*, it scales somewhat worse because of the view specialization of SAQ.

For Q4, *D2RQ* sends to the RDB three SQL queries without comparisons, while *SAQ-naive* sends three queries each including a comparison. Therefore *D2RQ* selects a much larger result set than needed and its performance is much worse than that of SAQ, since it does not utilize any index. Furthermore, Q4 could not be successfully processed by *D2RQ* for the largest data set of 9 GB, since the Java exception *GC overhead limit exceeded* was triggered. The reason is that the system tried to materialize in the Java heap entire columns retrieved from the RDB.

For Q1, which selects all RDB tables, *D2RQ* makes a special optimization and sends fewer queries to the RDB and therefore outperforms *SAQ-naïve*. However, *SAQ-GCT* still outperforms *D2RQ* for Q1 because even fewer SQL queries are generated.

To process Q6, *D2RQ* sends to the RDB an SQL query for each column in the RDB and does the filtering as post-processing, which does not scale well. *SAQ-naïve* scales much better for Q6, since the view specialization reduces the query substantially. The only SQL queries evaluated are those that select columns from tables fulfilling the filter condition, i.e. the tables *product*, *productfeature*, and *producttype*.

For Q7, *D2RQ* sends to the RDB around 1000 SQL queries accessing all RDB tables. The view specialization of *SAQ-naïve* outperforms *D2RQ* here by sending to the RDB much fewer SQL queries, i.e. 107 queries accessing only the tables whose attributes fulfil the filter condition, i.e. the *producer* and *vendor* tables.

Q9 is processed by *D2RQ* by sending to the RDB 17 SQL queries selecting values from all tables row-wise. All filtering is done by post-processing the extracted RDB values, which does not scale. In contrast, SAQ utilizes *is-literal* and *type-match* to send SQL queries with LIKE predicates to the DBMS, which utilizes indexes.

*D2RQ* uses an optimization similar to the *FKR* optimization of SAQ for processing Q10. Here *D2RQ* sends to the RDB 21 SQL queries selecting column-wise values from the tables *producer*, *producttype* and *productfeature* as *SAQ-FKR-ES*. *GCT* further improves the performance of *SAQ-FKR-ES-GCT*.

### 6.2.2. Query performance of *Virtuoso*

The debug logging of *Virtuoso* was used to investigate how it translates the SPARQL queries and what SQL queries were sent to the RDB.

The bound-property UNION query Q3 with no filters is processed by *Virtuoso* by sending to the RDB the same SQL queries as *SAQ-naïve*. Here, *Virtuoso* performs worse than *SAQ-naïve* since it tries to materialize in main memory the large result set, while SAQ streams the result.

For the bound-property UNION query Q4, which has a filter on each selected property, *Virtuoso* sends two SQL queries with arithmetic comparisons exactly as *SAQ-naïve* and additionally one or many parameterized SQL queries that do not contribute to the result. The latter is the reason for the worse performance.

For the selective unbound-property query Q8, *Virtuoso* sends to the RDB SQL queries extracting product data in a column-wise manner, as *SAQ-naïve*. An optimization such as *GCT* is not used. Despite that, for Q8 *Virtuoso* outperforms *SAQ-GCT* since the result set is very small and cached on the client during the first run, while for the next runs it is read directly from main memory.

*Virtuoso* processes the non-selective unbound-property query Q1 by sending to the RDB an SQL query for each column as *SAQ-naïve*. It scales much worse than *SAQ-naïve* since it does not use *GCT* and tries to materialize in memory the very large result set.

For the unbound-property queries Q2, Q5 and Q6, *Virtuoso* sends to the RDB an SQL query for each selected column as *SAQ-naïve* and in addition a large number of parameterized queries. For the larger database more than 1000 queries are sent to the RDB. Therefore it performs very badly.

Query Q7 could not be processed by *Virtuoso*. The following message was received: *The SPARQL optimizer has failed to process the query with reasonable quality.*

The text matching query Q9 is processed by *Virtuoso* by sending to the RDB SQL queries selecting column-wise attribute values from all tables followed by filtering as post-processing, which does not scale. Optimizations similar to *is-literal* and *type-match* are not used.

Finally, for Q10 *Virtuoso* sends to the RDB a number of SQL queries accessing all tables in the RDB. It does not use an optimization similar to *FKR* but nevertheless outperforms *SAQ-naïve*, since much fewer SQL queries are sent to the RDB. *SAQ-FKR-ES-GCT* is still faster.

## 7. Related work

The related work on preservation of relational databases, mapping relational databases to RDF, and query processing of unbound-property queries is reviewed.

### 7.1. Long-term preservation of relational databases

Testbed [40], SIARD [41] and RODA [20] are projects that have developed strategies for long-term preservation of relational databases based on XML. In both Testbed and RODA the data and metadata of relational databases are preserved as XML. SIARD

has an own format for preservation which is based on XML and SQL1999, and the industry standard ZIP. In contrast, in SAQ we use RDF to represent the relational database to archive. Both XML and RDF are neutral data formats that don't rely on current DBMS technology and provide hardware and software independence. These make both of them suitable for long-term preservation of databases. However, RDF has the following advantages comparing to XML. In RDF the identifiers are URIs which are universal global unique identifiers that allow identifiers from one database or table to be linked with identifiers from other data. Data can be represented as XML in many different ways depending on a defined DTD or XML schema [21] while the RDF-Schema (RDFS) in RDF provides standard meta-data representation for describing all kinds of data, including relational databases [11]. Furthermore, representing relational data as RDF allows migration from RDBs to RDF repositories which are gaining increasing popularity compared to XML native repositories.

In the above mentioned related approaches the entire relational database, both the data and schema are migrated into XML or XML based format and stored in a file. By contrast, in SAQ we provide selective archival of user-specified parts of a relational database using an extended SPARQL query language, A-SPARQL

### 7.2. Mapping and querying relational databases as RDF

Virtuoso RDF Views [26][27], D2RQ [7][8], and SquirrelRDF [35] are other systems that allow mapping of relational tables and views into RDF to make them queriable by SPARQL. These systems implement compilers that translate SPARQL directly to SQL. In contrast, SAQ first generates Datalog queries to a declarative RD-view of the relational database, and then transforms the SPARQL queries to SQL, based on logical transformations. We have shown that query transformations on this representation significantly improve performance for SPARQL unbound-property queries selecting RDB contents to archive.

The system closest to SAQ is Ultrawrap [17][18] where, like in SAQ, an RDF view over a relational database is generated as a union of sub-views. While the RDF view in Ultrawrap is defined in SQL in a specific SQL dialect, in SAQ the view is defined in a Datalog dialect and thus it is independent on the RDBMS. Furthermore, since the view in Ultrawrap is

defined in a concrete RDBMS the query optimizations are also dependent on the RDBMS, and thus the performance measurements in [18] show different results in different systems. By contrast, in SAQ the proposed optimizations are made in the SAQ query processor and are not dependent on the back-end RDBMS.

Unlike SAQ, neither D2RQ, nor Virtuoso, nor Ultrawrap includes the schema view in the RDF view of RDBs. The inclusion of the S-view is very important when archiving relational databases, since the database schema is needed to recreate an archived database. The logical rewrites of SAQ enable scalable processing over full RDF views, including the schema part.

### 7.3. Optimizing unbound-property queries and disjunctive queries

We did not find any published data on how D2RQ compiles SPARQL queries into SQL. The documentation on Virtuoso is very limited. However, by using the profiling tool of the DBMS and the debug logging of Virtuoso, we were able to analyse what queries were actually sent to the underlying RDB. This showed that neither D2RQ nor Virtuoso uses optimization for unbound-property queries similar to the SAQ rewrite optimizations *GCT*, *is-literal* and *type-match*. D2RQ uses an optimization similar to *FKR* to process queries with a join variable shared between two UPTPs, such as Q10.

SquirrelRDF also allows SPARQL queries to relational tables, but it does not support unbound-property SPARQL queries.

Ultrawrap tries to completely translate SPARQL to semantically equivalent SQL, without any pre- or post-processing. This is problematic for unbound-property queries, and in [18] the authors state that a SPARQL unbound-property query “doesn't have a concise, semantically equivalent SQL query”. In contrast, SAQ generates an execution plan where SQL queries are submitted to an RDB, and then streamed post-processing constructs URIs, RDF literals, and triple. We could not find any published data on how Ultrawrap translates SPARQL unbound-property queries to SQL. Nevertheless, there are experimental results with Ultrawrap on unbound-property queries in [18] and it can be concluded from these that Ultrawrap has no special optimizations. It is shown in [18] that an Ultrawrap query for unbound-property query performs worse than a “Native SQL” query, i.e.

a translated SQL query did not exploit the relational model as well as a native query.

Rather than semantic transformations directly on the original SPARQL code, SAQ makes all query transformations on Datalog expressions. The advantage with this approach is that it is a very general, well understood, and easy to extend with new transformation rules, if so needed. We have shown that the approach is possible without loss of efficiency.

Work on optimizing disjunctive database queries in general is described in [1][14][25]. The closest work to *GCT* is the combinatorial algorithm [25], which merges disjuncts with common sub-expressions in general disjunctive logical expression in order to avoid repeated evaluation of the same predicate on the same tuple. In contrast, the purpose of *GCT* is to group in a DNF predicate query fragments that can be translated to SQL, and *GCT* is therefore a simpler linear algorithm. The idea of bypass evaluation of disjunctive queries in [1][14] is based on implementing specialized operators that produce two output streams: the true-stream of the tuples that fulfil the operator's predicate and the false-stream of the tuples that do not match. The main benefit of the technique of bypass evaluation is in eliminating duplicates by avoiding unnecessary join operators. The purpose of *GCT* is not duplicate elimination, but to rewrite complex disjunctive queries for faster execution.

## 8. Conclusions and future work

An approach was presented for selective scalable long-term archival of RDBs as RDF in terms of SPARQL queries, implemented in the SAQ system. The proposed approach is suitable for archiving research data used in scientific publications where it is desirable to preserve only selected parts of an RDB. The archival of user-specified parts of a RDB is specified using an extension of SPARQL, A-SPARQL, having an archival statement for selective archival.

The SAQ system for long-term preservation of relational databases follows conceptually the OAIS reference model. In particular, this work concentrates on the functionality of the Ingest component in the OAIS model on generating the content information when preserving relational database content as RDF.

To evaluate the performance of typical archival queries, the ABench was defined that archives selected parts of databases generated by the Berlin benchmark data generator. In experiments, the SAQ opti-

mization strategies were evaluated by measuring the performance of A-SPARQL queries selecting triples for archival queries in ABench.

SAQ automatically generates an RDF view of an RDB called the *RD-view*. The RD-view can be queried and archived with A-SPARQL queries that are translated into SQL queries sent to the RDB. An archival query internally generates a corresponding CONSTRUCT SPARQL query. Since the archival query usually selects sets of attributes of tables to archive, the generated CONSTRUCT SPARQL query is typically an unbound-property or UNION query. To achieve scalable data preservation and recreation for such queries, SAQ uses some special query rewriting optimizations presented in this paper.

Using ABench queries and data generated by the Berlin benchmark generator, the rewriting optimizations were experimentally shown to improve query execution time compared with naïve processing. Compared with not using the optimizations, they reduce the number of SQL queries to execute and retrieve data in relational row order rather than in column order. The performance of SAQ was compared with that of other systems that support SPARQL queries to views of existing relational databases. It was shown experimentally that SAQ with the rewrite optimizations performs better than those systems for all queries returning large results. In general, the SAQ optimizations are useful not only for archival queries, but also for unbound-property and UNION queries.

Future work will include defining and evaluating new query rewrites for further improving the performance, for example for free text searches of RDB when data are archived based on LIKE. Another extension would be to perform the archiving based on what is reachable from a set of root data nodes, i.e. based on SPARQL queries with path expressions [31].

## References

- [1] A. Kemper, G. Moerkotte, K. Pethner, and M. Steinbrunn, Optimizing Disjunctive Queries with Expensive Predicates, in Proc. ACM SIGMOD, Conf. Management of Data, pp. 336-347 (1994)
- [2] A. Seaborne, D. Steer, and S. Williams, SQL-RDF, <http://www.w3.org/2007/03/Rdfrdb/papers/seaborne.html> (2007)
- [3] C. Bizer and A. Schultz, Berlin SPARQL Benchmark (BSBM) Specification-V3.1., <http://www4.wiwi.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/index.html> (2011).

- [4] C. Bizer and A. Schultz, Berlin SPARQL Benchmark (BSBM) Specification, Data Generator and Test Driver, <http://www4.wiwiw.fuberlin.de/bizer/BerlinSPARQLBenchmark/spec/20080912/index.html#datagenerator>
- [5] C. Bizer and A. Schulz, The Berlin SPARQL Benchmark, *Journal of Semantic Web and Information Systems*, special issue on scalability and performance of semantic web systems, Vol. 5, Issue 2, pp 1-24 (2009)
- [6] C. Bizer, R. Cyganiak, G. Garbers, O. Maresch and C. Becker, The D2RQ Platform v0.7 - Treating Non-RDF Relational Databases as Virtual RDF Graph. <http://www4.wiwiw.fuberlin.de/bizer/d2rq/spec/> (2009)
- [7] C. Bizer, and A. Seaborne, D2RQ-Treating Non-RDF Databases as Virtual RDF Graphs, Poster at 3rd International Semantic Web Conference (2004)
- [8] C. Bizer and R. Cyganiak, D2R Server-Publishing Relational Databases on the Semantic Web, Poster at the 5th International Semantic Web Conference (2006)
- [9] D. Giaretta, *Advanced Digital Preservation*, Springer (2011)
- [10] G. Fahl and T. Risch, Query Processing over Object Views of Relational Data, *The VLDB Journal*, Vol. 6, No. 4, pp 261-281 (1997)
- [11] H. Stuckenschmidt and F. Harmelen, *Information Sharing on the Semantic Web*, Springer, ISBN 3-540-20594-2 (2005)
- [12] J. Hunter, *Scientific Publication Packages – A Selective Approach to the Communication and Archival of Scientific Output*, *The International Journal of Digital Curation*, iss. 1 vol. 1, pp. 33-52, (2006)
- [13] J. Masanès, *Web Archiving*, Springer (2006)
- [14] J. Claussen, A. Kemper, K. Peithner and M. Steinbrunn, Optimization and Evaluation of Disjunctive Queries, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12, No 12, March/April (2000)
- [15] J. Petrini and T. Risch, Processing queries over RDF views of wrapped relational databases, in *Proceedings of the 1st International Workshop on Wrapper Techniques for Legacy Systems, WRAP 2004*, Delft, Holland (2004)
- [16] J. Petrini, *Querying RDF Schema Views of Relational Databases*, PhD Thesis, Uppsala University, Department of IT, ISSN1104-2516, <http://www.it.uu.se/research/group/udbl/Theses/JohanPetriniPhD.pdf> (2008)
- [17] J. F. Sequeda and D. Miranker, SPARQL Execution as Fast as SQL Execution on Relational Data, Poster at the 10th International Semantic Web Conference (2011)
- [18] J. F. Sequeda and D. Miranker, Ultrawrap: SPARQL Execution on Relational Data, Technical Report [http://apps.cs.utexas.edu/tech\\_reports/reports/tr/TR-2078.pdf](http://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2078.pdf)
- [19] J.F. Sequeda, M. Arenas and D. P. Miranker, On Directly Mapping Relational Databases to RDF and OWL, *WWW 2012*, April 16–20, 2012, Lyon, France
- [20] J. C. Ramalho, M. Ferreira, L. Faria, and Rui Castro. Relational Database Preservation through XML modelling. In *Extreme Markup Languages 2007*, (2007)
- [21] J. Tauberer, What is RDF and what is it good for?, <http://www.rdfabout.com/intro/#Comparing%20RDF%20with%20XML>, (2008)
- [22] L. Sweeney, k -Anonymity: A Model for Protecting Privacy, *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10 (5), 2002; 557-570, (2002)
- [23] M. Arenas, A. Bertails, E. Prud'hommeaux, and J. Sequeda, A Direct Mapping of Relational Data to RDF, *W3C Recommendation 27 September 2012*, <http://www.w3.org/TR/rdb-direct-mapping/>, (2012)
- [24] M. Jobst, *Preservation in Digital Cartography*, Springer (2011)
- [25] M. Muralikrishna and David J. DeWitt., Optimization of Multiple-Relation Multiple-Disjunct Queries, *PODS '88 Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART*, pp. 263-275 (1988)
- [26] O. Erling, *Declaring RDF views of SQL Data*, *W3C Workshop on RDF Access to Relational Databases*, 25-26 October, Cambridge, MA, USA (2007)
- [27] O. Erling and I. Mikhailov, *RDF Support in the Virtuoso DBMS*, Springer, ISSN: 1860-949X (Print) 1860-9503 (Online), in *Studies in Computational Intelligence*, Vol. 221(2009)
- [28] P. Buneman, S. Khanna, K. Tajima and W. Tan, Archiving Scientific Data, *ACM Transactions on Database Systems*, Vol. 29, No. 1, pp. 2-42, (2004)
- [29] S. Higgins, The DCC Curation Lifecycle Model, *International Journal of Digital Curation*, Vol. 3, No. 1, pp. 134-140 (2008)
- [30] S. Stefanova and T. Risch, Optimizing Unbound-property Queries to RDF Views of Relational Databases, *7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2011)*, Bonn, Germany, October 2011.
- [31] St. Harris St and A. Seaborne, *SPARQL 1.1 Query Language W3C Working Draft 24 July 2012*
- [32] W. Litwin, and T. Risch, Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6 (1992)
- [33] U. Borghoff, et al., *Long-Term Preservation of Digital Documents*, Springer (2010)
- [34] N-triples, *W3C RDF Core WG Internal Working Draft*, <http://www.w3.org/2001/sw/RDFCore/ntriples/>
- [35] SquirrelRDF, <http://jena.sourceforge.net/SquirrelRDF/>
- [36] Reference Model for an Open Archival Information System (OAIS) (2012). Recommended Practice CCSDS 650.0-M-2, Magenta Book, Consultative Committee for Space Data Systems
- [37] Selection of Research Data; Guidelines for appraising and selecting research data; A report by DANS and 3TU Datacenter, [http://www.surf.nl/nl/themas/openonderzoek/cris/Documents/SURF-share\\_Collectioneren\\_Selection%20of%20Research%20Data\\_DANS\\_3TU\\_DEft.pdf](http://www.surf.nl/nl/themas/openonderzoek/cris/Documents/SURF-share_Collectioneren_Selection%20of%20Research%20Data_DANS_3TU_DEft.pdf)
- [38] Ad hoc Strategic Committee on Information and Data, Final Report to the ICSU Committee on Scientific Planning and Review, <http://www.icsu.org/publications/reports-and-reviews/scid-report/scid-report.pdf>, (2008)
- [39] National Science Board: Long-Lived Digital Data Collections: Enabling Research and Education in the 21st Century, NSB 05-40, <http://www.nsf.gov/pubs/2005/nsb0540/nsb0540.pdf> (2005)
- [40] Digital Preservation Testbed. From Digital Volatility to Digital Permanence: Preserving Databases. Technical report, Dutch National Archives and the Dutch Ministry of the Interior and Kingdom Relations, (2003)
- [41] Swiss Federal Archives SFA Unit Innovation and Preservation. SIARD Format Description. Technical report, Federal Department of Home Affairs, FDHA, Berne, (2008)
- [42] Virtuoso Jena Provider, OpenLink Virtuoso Universal Server: Documentation, <http://docs.openlinksw.com/virtuoso/rdfnativestorageproviders.html#Rdfnativestorageprovidersjena> (2009)
- [43] Allegro Graph, <http://www.franz.com/agraph/allegrograph/>
- [44] Sesame, <http://www.openrdf.org/>
- [45] Virtuoso Universal Server, <http://virtuoso.openlinksw.com/>