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* 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 relational databases as RDF triples, implemented in the *SAQ* (Semantic Archive and Query) system. In SAQ an RDF view of a relational database, called the *RD-view*, is automatically generated. The RD-view can be queried by arbitrary SPARQL queries. Long-term preservation as RDF of selected parts of a database is specified in an extended SPARQL dialect, *A-SPARQL*, as an *archival query*. A-SPARQL provides flexible selection of data to be archived in terms of SPARQL-like queries to the RD-view, which produces a *data archive* file containing the RDF-triples representing the relational data content to be preserved. It 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 the SPARQL queries selecting triples for archival queries in ABench. The performance of the same 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

For long-term preservation of data, it is desirable for the contents of a database to be unloaded in a neutral format, so that it can be reconstructed and queried after a very long time using current technologies for data representation, which are continuously evolving. To provide a database technology-independent format for long-term preservation of data, we propose an RDFS-based neutral format. Semantic Web technologies and in particular RDF and RDFS [11] seem promising for long-term preservation of databases.

In this paper we present an approach for scalable long-term archival of relational databases (RDBs) as RDF, implemented in the *SAQ* (Semantic Archive and Query) system. The proposed approach provides long-term preservation of RDBs and a migration path between different kinds of relational and RDF DBMSs.

It is usually desirable to preserve only parts of an RDB, e.g. only data about a specific set of artefacts in the database. Therefore SAQ provides selective archival of user-specified parts of a RDB using an extended SPARQL query language, A-SPARQL.

To map a relational database into RDFS, SAQ automatically generates an RDF view of the relational

*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.

database called the *RD-view*. The RD-view is defined in terms of an RDFS ontology for describing RDB schemas in general. The generated RD-view can be queried with SPARQL queries that are translated by the SAQ query processor into SQL statements sent to the RDBMS. In order to preserve both schema and data from an RDB, it is important to represent not only the contents of the RDB as RDFS, but also the schema. Therefore the RD-view contains 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 [8][9][18], the data view mappings are defined according to the direct mapping requirements in [3].

To select the parts of an RDB to archive, a SAQ user defines an *archival query* to the RD-view in A-SPARQL. In the archival query the parts of the database to archive are specified either as a list of RDF classes or properties in the RD-view, or as a SPARQL-like query to the RD-view. In addition, the *destination* clause specifies where the archived relational data and schema are stored. For example, the classes representing the tables named *product* and *offer* in the relational database *Products* of the Berlin Benchmark dataset [1][6] are archived with this archival query:

```
ARCHIVE AS 'data.nt', 'schema.nt'
FROM <Products>
CLASSES  saq:product, saq:offer ;
```

In the example 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'* in the example.

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. [26].

An archival query in A-SPARQL internally generates a corresponding SPARQL query to select the triples of the database to archive. 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, which makes the generated SPARQL queries UNION CONSTRUCT queries.

Furthermore, 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 *unbound-property triple patterns (UPTP)*, and the queries having such triple patterns *unbound-property queries* [20]. Those triple patterns (*TP*s) where the properties are URIs representing RDF properties in the RD-view we call *bound-property triple patterns (BPTP)* and the queries having only BPTPs *bound-property queries*. To achieve scalable data preservation and recreation, we developed some special query rewriting optimizations for optimizing archival 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 that archive selected parts of databases generated by the Berlin benchmark data generator [5]. A new benchmark was developed since the archival queries are often CONSTRUCT unbound-property queries with UNION clauses, which is not covered by any existing benchmark.

In experiments, the SAQ optimization strategies were evaluated by measuring the performance of the SPARQL queries generated for the archival queries in ABench. These 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, Section 3 presents the SAQ system and the archival queries in A-SPARQL, 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.

Motivating example

Fig. 1 shows a small RDB called *Products* that 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 *productfeature-product* is a many-to-many link table between the tables *product* and *productfeature*.

Table *product*

pnr	label	pNum1	producer
1	cicatrices	100	2
2	emulsifying	450	3

Table productfeature

pfnr	publishDate
3	2000-06-22
4	2000-07-08

Table *productfeatureproduct*

product	productFeature
1	3
2	3
2	4

Table *producer*

prodnr	country
2	DE
3	SE

Fig. 1. RDB Products.

A user wants to preserve data about products produced in Sweden and having *pNum1* > 348. The relational data about products have to be preserved as RDF, but also sufficient meta-data to allow the preserved data to be later reconstructed as a new database.

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              saq:product .
?product   saq:product_producer  ?producer .
?producer  saq:producer_country  'SE' .
?product   saq: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*

pnr	label	pNum1	producer
2	emulsifying	450	3

Table *R_productfeature*

pfnr
3
4

Table *R_producer*

prodnr
3

Table *R_productfeatureproduct*

product	productFeature
2	3
2	4

Fig. 2. Reconstructed RDB.

2. The Semantic Archive and Query system

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.

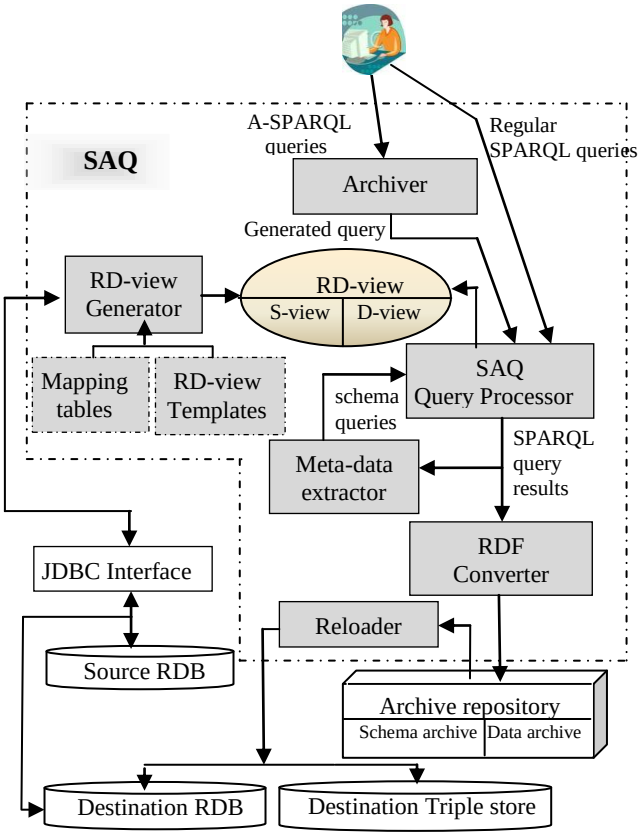


Fig. 3. SAQ.

The *RD-view generator* automatically generates the *RD-view* over the *source RDB* by reading the database schema though 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. The *SAQ query processor* executes arbitrary SPARQL queries to the RD-view by accessing the source RDB through the JDBC interface.

2.1. Archival queries

When the source RDB, or parts of it, is to be preserved as RDF, a user specifies an *archival query* in A-SPARQL. This archival query is translated by the *archiver* into a regular SPARQL *generated query*.

Archival queries have the following syntax:

```
ARCHIVE AS data archive file,
           schema archive  file
FROM RD-view URI
[CLASSES list of classes]
[PROPERTIES list of properties]
[TRIPLES {archived triple pattern}
 WHERE {archive restriction}
 UNION
 TRIPLES {archived triple pattern}
 WHERE {archive restriction}
 UNION
 …                                    ]
```

For example:

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

2  ARCHIVE AS 'data2.nt', 'schema2.nt'
   FROM <Products>
   CLASSES saq:product

3  ARCHIVE AS 'data3.nt', 'schema3.nt'
   FROM <Products>
   PROPERTIES  saq:product_label,
               saq:producttype_label

4 ARCHIVE AS 'data4.nt', schema4.nt'
   FROM <Products>
   TRIPLES {?subject ?property ?value}
   WHERE {
   FILTER REGEX(str(?property),'label')}
```

An archival query is specified by an *ARCHIVE* statement where the *data archive file* and the *schema archive file* are specified. The *FROM* clause specifies the *RD-view URI* representing the database to archive. The body of an archival query is specified as an explicit RDF *list of classes* or *list of properties* to archive. It can also be specified by a (union of) *TRIPLES* expression that selects the triples to archive specified as a single *archived triple pattern* whose bindings are selected from the RD-view by the *archive restriction* in the WHERE clause. An archive restriction is a general conjunctive SPARQL WHERE clause to restrict the triples to archive. The union of several TRIPLES expressions can be specified to archive several different RD-view triple patterns.

In the example, Query 1 archives the entire database as N-triples [23] and stores it as a data archive file *'data1.nt'* and schema archive file *'schema1.nt'*. Query 2 archives only the RDFS class having the URI <*saq:product*> and its properties. Query 3 archives only the RDF properties having URIs <*saq:product_label*> and <*saq:producttype_label*>. Finally, query 4 archives all subjects having properties whose URIs match *label*.

When an archival query is processed by the archiver and translated into a corresponding generated query in SPARQL, this is sent to the *SAQ query processor*. The generated query retrieves the data to ar-

chive from the RDB. Other SPARQL queries are sent directly to the SPARQL query processor.

An archival query is straight-forward to translate into a CONSTRUCT SPARQL query, very often a CONSTRUCT-UNION query when unions of sets of triples are archived. The translation rules are presented in Appendix 1.

During the execution of the generated query, the property indicators of the triples to be archived are collected by the *meta-data extractor* while scanning the result stream of the generated query. When all data to archive have been scanned, the meta-data extractor joins the collected properties with the S-view by issuing s*chema queries*.

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

2.2. Restoring a database

Later on, when a preserved database is to be restored, the *reloader* reads from the archive repository the two archive files to restore 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. The destination RDB can then be queried or archived with A-SPARQL using SAQ. When the recreated database 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.

3. The archival benchmark ABench

The archival benchmark ABench consists of archival queries defining specified parts of a database to archive. The content to archive is specified either by listing RDF classes and properties to archive, or by SPARQL-like queries. The benchmark uses the Berlin benchmark dataset generator.

Table 1, Table 2 and Table 3 contain the archival queries of the benchmark ABench, together with the corresponding SPARQL CONSTRUCT queries generated by SAQ. The archival queries are denoted *Ai* and the generated SPARQL queries are denoted *Qi*.

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*, which retrieves the entire RDB tables *product* and *offer* for archival. The generated SPARQL query Q2 is an unbound-property UNION query of properties of the classes to archive.

Query A3 archives all values of explicitly specified properties. Here the generated SPARQL query Q3 is a bound-property UNION query of three BPTPs, where the properties are known URIs, one for each explicit property indicator 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 BPTPs.

Query A5 is similar to A2, but it constrains the properties of class *product* to archive. It corresponds to retrieving 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 URI matches 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 represented by URIs matching 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 with URIs that match the defined string. SQL queries selecting attributes from other tables would be unnecessary.

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. Thus the archived value is implicitly defined as a non-URI. 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.

4. Query processing in SAQ

The RD-view, the query processing steps in SAQ and the SAQ query rewrite optimizations are described in this section.

Table 1. ABench queries

Archival query	**Generated CONSTRUCT query**
Query **A1**: Archive the entire database	Query **Q1**:

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

Query **A2**: Archive the entire classes <*saq:product*> and <*saq:offer*>	Query **Q2**:

`ARCHIVE AS 'data2.nt', 'schema2.nt'`
`FROM <Products>`
`CLASSES saq:product, saq:offer ;`
	`CONSTRUCT { ?subject ?property ?value }`
	`FROM <Products>`
	`WHERE {{ ?subject ?property ?value .`
	`?subject rdf:type rdfs:Class .`
	`?subject rdf:type saq:product }`
	`UNION`
	`{ ?subject ?property ?value .`
	`saq:offer rdf:type rdfs:Class .`
	`?subject rdf:type saq:offer }}`

Query **A3**: Archive all values of the specific properties <*saq:product_label*>, <*saq:product_price*>, <*saq:offer_webpage*>	Query **Q3**:

`ARCHIVE AS 'data3.nt', 'schema3.nt'`
`FROM <Products>`
`PROPERTIES saq:product_label`
`saq:product_price, saq:offer_webpage;`
	`CONSTRUCT`
	`{ ?subject1 saq:product_label ?value1 .`
	`?subject2 saq:product_price ?value2 .`
	`?subject3 saq:offer_webpage ?value3 }`
	`FROM <Products>`
	`WHERE`
	`{{?subject1 saq:product_label ?value1 .}`
	`UNION`
	`{?subject2 saq:product_price ?value2 }`
	`UNION`
	`{?subject3 saq:offer webpage ?value3 } }`

Query **A4**: Archive *products* property *pNum1*for values > 214 and property *pNum3* for values < 348, and *reviews* property *text* for values matching the string *'time'* if the *reviews* have *rating4* > 8 .	Query **Q4**:

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

Table 2. ABench queries

Archival query	**Generated CONSTRUCT query**
Query **A5**: Archive the class <saq:*product*> except the RDF properties <saq:*product_label*> and <*saq:product_pNum1*>	Query **Q5**:

`ARCHIVE AS 'data5.nt', 'schema5.nt'`
`FROM <Products>`
`TRIPLES { ?subject ?property ?value }`
`WHERE`
`{?subject rdf:type saq:product .`
`saq:product rdf:type rdfs:Class .`
`FILTER (?property != saq:product_label ).`
`FILTER (?property != saq:product_pNum1 ) };`
	`CONSTRUCT { ?subject ?property ?value }`
	`FROM <Products>`
	`WHERE`
	`{ ?subject ?property ?value .`
	`?subject rdf:type saq:product .`
	`saq:product rdf:type rdfs:Class .`
	`FILTER (?property != saq:product_label ).`
	`FILTER (?property != saq:product_pNum1 ) }`

Query **A6:** Archive entire classes whose URIs match the string *'product'*	Query **Q6**:

`ARCHIVE AS 'data6.nt', 'schema6.nt'`
`FROM <Products>`
`TRIPLES {?subject ?property ?value}`
`WHERE {?class rdf:type rdfs:Class .`
`?subject rdf:type ?class .`
`FILTER REGEX (str(?class), 'product' ) };`
	`CONSTRUCT { ?subject ?property ?value }`
	`FROM <Products>`
	`WHERE`
	`{ ?subject ?property ?value .`
	`?class rdf:type rdfs:Class .`
	`?subject rdf:type ?class .`
	`FILTER REGEX (str(?class), 'product' ) }`

Query **A7**: Archive all subjects having a property with URI matching the string *'homepage'*.	Query **Q7**:

`ARCHIVE AS 'data7.nt', 'schema7.nt'`
`FROM <Products>`
`TRIPLES {?subject ?property ?value }`
`WHERE {?subject ?property1 ?value1 .`
`FILTER REGEX (str(?property1), 'homepage')}`
	`CONSTRUCT {?subject ?property ?value }`
	`FROM <Products>`
	`WHERE`
	`{ ?subject ?property ?value .`
	`?subject ?property1 ?value1 .`
	`FILTER REGEX (str(?property1), 'homepage')}`

Query **A8**: Archive all properties of the subjects from class *product* having *productFeature* 3 and 4, and *pNum1* > 348	Query **Q8**:

`ARCHIVE AS 'data8.nt', 'schema8.nt'`
`FROM <Products>`
`TRIPLES {?product ?property ?value }`
`WHERE {`
`?product rdf:type saq:product .`
`?product saq:product_label ?label .`
`?product rdf:type saq:product .`
`?product db:productFeature`
`saq:productFeature/_3 .`
`?product db:productFeature`
`saq:productFeature/_4 .`
`?product saq:product_pNum1 ?pn1 .`
`FILTER (?pn1 > 348) }`
	`CONSTRUCT { ?product ?property ?value }`
	`FROM <Products>`
	`WHERE {`
	`?product ?property ?value .`
	`?product rdf:type saq:product .`
	`?product saq:product_label ?label .`
	`?product db:productFeature`
	`saq:productFeature/_3 .`
	`?product db:productFeature`
	`saq:productFeature/_4 .`
	`?product saq:product_pNum1 ?pn1 .`
	`FILTER (?pn1 > 348) }`

Query **A9**: Archive all classes whose literal property values contain the specific string *'symbols'*	Query **Q9**:

`ARCHIVE AS 'data9.nt', 'schema9.nt'`
`FROM <Products>`
`TRIPLES {?subject ?property ?value }`
`WHERE { FILTER REGEX (?value, 'symbols') }`
	`CONSTRUCT {?subject ?property ?value }`
	`FROM <Products>`
	`WHERE {?subject ?property ?value .`
	`FILTER REGEX (?value, 'symbols')}`

Table 3. ABench queries

Archival query	**Generated CONSTRUCT query**
Query **A10**: Archive all subjects related by a property to another subject identified by the URI *<saq:product/_2549 >*	Query **Q10**:

`ARCHIVE AS 'data10.nt', 'schema10.nt'`
`FROM <Products>`
`TRIPLES {?subject ?property ?value }`
`WHERE {saq:product/_2549 ?relation ?subject}`
	`CONSTRUCT {?subject ?property ?value }`
	`FROM <Products>`
	`WHERE {?subject ?property ?value .`
	`saq:product/_2549 ?relation ?subject }`

4.1. The RD-view

First, we describe how the RD-view is defined in SAQ in a Datalog dialect [22]. A specialized RD-view for a given RDB is automatically generated by SAQ following the recommendations in [3] for mapping of relational databases to RDF. As in [3], 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 schema 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 relational database and the corresponding RD-view classes and properties. It is defined in terms of a number of *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.

There are six mapping tables in SAQ:

- The *class table* maps relational table names to RDFS class URIs.
- The *property table* maps relational column names to RDF property URIs.
- The *foreign key table* maps foreign keys to corresponding property URIs.
- The *many-to-many table* maps link tables to corresponding property URIs.
- The *type table* maps relational data types to corresponding XML Schema (XSD data types.

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 very large union of unions of sub-views representing relational schema concepts about tables, columns, types, primary keys and foreign keys. 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, the system generates a D-view for each specific relational database. Thus the definition of the D-view is different for different relational databases. 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 [14].

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 a relational database 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 literals the values of a column *A* in table *T*.
- One *foreign key view* FK_F is generated for each foreign key relationship *F*, representing foreign key values by URIs.
- One *many-to-many relationship view* MM_L is generated for each link table, representing values in link tables as URIs.
- One *row class view* RC_T is generated for each non-link table *T* to represent the classes of its row identifiers.

Thus a generated data view D-view in SAQ has the following structure:

`D-view(s,p,v) :-`	(1)

$\underset{T.A}{OR}\ C_{T.A}(s,p,v)$ $\underset{}{OR}$ $\underset{F}{OR}\ FK_F(s,p,v)$ $\underset{}{OR}$ $\underset{L}{OR}\ MM_L(s,p,v)$ $\underset{}{OR}$ $\underset{T}{OR}\ RC_T(s,p,v)$	

where $\underset{T.A}{OR}$ denotes a disjunction over all attributes *T.A* in all tables *T* in the database, $\underset{F}{OR}$ denotes a disjunction over all foreign key relationships *F* in the database, $\underset{L}{OR}$ denotes a disjunction over all link tables *L* in the database, and $\underset{T}{OR}$ denotes a disjunction over all tables *T* in the database.

A D-view generated by SAQ for the Berlin SQL data set contains the following sub-views:

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

4.2. Query processing steps in SAQ

The D-view is usually very large, containing many disjunctions. Naive processing of such a view in the RDB is slow. SAQ therefore applies some special rewrite techniques in a number of steps for transforming archival queries to the RD-view into efficient SQL statements sent to the RDB.

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 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 the transformation called *view specialization* [14]. 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.

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

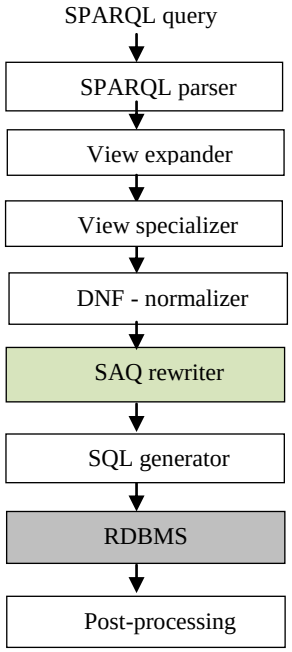


Fig. 4. SAQ Query processing.

i. The disjunction for an expanded BPTP with the structure *(?s P_i ?v)* is reduced into a single *property conjunction* for P_i representing the single sub-view in the D-view having the property $p = Pi$. Here we assume that all the RDF properties P_i mapping relational attributes have unique URIs.

ii. 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.

iii. The disjunction for an expanded 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 mapped by a URI-construct predicate to rows in *T*.

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

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.

After normalization a UNION of 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* [20] 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. All 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.

4.3. 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 select sets of properties to archive they are mostly unbound-property queries, so the query transformation optimizations for unbound-property queries are presented here. The processing and optimization of bound-property queries to an RD-view uses the techniques described in [13][14] and is outside the scope of this paper.

All transformations assume 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.

4.3.1. 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

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

Rewrite ***Query***	**GCT**	**Eliminate S-view**	**is-literal**	**type-match**	**FKR**
Q1	X				
Q2	X	X			
Q3	X				
Q4					
Q5	X	X			
Q6	X	X			
Q7	X	X			
Q8	X	X			
Q9			X	X	
Q10	X	X			X

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 attributes from tables. The GCT is not applicable on queries Q4 and Q9, since they retrieve a single attribute per table.

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:

i. 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.
ii. Allocate a hash table that for each extracted conjunction maintains mappings to the disjuncts from which its terms have been extracted.
iii. 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.
iv. 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 conjunctions.

The pseudo code of GCT can be found in Appendix 2.

4.3.2. 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. Thus SQL code to access foreign keys and links is not generated and the number of generated SQL queries is reduced.

4.3.3. 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.). Thus the rule is applied for 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.

4.3.4. Foreign key relationship (FKR) reduction

The *FKR* rule reduces SPARQL unbound-property queries where a subject-object join variable is shared between two UPTPs. SQL queries are restricted to tables where there is a foreign key relationship between the tables referenced by the joined UPTPs.

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.

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*.

4.3.5. Eliminate S-view reduction

The *eliminate S-view* rule reduces unbound-property queries so that an S-view subject is never joined with the subject of a 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 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 UBTP *(?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*.

5. 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 [18] and D2RQ [8], 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 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 triples in the SAQ RD-view and the number of the query result triples for Q1-Q10.

Table 5. RDB sizes and number of result triples for Q1-Q10 when using SAQ

	RDB1	**RDB2**	**RDB3**
Phys. size	184 MB	1.8 GB	9 GB
Triples	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

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 the RDF view were run from a Java program, implementing a Jena Provider [25], 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 [7]. 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 [7]. The Java heap size was set to 4 GB.

g) The default mappings of the analyzed 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 back-end 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.

5.1. 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-naïve. Table 6 also shows the number of SQL queries sent to the RDB for the different approaches.

5.1.1. Impact of GCT

The performance of GCT (***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.

5.1.2. Impact of eliminate S-view

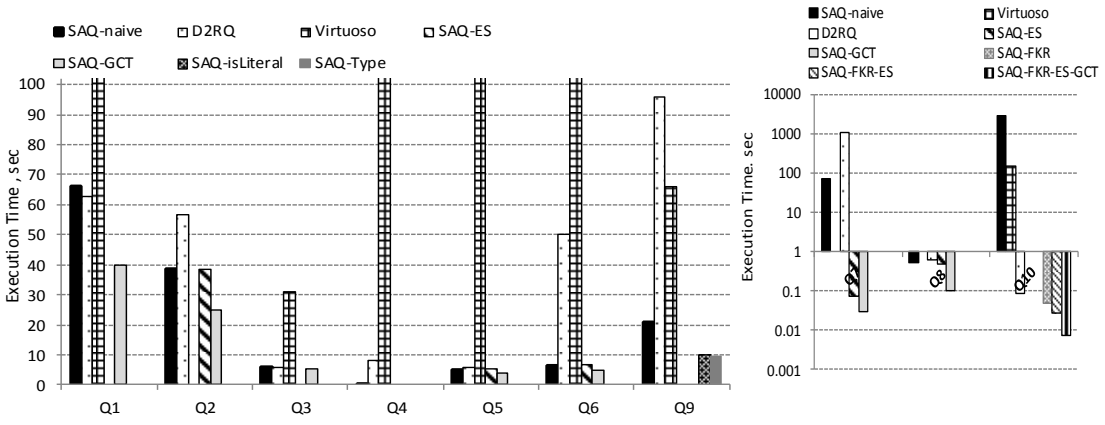
The *eliminate S-view* reduction (***SAQ-ES)*** slightly improves (by 1-3%) the performance for unbound-property queries with one UPTP, i.e. Q2, Q5, Q6 and Q8, by reducing the number SQL queries. In contrast, *eliminate S-view* significantly improves the performance for unbound-property queries with 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%.

5.1.3. 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.

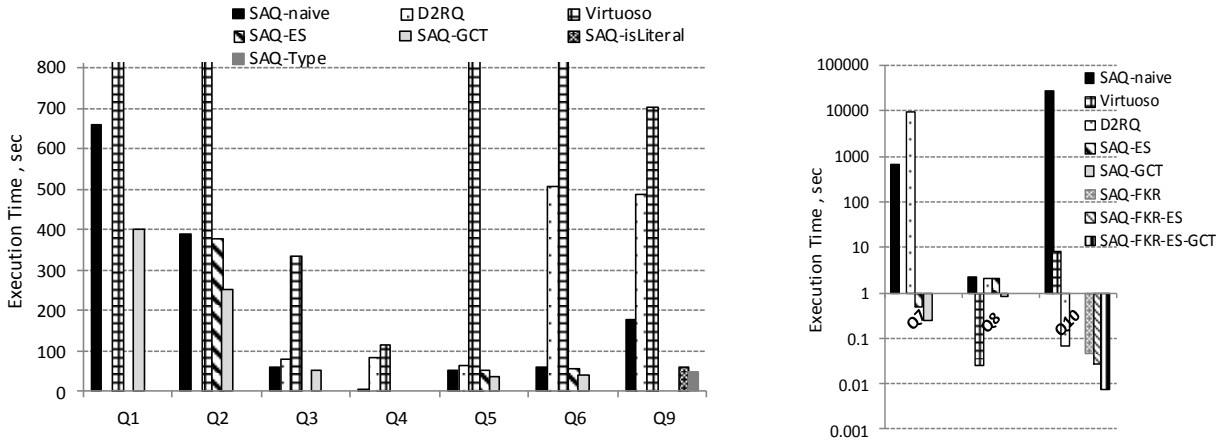
The *type-match* reduction (***SAQ-Type***) further improves the performance for Q9 by 200-300%. The improvement is because without *type-match*, 40 unnecessary SQL queries selecting relational columns of type different than VARCHAR are generated and sent to the RDB.

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



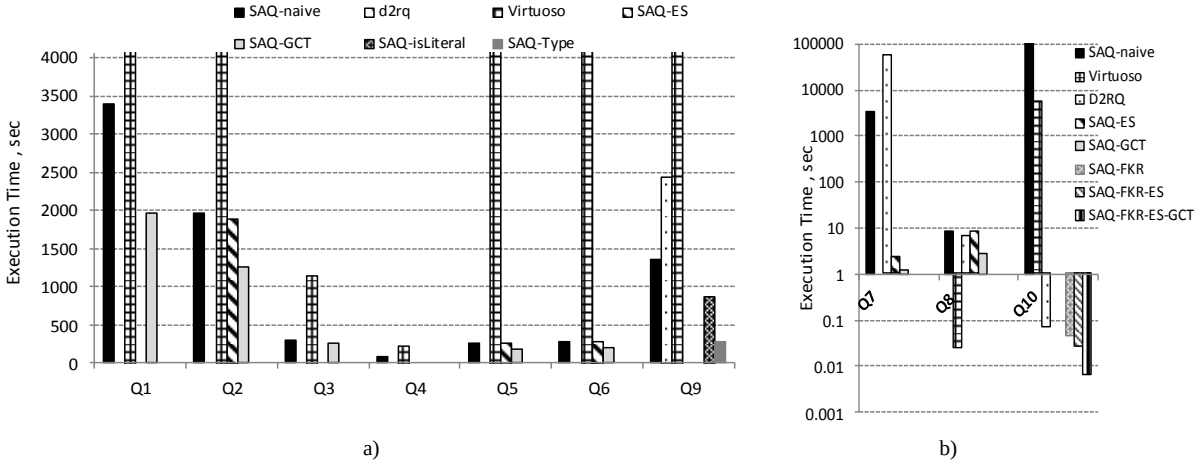
a) b)

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



a) b)

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



a) b)

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

5.1.1. 5.1.4. 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.

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

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

Rewrite Query	**SAQ-naive**	**GCT**	**ES**	**is-literal**	**type-match**	**FKR**
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)

5.2. Query performance of other systems

To analyze 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.

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-naive***. However, ***SAQ-GCT*** still outperforms ***D2RQ*** for Q1 because 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 [14] 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 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***. In addition, GCT improves the performance of ***SAQ-FKR-ES-GCT***.

5.2.2. 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-naive*** since it tries to materialize in main memory the large result set, while SAQ streams the results.

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-naive*** and additionally one or many parameterized SQL queries. 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***. Then GCT is not used. It scales much worse than ***SAQ-naïve*** since it 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-naive*** 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.

6. Related work

Virtuoso RDF Views [18][19], D2RQ [8][9], Ultra-wrap [15][16], and SquirrelRDF [1][24] 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 perfor-

mance for SPARQL unbound-property queries selecting RDB contents to archive.

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

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 analyze 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 [16] 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, in which streamed post-processing combines the SQL result scans. 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 [15] [16] and it can be concluded from these that Ultrawrap has no special optimizations. It is shown in these papers that an Ultrawrap 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.

Work on optimizing disjunctive database queries in general is described in [1] [12][17]. The closest work to *GCT* is the combinatorial algorithm [17], which merges disjuncts with common subexpressions 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][12] 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.

7. Conclusions and future work

In this paper we present an approach for scalable long-term archival of RDBs through SPARQL queries, implemented in the SAQ system. SAQ automatically generates an RDF view of an RDB called the *RD-view*. The RD-view can be queried with SPARQL queries that are translated into SQL queries sent to the RDB. To flexibly select parts of an RDB to archive, a SAQ user defines an archival query in an extension of SPARQL having archival statements, A-SPARQL. In A-SPARQL the parts of the RDB to archive are specified either as a list of RDF classes, properties in the RD-view, or as a SPARQL-like archive query to the RD-view. 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. These query rewriting strategies were evaluated in a new benchmark for archival queries called *ABench*. ABench consists of a set of archival queries that archive selected parts of databases generated by the Berlin benchmark data generator.

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. Furthermore, they allow RDB indices to be utilized for better scalability. 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. Thus the SAQ optimizations are useful not only for archival queries, but also for unbound-property and UNION queries in general.

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 [21].

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] A. Sören, L. Feigenbaum, D. Miranker, A. Fogarolli and J. Sequeda, Use Cases and Requirements for Mapping Relational Databases to RDF, W3C Working Draft 8, http://www.w3.org/TR/rdb2rdf-ucr/ (2010)

[4] C. Bizer and A. Schultz, Berlin SPARQL Benchmark (BSBM) Specification–V3.1.,http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/index.html (2011).

[5] C. Bizer and A. Schultz, Berlin SPARQL Benchmark (BSBM) Specification, Data Generator and Test Driver, http://www4.wiwiss.fuberlin.de/bizer/BerlinSPARQLBenchmark/spec/20080912/index.html#datagenerator

[6] 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)

[7] 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.wiwiss.fu-berlin.de/bizer/d2rq/spec/ (2009)

[8] C. Bizer, and A. Seaborne, D2RQ-Treating Non-RDF Databases as Virtual RDF Graphs, Poster at 3rd International Semantic Web Conference (2004)

[9] C. Bizer and R. Cyganiak, D2R Server-Publishing Relational Databases on the Semantic Web, Poster at the 5th International Semantic Web Conference (2006)

[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. 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)

[13] 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)

[14] 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)

[15] 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)

[16] 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

[17] 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)

[18] O. Erling, Declaring RDF views of SQL Data, W3C Workshop on RDF Access to Relational Databases, 25-26 October, Cambridge, MA, USA (2007)

[19] 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)

[20] 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.

[21] St. Harris St and A. Seaborne, SPARQL 1.1 Query Language W3C Working Draft 24 July 2012

[22] 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)

[23] N-triples, W3C RDF Core WG Internal Working Draft, http://www.w3.org/2001/sw/RDFCore/ntriples/

[24] SqirrelRDF, http://jena.sourceforge.net/SquirrelRDF/

[25] Virtuoso Jena Provider, OpenLink Virtuoso Universal Server: Documentation, http://docs.openlinksw.com/virtuoso/rdfnativestorageproviders.html#Rdfnativestorageprovidersjena (2009)

[26] Virtuoso Universal Server, http://virtuoso.openlinksw.com/

Appendix 1: Rules for translation archival queries in A-SPARQL into SPARQL queries

An archival query in A-SPARQL is straightforward to translate into a CONSTRUCT SPARQL query. It is very often a CONSTRUCT-UNION query, since it unions sets of triples to archive. The translation rules are:

1. The following translation is made when the content to archive is specified as explicit lists of classes and/or properties is in A2 and A3:
    a. The CONSTRUCT clause of the generated SPARQL query is constructed by one TP for all specified classes, and one TP for each specified property where:

i. The TP for classes where all properties are archived is the UPTP as in Q2:

```
?subject ?property ?value.
```

ii. The TP for a property with URI *P* to be archived is always a BPTP as in Q3:

```
?subject P ?value.
```

b. The WHERE clause of the generated SPARQL query consists of one UNION section per specified class or property, where:

i. A UNION section is generated for each archived class with URI *S* having the following TPs as in Q2:

```
{?subject ?property ?value.
?subject rdf:type rdfs:Class.
?subject rdf:type S}
```

ii. A UNION section is generated for each archived property *P* as in Q3:

```
{?subject P ?value}
```

2. When the contents to archive are specified by TRIPLES-WHERE clauses as in A4-A10, the following translation rules are used:

a. The CONSTRUCT clause of the generated SPARQL query contains the TPs from all TRIPLES clauses in the A-SPARQL query.

b. The WHERE clause of the generated SPARQL query consists of one UNION section for each TRIPLES-WHERE clause, where

i. Each UNION contains the set of TPs and SPARQL functions specified in the corresponding TRIPLES clause together with the TPs defined in the corresponding WHERE clause.

ii. If there is no WHERE clause in the A-SPARQL query as in A1, both the CONSTRUCT and WHERE clauses in the translated query contain only the TPs specified in the TRIPLES clause.

Appendix 2. Pseudo code of the GCT algorithm

```
Function GCT(P, gf) -> GP
Input:     P - a DNF predicate with normal-
ized 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:=null
3. for each disjunct D in P do
4.   if D is an atom then GP:= orify(GP,D)
5.   else if D is not a conjunction then
                          GP:=orify(D,GP)
6.   else if D has only one term then
                          GP:=orify(D,GP)
7.   else CT:=gf(D) /*CT is a list of common
terms)*/
8.        if CT=null then GP:=orify(D,GP)
9.        else put in Ht(key=CT):=
               orify((D with CT removed),
               (existing value for CT in Ht
))
10. for each (CT' and valueCT') in Ht do
11.   GP:=orify(andify(CT', valueCT'),GP)
14. return GP
```

The function *orify(x,y)* forms a disjunction between predicates *x* and *y*, and *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.