Optimizing Unbound-property Queries to RDF Views of Relational Databases

Silvia Stefanova, Tore Risch

Uppsala University, Department of Informational Technology
Box 337, SE-751 05 Uppsala, Sweden
{Silvia.Stefanova, Tore.Risch}@it.uu.se

Abstract. SAQ (Semantic Archive and Query) is a system for querying and long-term preservation of relational data in terms of RDF. In SAQ relational data in a back-end DBMS is exposed as an RDF view, called the *RD-view*. SAQ can process arbitrary SPARQL queries to the RD-view. In addition long-term preservation as RDF of selected parts of a relational database is specified by SPARQL queries to the RD-view. Such queries usually select sets of RDF properties and thus in the query definition a property p is unknown. We call such queries *unbound-property queries*. This class of queries is also present in the SPARQL benchmarks. We optimize unbound-property queries by introducing a query transformation algorithm called *Group Common Terms, GCT*. It pulls out from a DNF normalized query those common terms that can be translated to SQL predicates accessing the relational database. Our experiments using the Berlin SPARQL benchmark show that GCT improves substantially the query execution time to a back-end commercial relational DBMS for both selective and unselective unbound-property queries. We compared the performance of our approach with the performance of other systems processing SPARQL queries over views of relational databases and showed that GCT improves scalability compared to the approaches used by the other systems.

Keywords: SPARQL queries, RDF views of relational databases, query optimization, query rewrites, unbound property queries

1 Introduction

Semantic Web technology and, in particular, RDF and RDFS seem promising for both search and long-term preservation of any kind of data including data currently stored in relational databases. In order to investigate the use of RDF for both search and archival of existing relational databases we have developed the SAQ (Semantic Archive and Query) system. In SAQ relational data in a back-end DBMS is exposed as RDF by a view, called the *RD-view,* represented in a Datalog dialect. The RD-view is automatically generated by accessing the database schema. SAQ can process arbitrary SPARQL queries to the RD-view. Long-term preservation as RDF of the contents of selected parts of a relational database is specified by SPARQL queries to the RD-view.

In the RD-view tables are represented as RDFS classes and attributes as RDF properties. Each data value in the relational database is viewed as a triple *(s, p, v)*, where the subject *s* is a URI identifying a row in a relational table, *p* is an RDF property representing the column (i.e. attribute) where a value is stored, and *v* is the data value of the attribute for the row. In SAQ a SPARQL query to the RD-view is transformed into an execution plan containing SQL calls to the back-end relational DBMS followed by post-processing.

Queries to archive database contents typically select sets of attributes of tables to archive. This corresponds to selecting sets of RDF properties in the RD-view to be archived, for example all properties of the class representing the table *offer*. Therefore, in such SPARQL queries a property *p* in some triple pattern is not known. We call such queries *unbound-property queries*. Moreover, unbound-property queries are also present in SPARQL benchmarks. For example, in the SP2Bench benchmark [18] queries *Q9* and *Q10* are unbound-property queries, and in the Berlin SPARQL benchmark [2] *Query 9* and *Query 11* are unbound-property queries, while *Query 9* is a DESCRIBE query that can be expressed as an unbound-property query. Since *p* is unknown in unbound-property queries, the translation from SPARQL to SQL is not trivial and can ”easily result in large unions” of sub-queries [7] and therefore “using of variables in the predicate position is discouraged” [8].

In this paper we present a novel approach for optimizing unbound-property queries by implementing a predicate rewrite rule called *group common terms (GCT)*. GCT is shown to substantially improve SPARQL query execution time. It partially denormalizes disjunctive normal form (DNF) predicates to form query fragments doing select-project-joins over the back-end relational tables. The reason for the performance improvement is that GCT generates execution plans that access data in row-order, which is substantially more efficient to process than without GCT, where data is accessed column-wise. In the performance section it will be shown that our approach improves query performance compared to a naïve approach without using GCT. Furthermore, we show that SAQ with GCT executes unbound-property queries substantially faster than other systems able to process SPARQL queries to views of relational databases [4][8]. By investigating the SQL queries emitted by the other systems, we show that they do not employ query transformations similar to GCT.

The rest of this paper is organized as follows. First, in Section two unbound-property queries are defined and exemplified. In Section three the architecture of the SAQ system is presented, the RD-view is defined, and the steps of the query processing in SAQ are explained. In Section four the GCT algorithm is presented. Section five analyzes the performance of SAQ for unbound-property queries and compares it with related systems. Section six describes related work and finally Section seven summarizes.

2 Unbound-property Queries

A *bound-property triple pattern* is a SPARQL triple pattern *(s, P, v)* where the property *P* is a URI representing an RDF property, e.g. *(?s1*

saq:product#review/person ?s.). For SPARQL queries to an RD-view the property must match a URI representing a relational column, otherwise the result is empty.

An *unbound-property triple pattern* is a triple pattern *(s, u, v)* where *u* is a variable, e.g. *(?s ?p ?v)* or *(%offerXYZ% ?property ?hasValue)*.

A *bound-property query* is a SPARQL query having only bound-property triple patterns. An *unbound-property query* is a SPARQL query having one or several unbound-property triple patterns. Finally, a *simple unbound-property query* is an unbound-property query with a single unbound-property pattern.

Fig. 1 shows a small relational database *product* having three tables *offer*, *person,* and *review*. The tables are parts of the relational database generated by the Berlin SPARQL benchmark data generator [1]. In the example, they are populated with two offers, two persons, and a review made by each of them. In the scalability experiments later, we use the full Berlin benchmark relational database with the tables *product*, *offer*, *person*, *producer*, *productfeature*, *productfeatureproduct*, *producttype*, *producttypeproduct*, *review*, and *vendor*.

The columns *onr*, *nr,* and *rnr* are the primary keys in the tables, while the column *person* in table *review* references the column *nr* in table *person* as foreign key.

Table *offer*

onr	**price**	**deliveryDays**
5	854.18	3
7	440.9	5

Table *review*

rnr	**person**	**reviewDate**
10	1	2007-09-16
166	8	2006-05-12

Table *person*

nr	**name**	**country**	**publisher**
1	Caryn	KR	9
8	Linda-Nada	AT	3

Fig. 1. *product* database

The following queries are examples of SPARQL queries to the RD-view for preservation and search of the *product* database. They represent different kinds of unbound-property queries with varying selectivity.

Query Q1

```
SELECT ?s ?p ?v
FROM <product>
WHERE {?s rdf:type %offerType%.
       ?s ?p        ?v        }
```

(1)

Query Q2

```
SELECT ?property ?hasValue ?isvalueOf
FROM <product>
WHERE {
        {%offerXYZ% ?property ?hasValue }
        UNION
        {?isValueOf ?property %offerXYZ%} }
```

(2)

Query Q3	(3)

`SELECT ?s ?p ?v`
`FROM <product>`
`WHERE{ ?s saq:product#person/name %nameXYZ%.`
`?s ?p ?v }`	

Query Q4	(4)

`SELECT ?s ?p ?v`
`FROM <product>`
`WHERE { ?s1 saq:product#review/person ?s .`
`?s saq:product#person/country 'JP'.`
`?s ?p ?v }`	

In the queries <*product*> denotes the URI for the RD-view of the database product.

Q1, Q3, and *Q4* are simple unbound-property queries, while *Q2* is a union of simple unbound-property queries.

Query *Q1* converts the entire table *offer* to RDF triples. *%offerType%* is the URI associated with table *offer*. *Q1* is a very unselective unbound-property query.

Q2 is a highly selective query that retrieves all information about an offer *%offerXYZ%*, i.e. a single row from the table *offer*. *Q2* is called 'Query 11' in the Berlin SPARQL Benchmark [1][2]. *Q2* searches for both the properties and the inverse properties of an explicitly given offer.

Q3 retrieves all the properties of a person for a given name *%nameXYZ%*. Like *Q2* query *Q3* is highly selective, it selects one row from the table *person*. The difference from *Q2* is that the subject of the unbound-property is not explicitly specified, so the row in the relational database cannot be identified by the URI as in *Q2*.

Query *Q4* retrieves all properties of the 10% of all reviewers coming from country 'JP'. Unlike *Q3* the name of a described person is not explicitly specified, but are retrieved by joining the tables *person* and *review*. It is an unbound-property query with a join. *Q4* is more selective than *Q1* but much less selective than *Q2* and *Q3* since it retrieves more rows when the database grows.

It is shown in section five that the performance of the unbound-property queries *Q1…Q4* is substantially improved by applying the proposed GCT rule. Queries like *Q2* defined with a 'UNION' clause are handled as a union of several non-disjunctive queries, where GCT is applied on each sub-query in the union. The unusual case of unbound-property queries with several unbound-property patterns in a conjunction are also handled by SAQ, but are outside the scope of this paper. The processing of bound-property queries to an RD-view is also outside the scope and was described in [15][16].

3 SAQ

The architecture of the SAQ system is presented in Fig. 2. The *source DB* is the underlying relational database, which can be queried and preserved by SAQ. The *RD-view generator* generates the *RD-view* over the source DB from the database schema

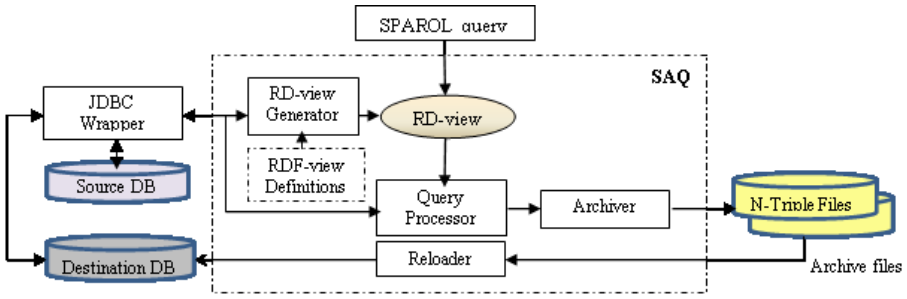


Fig. 2. SAQ Architecture

by using the *RD-view definitions*. The *query processor* executes arbitrary SPARQL queries to the RD-view by accessing the source DB through the *JDBC wrapper*.

When the source DB or part of it is to be preserved as RDF a SPARQL query extracting the desired content is sent to the query processor. The query result is transformed by the *archiver* into RDF triples and stored in a repository of *archive files* as N-Triples [14]. The archiver stores two N-triple files – one with the transformed query result, called the *data archive* and another one with schema information, called the *schema archive*. For instance, when all instances of class *%offerType%* representing the content of the relational table *offer* are unloaded as RDF, *Q1* is executed by the query processor in SAQ.

Later on, when a preserved database is to be restored, the *reloader* reads the archive files from the repository and makes it live again by reloading it into a *destination DB*. It first reads the schema archive to generate the relational database schema and then loads the data by reading the data archive.

3.1 RD-view Definition

In the RD-view each relational table is represented as an RDFS class, while each column is represented as an RDF property, as prescribed in [22]. For example, the table *person* is represented by the RDFS class URI *saq:product#person.* The URIs *saq:product#person/name* and *saq:product#person/country* are the RDF properties associated with the columns *name* and *country* of table *person*.

SAQ contains four meta-tables *cMap*, *pMap, fkMap,* and *rmmMap* providing mappings between RDF resources and the relational schema. The *class mapping table, cMap(T, ClassID),* maps 1-1 each relational table named *T* to the corresponding RDFS class *ClassID*. Here *T* is primary key and *ClassID* is secondary key. The *property mapping table, pMap(T, A_j, PropID),* maps each column (attribute) named A_j in table *T* into an RDF property *PropID*. The composite primary key is $T + A_j$, and *PropID* is secondary key. The foreign key mapping table, *fkMap(T, A_i, T', ResID)* maps a foreign key attribute A_i in a table *T* referencing table *T'* into an RDF resource *ResID*. The composite primary key of *fkMap* is $T + A_i + T'$, and *ResID* is secondary key. Finally, the many-to-many relationship mapping table, *rmmMap(T, A_m, A_n, T', T'', RmmID)* maps the attribute pair (A_m, A_n) of the composite primary key in *T*,

where A_m and A_n are foreign key attributes referencing tables *T'* and *T''* into an RDF resource *RmmID*.

SAQ populates the meta-tables by accessing the catalogue of the relational database to construct identities of *ClassID, PropID, ResID,* and *RmmID*. Furthermore, the user can update the mapping tables to override default mappings in order to match some ontology or to limit data access.

In the following Datalog definitions we use capital letters to denote constants and small letters to denote variables.

The RD-view, defined in Datalog is a union of the following sub-views: the *relational column views* $C_{T,A}$, *the foreign key views*, $FK_{T,A}$, the *many-to many relationship views*, MM_T, and the *row class views,* RC_T. For instance, the RD-view for the *product* database, *RD-view*$_{product}$ is a union of:

- the relational column views for attributes *price* and *deliveryDays* of table *offer*
- the relational column views for attributes *name*, *country,* and *publisher* of table *person*
- the relational column view for attribute *reviewDate* of table *review*
- the foreign key view for attribute *person* of table *review*
- the row class views for tables *offer*, *person,* and *review*

The RD-view for the *product* database is

$$
\begin{aligned}
&\text{RD-view}_{\text{product}}(s,p,v) \text{ :-}\\
&C_{\text{offer.price}}(s,p,v) \quad \text{OR}\\
&C_{\text{offer.deliveryDays}}(s,p,v) \quad \text{OR}\\
&C_{\text{person.name}}(s,p,v) \quad \text{OR}\\
&C_{\text{person.country}}(s,p,v) \quad \text{OR}\\
&C_{\text{person.publisher}}(s,p,v) \quad \text{OR}\\
&C_{\text{review.reviewDate}}(s,p,v) \quad \text{OR}\\
&FK_{\text{review.person}}(s,p,v) \quad \text{OR}\\
&RC_{\text{offer}}(s,p,v) \quad \text{OR}\\
&RC_{\text{person}}(s,p,v) \quad \text{OR}\\
&RC_{\text{review}}(s,p,v)
\end{aligned}
\tag{5}
$$

In general, an RD-view in SAQ has the following structure:

$$
\begin{aligned}
&\text{RD-view}(s,p,v) \text{ :-}\\
&\underset{T}{\text{OR}}\ (\underset{A}{\text{OR}}\ (C_{T.A}\ (s,p,v))) \quad \text{OR}\\
&\underset{T}{\text{OR}}\ (\underset{A}{\text{OR}}\ (FK_{T.A}(s,p,v))) \quad \text{OR}\\
&\underset{T}{\text{OR}}\ (MM_T(s,p,v)) \quad \text{OR}\\
&\underset{T}{\text{OR}}\ (RC_T(s,p,v))
\end{aligned}
\tag{6}
$$

where $\underset{B}{OR}\ (P)$ denotes a disjunction over all *P* for each possible value of *B*.

In general, a relational column view $C_{T,A}$ (7a) is defined for each attribute (column) A that is neither primary key nor a foreign key attribute of each table *T*.

```
C_T.A (s,p,v) :-                                  |  C_review.reviewDate(s,p,v):
R_T(a_1,..,a_i,..,a_r)                      AND   |  R_review(rnr,person,reviewDate)   AND
cMap(T,cid)                                 AND   |  cMap('review',cid)                AND
pMap(T,A_i,p)                               AND   |  pMap('review,'reviewDate',p)      AND
rowid(cid,(a_1,..,a_k),s)                   AND   |  rowid(cid,(rnr),s)                AND
v= a_i                                            |  v = reviewDate
```

a) b) (7)

In (7a) R_T is the *source predicate* representing the relational database table *T*, and A_j is the name of the attribute *A* in *T*. (a_1, ...,a_j, ..., a_r) is a tuple representing a row in *T*. The primary key of *T* is represented by the tuple (a_1,...,a_k). The variable *cid* is bound to the RDFS class associated with table *T*. The *rowid* predicate maps row identifiers to relational rows. It creates a unique URI *s* representing a row identifier in *T* by concatenating the class associated with a table and the primary key of a row. The predicate *rowid* is implemented in SAQ as a multidirectional foreign predicate [12], which implements both i) the construction of a new row identifier as described above and ii) its inverse to access the primary key based on a known row identifier. The URI *p* represents a relational attribute as an RDF property. The *attribute variable* a_j (and its alias *v*) holds the value of attribute A_j in a row.

Example (7b) shows the relational column view $C_{review,reviewDate}$ that represents attribute *reviewDate* in table *review*. R_{review} is the source predicate of table *review* and *cid* is bound to its associated class.

A *foreign key view* $FK_{T,A}$ for non-composite foreign key *A* in table *T* has the structure in (8a). $FK_{T,A}$ is defined in terms of the class URI *cid* associated with *T*, the class URI *cid'* associated with the table *T'* owning the foreign key, and the name of the foreign-key attribute, A_i. For example, (8b) shows the foreign key view $FK_{review,person}$. SAQ supports composite key foreign key views as well, which is not elaborated here.

```
FK_T.A(s,p,v) :-                                  |  FK_review.person(s,p,v) :-
R_T(a_1,..,a_i,..,a_r)                      AND   |  R_review(rnr,person,reviewDate)  AND
cMap(T,cid)                                 AND   |  cMap('review',cid)                AND
rowid(cid,(a_1,..,a_k),s)                   AND   |  rowid(cid,(rnr),s)                AND
fkMap(T,A_i,T',p)                           AND   |  fkMap('review','person',          AND
                                                  |                 'person',p)
cMap(T',cid')                               AND   |  cMap('person',cid')               AND
rowid(cid',(a_i),v)                               |  rowid(cid',(nr),v)
```

a) b) (8)

A *many-to-many relationship view* $MM_{T,A,B}$ defined a many-to-many relationship between two other tables *T'* and *T''* represented by a connection table *T* of foreign keys (*A*, *B*) [22]. This is not further elaborated here.

Finally, a *row class view* RC_T (9a) represents the RDF classes of the row identifiers in a relational table *T*. For example, (9b) shows the row class view for table *review,* RC_{review}.

```
RC_T(s,p,v) :-                          |  RC_review(s,p,v) :-
R_T(a_1,...,a_r)                  AND   |  R_review(rnr,person,reviewDate) AND
cMap(T,cid)                       AND   |  cMap('review',cid)               AND
rowid(cid,(a_1,..,a_k),s)         AND   |  rowid(cid,(rnr),s)               AND
p = rdf:type                      AND   |  p = rdf:type                     AND
v = cid                                 |  v = cid
            a)                          |              b)
```
(9)

3.2 Query Processing

The main steps of the query processing in SAQ are illustrated by Fig. 3. The *SPARQL parser* transforms the SPARQL query into a Datalog expression where each SPARQL triple pattern becomes a reference to the RD-view. The *RD-view expander* recursively expands each RD-view reference in the query into a disjunctive expanded RD-view. It thereby looks up the mapping tables *cMap, pMap*, *fkMap,* and *rmmMap* to replace the variables in the expanded RD-view with corresponding URIs. Then it simplifies the query by unifying terms [9]. Each bound-property triple pattern is thereby simplified into a single conjunction [15] since the property URI determines the accessed table's attribute. However, each unbound-property triple pattern will remain a disjunction. The *DNF-normalizer* transforms the simplified query to a disjunctive normal form (DNF) predicate. The *GCT transformer* applies the GCT rewrite rule to transform the DNF predicate into a more efficient representation. It groups those common terms in different disjuncts of the DNF predicate that can be translated to SQL. The *SQL generator* generates calls to SQL for each grouped query fragment. The *post-processor* transforms the result from the SQL queries sent to the relational DBMS, e.g. it constructs URI objects and forms SPARQL result tuples. All processing in the system is streamed so that no large intermediate collections are generated.

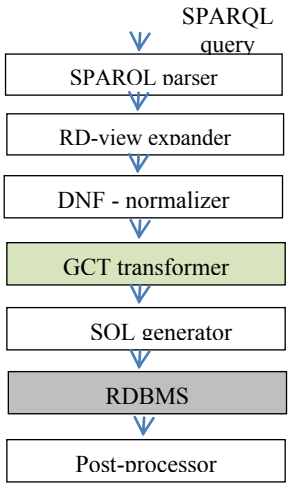


Fig. 3. Query processing in SAQ

4 The GCT Rule

The GCT rule is applied on a DNF predicate. It extracts from the disjuncts common terms that can be translated to SQL queries, i.e. source predicates R_T and SQL comparisons predicates. After GCT the DNF predicate becomes a disjunction of conjunctions between grouped terms and disjunctions of the remaining terms with the grouped terms removed. The remaining terms cannot be expressed in SQL and must be post-processed. For example, (10) shows the view expanded, simplified, and DNF

normalized unbound-property query *Q4* where the tables *cMap*, *pMap,* and *fkMap* have been looked up by the RDF-view expander to obtain for each table *T* its associated RDFS class C_T, and the RDF properties $P_{T,Aj}$=*saq:product#T/*A_j representing the attributes A_j in *T*.

```
Q4(s,p,v) :-
(R_person(nr,v,'JP',publisher)            AND
 R_review(rnr,nr,reviewDate)              AND
 rowid(C_person,(nr),s)                   AND
 rowid(C_review,(rnr),s1)                 AND
 p = saq:product#person/name)                   OR
(R_person(nr,name,'JP',publisher)         AND
 R_review(rnr,nr,reviewDate)              AND
 rowid(C_person,(nr),s)                   AND
 rowid(C_review,(rnr),s1)                 AND
 p = saq:product#person/country)                OR
 v = 'JP' )                                                    (10)
(R_person(nr,name,'JP',v)                 AND
 R_review(rnr,nr,reviewDate)              AND
 rowid(C_person,(nr),s)                   AND
 rowid(C_review,(rnr),s1)                 AND
 p = saq:product#person/publisher)              OR
(R_person(nr,name,'JP',publisher)         AND
 R_review(rnr,nr,reviewDate)              AND
 rowid(C_person,(nr),s)                   AND
 rowid(C_review,(rnr),s1)                 AND
 p = rdf:type                             AND
 v = C_person )
```

We notice that bold marked terms, representing source predicates, can be pulled out and later translated to a single SQL join query. In the example GCT will produce the predicate:

```
Q4(s, p, v) :-
(R_person(nr,name,'JP',publisher)         AND
 R_review(rnr,nr,reviewDate))             AND
  (rowid(C_person,(nr),s)                 AND
   rowid(C_review,(rnr),s1)               AND
   p = saq:product#person/name            AND
   v = name)                                    OR
  (rowid(C_person,(nr),s)                 AND
   rowid(C_review,(rnr),s1)               AND
   p = saq:product#person/country         AND                  (11)
   v = 'JP')                                    OR
  (rowid(C_person,(nr),s)                 AND
   rowid(C_review,(rnr),s1)               AND
   p = saq:product#person/publisher       AND
   v = publisher)                               OR
  (rowid(C_person,(nr),s)                 AND
   rowid(C_review,(rnr),s1)               AND
   p = rdf:type                           AND
   v = C_person)
```

In (11) the bold marked accesses to the source predicates are broken out from the disjunction and referenced only once, while in (10) the source predicates are referenced in each disjunct.

Without GCT the SQL generator will construct several SQL queries, one for each conjunction that can be translated to SQL. For example, in (10) the four bold marked conjunctions would produce these four SQL queries:

```
1) SELECT name from person p, review r WHERE p.country='JP'
   AND p.nr=r.person
2) SELECT country from person p, review r WHERE
   p.country='JP' AND p.nr=r.person
3) SELECT publisher from person p, review r WHERE
   p.country='JP' AND p.nr=r.person
4) SELECT nr from person p, review r WHERE p.country='JP'
   AND p.nr=r.person
```
(12)

The four queries would be sent to the relational DBMS and their result tuples postprocessed in sequence. Binding of *s*, *p* and *v* has to be performed by the not bold marked post-processing predicates in (10) after the rows are retrieved since object construction and result variable bindings cannot be expressed in SQL.

With GCT applied in (11) a single SQL query is produced from the grouped terms:

```
SELECT nr, name, country, publisher from person p,
review r WHERE p.country='JP' AND p.nr=r.person
```
(13)

In (11) the values from the relational tables are retrieved in row-order, while in (10) they are retrieved column-by-column. Therefore the execution plan generated from (11) is more efficient than the execution plan from (10).

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

(i) In a pre-step, normalize the variable names of the DNF predicate so that the same variable names are used in each disjunct.

(ii) Allocate a hash table that for each predicate group maintains mappings to the disjuncts from which its predicates have been extracted.

(iii) For each disjunct, extract the source predicates and SQL comparison predicates to form a conjunctive predicate group to extract. Use the entire extracted conjunction as key in the hash table.

(v) After the entire DNF predicate is scanned, go through the hash table and form for each extracted conjunction *c* a conjunction between *c* and the remaining terms in the disjuncts where *c* occurs. Finally, form a disjunction of all constructed conjunctions. In the example this transforms (10) into (11).

The pseudo code of the GCT algorithm is the following:

```
Function GCT(P, gf) -> 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:=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
```

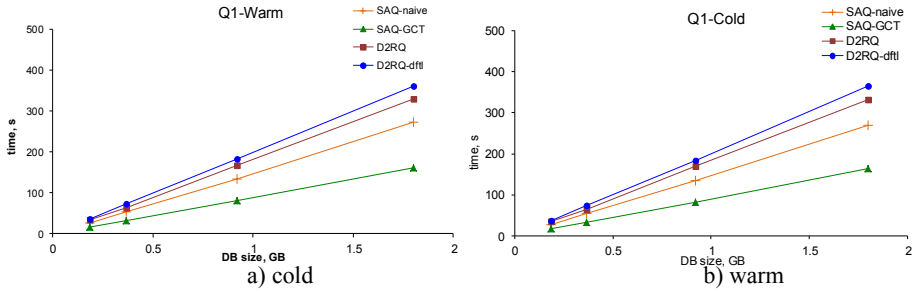
The function *orify(x,y)* forms a disjunction between predicates *x* and *y*, and *andify(x,y)* forms a conjunction.

We notice 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 Performance

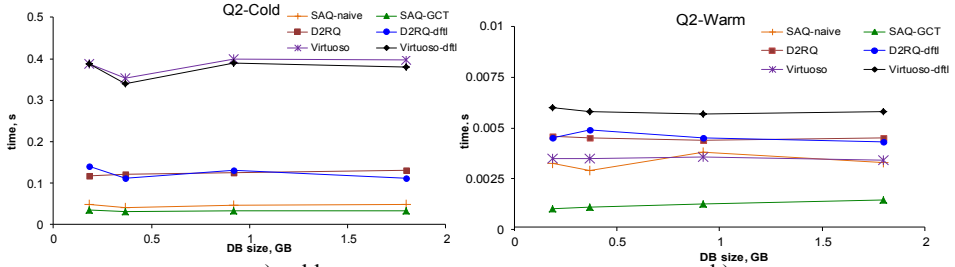
The measurements were made on a PC Intel(R) Core(TM), 2Quad CPU Q9400 with 2.67 GHz and 8 GB RAM running 64-bits Windows 7 Professional. The impact of GCT was evaluated for the unbound-property queries *Q1, Q2, Q3,* and *Q4*. We compare the performance of SAQ with Virtuoso RDF Views [8] and D2RQ [4], all systems accessing the same back-end MS SQL Server database. The experiment configuration was the following:

1. MS SQL server 2008 was configured with the default settings for the min and max server memory.
2. The SQL data was generated by the Berlin benchmark data generator [1][2] and loaded into the relational database.
3. Non-clustered, non-unique indexes were put on the columns *country* and *name* in the *person* table to speed up queries *Q3* and *Q4*.
4. For Virtuoso RDF Views, the RDF view to the underlying relational database was generated on the Virtuoso server (ver. 06.02.3128, Windows-64) by using the *Virtuoso Conductor* tool. The SPARQL queries to the RDF view were run from a Java program, implementing a *Jena Provider* [23], which allows users to query Virtuoso RDF views from Java. The Java heap size was set to 1 GB.
5. For D2RQ (v.07), the RDF view of the underlying RDBMS was generated by D2RQ's auto-generated mapping script [3]. In the generated script we inserted the option "*d2rq:useAllOptimizations true"* to guarantee that we use full optimization in D2RQ. The SPARQL queries were run from a Java program calling the D2RQ Engine through Jena2 [3]. The Java heap size was set to 1 GB.
6. All measurements were made five times and the mean values plotted. The standard deviation was less than 10% in all measurements.
7. The default mappings of the analyzed systems SAQ, Virtuoso RDF Views, and D2RQ all produce different results. For example, some redundant labels and inverse properties are produced by Virtuoso RDF Views and D2RQ. To make fair comparisons we configured the systems so that they all generated the same query



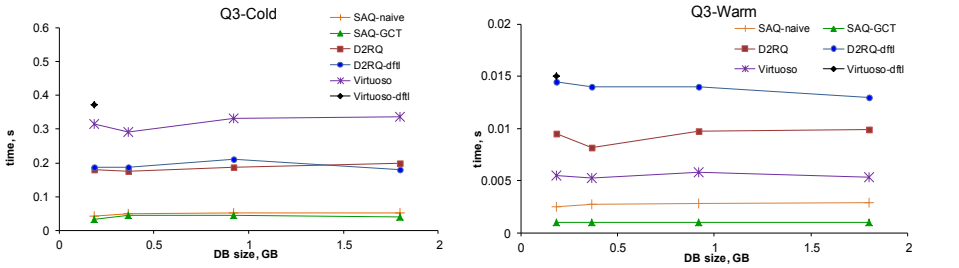
a) cold b) warm

Fig. 4. Execution times for Q1 up to 1.8 GB database



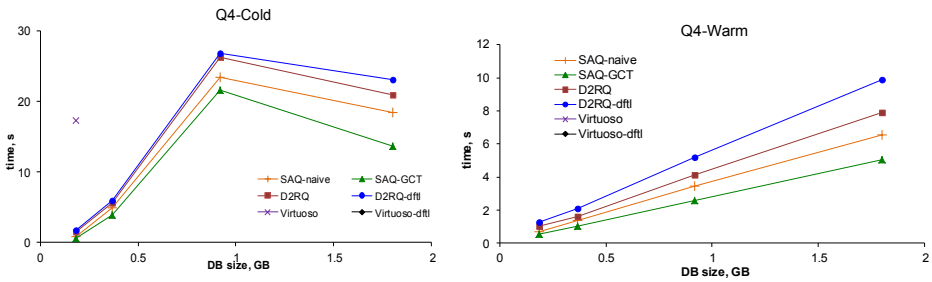
a) cold b) warm

Fig. 5. Execution times for Q2 up to 1.8 GB database



a) cold b) warm

Fig. 6. Execution times for Q3 up to 1.8 GB database



a) cold b) warm

Fig. 7. Execution times for Q4 up to 1.8 GB database

result. To investigate whether the performance is better with the default mappings we also measured Virtuoso RDF Views and D2RQ with their default mappings. The following notation is used in the performance diagrams:

- ***Virtuoso***: Virtuoso RDF Views configured with the SAQ mappings.
- ***Virtuoso-dflt***: Virtuoso RDF Views configured with the system default mappings.
- ***D2RQ***: D2RQ configured with the SAQ mappings.
- ***D2RQ-dflt***: D2RQ configured with the system default mappings
- ***SAQ-naive***: SAQ without GCT
- ***SAQ-GCT*:** SAQ with GCT

In all cases the time spent in executing the query by the relational database followed by post-processing is measured, thus not including the time for preparing the SPARQL query by the respective system. The cold execution measurements were made immediately after flushing the buffer pool, while the warm ones were made by re-executing the query immediately after a cold query was run. The cold execution times include reading data from disk and SQL query optimization in the DBMS server. Since the back-end DBMS has a statement cache a same SQL query executed twice will be optimized the first time it is received. Therefore, the warm executions do not include back-end DBMS query optimization time.

Table 1 Speed-up of ***SAQ-GCT*** compared to other approaches

Q1						
system	***SAQ-GCT***	***SAQ-naive***	***D2RQ***	***D2RQ-dftl***	***Virtuoso***	***Virtuoso-dftl***
cold	1	1.65	2	~ 2.2	>8 hours	>8hours
warm	1	1.6	2	~ 2.2	>8hours	>8hours
SQL queries	1	11	11	13	>1000	>1000
Q2						
cold	1	1.3 - 1.5	3.5 - 4	3.5 - 4	11.4 - 12.4	11 - 12
warm	1	2.2 - 3.2	3 - 4.5	3 - 4.5	2.3 - 3.5	4 - 6
SQL queries	1	10	4	4	11	18
Q3						
cold	1	1.2 - 1.3	4.2 - 5	4 - 5.5	6.6 - 9	11 - 480
warm	1	2.5 - 3	8 - 9.7	13 - 14	5.3 - 5.7	15 - 380
SQL queries	1	6	6	8	9	12
Q4						
cold	1	1.2 - 1.3	1.2 - 1.5	1.2 - 2.8	28 - 800	100 - 3000
warm	1	1.3	1.6 - 2	2	30 - 2200	120 - 8000
SQL queries	1	6	6	8	>1000	>1000

The results from the measurements are presented in Fig. 4-7. The figures show the execution times for *Q1, Q2, Q3,* and *Q4* while scaling the generated Berlin benchmark dataset from 10M to 100M [1][2], which corresponds to scaling the relational database from 312 MB to 1.8 GB. The number of RDF triples in the RD-view varied from 3 949 935 to 38 771 340.

Table 1 summarizes the speed-up of the different approaches for *Q1-Q4* compared to ***SAQ-GCT***. In particular, the ***SAQ-naive*** column shows the speed-up of GCT.

The performance of ***SAQ-GCT*** for simple unbound-property queries is better than all compared implementations. Furthermore, GCT always improves performance substantially (20-65%) for queries to cold databases, where the execution time is dominated by disk accesses on the database server. For the queries *Q2* and *Q3,* which select a single row from the buffer pool in a warm database, the improvement is even better (220-320%). The reason is that without GCT more SQL requests are sent to the server and therefore the communication overhead dominates when the server time is insignificant.

To analyze how the other systems process unbound-property queries we measured their performance and investigated what SQL queries were sent to the relational database. For ***D2RQ*** we used the profiling tool of MS SQL Server 2008 to obtain the SQL queries sent to the DBMS. Normally ***D2RQ*** sends exactly the same SQL queries as ***SAQ-naive*** so GCT is not used. For *Q2* ***D2RQ*** makes a special optimization when the subject of a triple pattern is a constant URI so fewer queries are sent.

Virtuoso RDF Views translates unbound-property queries to SQL using an unknown algorithm [7][8]. The debug logging of Virtuoso was used to investigate what SQL queries were sent to the relational database. For *Q2* and *Q3* ***Virtuoso*** also sends exactly the same queries as **SAQ-naive** plus a number of additional queries. For the non-selective queries *Q1* and *Q4* more than 1000 additional SQL queries were sent and the processing did not scale for large databases.

6 Related Work

Virtuoso RDF Views [7][8] and D2RQ [3][4][5] are other systems that allow mapping of relational tables and views into RDF to make them queryable by SPARQL. These systems implement compilers that translate SPARQL directly to SQL. By 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 the particular query transformation GCT significantly improves performance for unbound-property queries.

We did not find any publication of how D2RQ compiles unbound-property SPARQL queries into SQL. The documentation for Virtuoso is very limited [7][8]. 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 DBMS, showing that neither of those systems uses anything similar to GCT. SquirrelRDF also allows SPARQL queries to relational tables, but does not support unbound-property SPARQL queries [19] [20].

Work on optimizing disjunctive database queries in general is described in [6][11][13]. The closest work to GCT is the combinatorial algorithm [13], 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. By contrast, the purpose of GCT is to group in a DNF predicate query fragments that can be translated to SQL, and therefore the simpler linear GCT algorithm can be used.

The idea of bypass evaluation of disjunctive queries in [6][11] is based on implementing specialized operators that produce two output streams: the true-stream of the tuples that fulfill the operator's predicate and the false-stream of the tuples that do not match. The main profit 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

We have presented an approach to optimize simple unbound-property SPARQL queries to RDF views over back-end relational databases in a system called SAQ for querying and archiving relational databases as RDF. Simple unbound-property queries retrieve dynamic sets of properties for given subjects, which is important for archiving selected parts of a database with SPARQL. Such queries are optimized by the presented GCT (Group Common Terms) query transformation rule, which groups those common terms from a DNF predicate that can be translated to SQL.

By using data from the Berlin SPARQL benchmark, GCT was shown to improve query execution time compared to naïve processing. Compared to not using GCT, it reduces the number of SQL queries to execute and retrieves data in relational row order rather than column order. The performance of SAQ was compared to other systems that support SPARQL queries to views of existing relational databases. It was shown experimentally that SAQ with GCT performs better than those systems, since they do not use any similar transformation strategy.

Future work includes investigating the impact of GCT and other rewrite rules on the performance of other kinds of queries, such as queries with multiple unbound-property triple patterns and other kinds of archival queries.

References

1. Bizer, C. and Schultz, A.: The Berlin SPARQL Benchmark (BSBM) Specification – V3.0, http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/index.html (2010).
2. Bizer, C. and Schulz, A.: 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)
3. Bizer, C., Cyganiak R., Garbers, G., Maresch, O., and Becker C.: The D2RQ Platform v0.7 - Treating Non-RDF Relational Databases as Virtual RDF Graph. http://www4.wiwiss.fu-berlin.de/bizer/d2rq/spec/ (2009)
4. Bizer, C. and Seaborne A.: D2RQ-Treating Non-RDF Databases as Virtual RDF Graphs, Poster at 3rd International Semantic Web Conference (2004)
5. Bizer, C. and Cyganiak, R.: D2R Server-Publishing Relational Databases on the Semantic Web, Poster at the 5th International Semantic Web Conference (2006)
6. Claussen, J., Kemper, A., Peithner, K., and Steinbrunn, M.: Optimization and Evaluation of Disjunctive Queries, IEEE Transactions on Knowledge and Data Engineering, Vol. 12, No 12, March/April (2000)

7. Erling, O.: Declaring RDF views of SQL Data, W3C Workshop on RDF Access to Relational Databases, 25-26 October, Cambridge, MA, USA (2007)
8. Erling, O and Mikhailov, I.: RDF Support in the Virtuoso DBMS, Springer, ISSN: 1860-949X (Print) 1860-9503 (Online), in Studies in Computational Intelligence, Vol. 221(2009)
9. Fahl, G. and Risch, T.: Query Processing over Object Views of Relational Data, The VLDB Journal , Vol. 6, No. 4, pp 261-281 (1997)
10. Garcia-Molina, H., Ullman, J. D. and Widom, J.: Database Systems, the complete book, Prentice Hall (2002).
11. Kemper, A., Moerkotte, G., Pethner, K., and Steinbrunn, M.: Optimizing Disjunctive Queries with Expensive Predicates, in Proc. ACM SIGMOD, Conf. Management of Data, pp. 336-347 (1994)
12. Litwin, W. and Risch, T.: 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)
13. Muralikrishna, M. and Witt, D. J De.: Optimization of Multiple-Relation Multiple-Disjunct Queries, PODS '88 Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART, pp. 263-275 (1988)
14. N-triples, W3C RDF Core WG Internal Working Draft, http://www.w3.org/2001/sw/RDFCore/ntriples/
15. Petrini, J. and Risch, T.: 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. Petrini, J.: Querying RDF Schema Views of Relational Databases, PhD Thesis, Uppsala University, Department of IT, ISSN 1104-2516, http://www.it.uu.se/research/group/udbl/Theses/JohanPetriniPhD.pdf (2008)
17. Risch, T. and Josifovski, V.: Distributed Data Integration by Object-Oriented Mediator Servers, Concurrency and Computation: Practice and Experience, J. 13(11), John Wiley & Sons (2001)
18. Schmidt, M., Hornung, T., Lausen, G. and Pinkel, C.: SP2Bench: A SPARQL Performance Benchmark, ICDE 2009, pp. 222-233 (2009)
19. Seaborne, A., Steer, D., and Williams, S.: SQL-RDF, http://www.w3.org/2007/03/RdfRDB/papers/seaborne.html (2007)
20. SqirrelRDF, http://jena.sourceforge.net/SquirrelRDF/
21. Stuckenschmidt, H. and Harmelen, F.: Information Sharing on the Semantic Web, Springer, ISBN 3-540-20594-2 (2005)
22. Sören, A., Feigenbaum, L., Miranker, D., Fogarolli, A., and Sequeda J.: Use Cases and Requirements for Mapping Relational Databases to RDF, W3C Working Draft 8, http://www.w3.org/TR/rdb2rdf-ucr/ (2010)
23. Virtuoso Jena Provider, OpenLink Virtuoso Universal Server: Documentation, http://docs.openlinksw.com/virtuoso/rdfnativestorageproviders.html#Rdfnativestorageprovidersjena (2009)