Scalable RDF Views of Relational Databases through Partial Evaluation

Johan Petrini and Tore Risch

Department of Information Technology
Uppsala University
Sweden
{Johan.Petrini,Tore.Risch}@it.uu.se

Abstract

The semantic web represents meta-data as a triple relation using the RDF data model. We have developed a system to process queries to RDF views of entire relational databases. Optimization of queries to such views is challenging because i) RDF views of entire relational databases become large unions, and ii) queries to the views are more general than relational database queries, making no clear distinction between data and schema. As queries need not be expressed in terms of a schema, it becomes critical to optimize not only data access time but also the time to perform the query optimization itself. We have developed novel query optimization techniques for scalable queries to RDF views of relational databases. Our optimization techniques are based on partial evaluation, a method for compile time evaluation of subexpressions. We show dramatic improvements in query optimization time when scaling the query size while still producing high quality execution plans. Our query optimization techniques enable execution of real-world queries to RDF views of relational databases.

1. Introduction

The semantic web initiative [2] aims at providing Internet-wide standards for semantically enriching and describing web resources. Using the standards RDF [23] and RDF-Schema (RDFS) [19] any resource can be annotated with *properties* describing its structure and contents. The resources are not limited to web pages but include everything that can be identified through the web. Semantic web applications, for example Dublin Core [10], Open Directory [16], RSS 1.0 [21], use RDF for different kinds of tasks.

RDF repository systems [5][6][9][19][27] offer storage of RDF data and the ability to search RDF data using a query language. However, as most information still resides in relational databases, it is desirable that this information is also exposed to the semantic web through RDF.

The SWARD (Semantic Web Abridged Relational Databases) system provides scalable RDF views of all data stored in an existing relational database. The system supports general queries over these views in a query language. Because RDF views include both schema data and table content data they are very flexible and. Unlike SQL, queries can mix meta-data and table access, e.g. a query can easily be expressed that finds those columns in the relational database containing the value 'Johan'.

The semantic web represents a database as a collection of *RDF triples*[1] on the format *(s, p, v)* where *s* is called the *subject* (modeling some entity), p is called the *predicate* (modeling some property of an entity) of s, and *v* is called the *object* (the value of p). To avoid confusion with ordinary programming terminology we use the terms *property* and *value* instead of *predicate* and *object*. The objects representing *s*, *p*, and *v* are called *RDF resources*. To uniquely identify an object RDF uses URIs [4].

SWARD represents RDF triples as a large union view of queries extracting the exported data from a relational database. Such a view is called the *triple extent view* for the relational database. The triple extent view provides a very general relational view of the database, similar to the 'reference relations' with four columns proposed by [14].

A triple extent view is defined as a large union of subqueries each accessing one exported column in the relational database. Such a subquery is called a *column extent view*. As real-life relational databases often have many columns, queries to the triple extent view require

[1] Called *statements* in RDF.

efficient processing of queries over large unions of many column extent views. In this paper we evaluate a number of different strategies to achieve scalability with respect to i) query execution as the database increases, and ii) query optimization as the size of the query increases. The latter turns out to be critical for RDF queries to triple extent views. The reason is that RDF queries generate many self-joins to large triple extent union views. We show that traditional query processing does not scale w.r.t. query optimization time.

It is particularly important that queries that do not access the database schema but only its contents, *database contents queries*, are processed efficiently. These are the kinds of queries that are normally used in relational databases and it is desirable that they scale. Such RDF queries contain explicit references to RDF resources identifying relational table columns, called *column identifiers*. In this paper we show how to optimize conjunctive database contents queries to triple extent views of relational databases.

SWARD allows the triple extent view definitions to be generated semi-automatically. The system generates triples extent views of relational databases in terms of automatically generated *RDF proxy resources* identifying data from a relational database. The identifiers for these RDF proxy resources are systematically generated from the database schema. However, the user normally defines the queries in terms of a predefined set of named RDF properties, usually called a *terminology* (or ontology), e.g. Dublin Core [10]. This terminology is normally different from automatically generated RDF proxy resources. Therefore, when defining a SWARD wrapper of a relational database the implementer also provides an optional mapping table between the *external* resources in the standard terminology and the automatically generated *internal* resources. This mapping information is part of SWARD and is used by the triple extent view definitions.

We have evaluated the following query processing strategies for queries to a triple extent view:

Expand-normalize-optimize (ENO)

The naïve ENO strategy uses a rather traditional query optimization technique doing i) view expansion of the triple extent view and the column extent views, ii) normalization to disjunctive normal form producing a large union of conjunctive subqueries, iii) elimination of common subexpressions in the subqueries when accessing several columns in the same relation [11], and iv) query decomposition of each subquery. After decomposition each subquery plan contains a call to SQL combined with some pre and post-processing. This produces a good decomposed execution plan but at very high query optimization cost because of the normalization.

ENO with partial evaluation, ENOP

The reason for slow query processing with ENO is that an expanded union view becomes large and this causes the normalization to be very slow. In particular, assuming that the column identifiers are known, it turns out that for each reference in the query to the triple extent view; only one subquery in the view is actually relevant. Thus if we could infer this before normalization it would improve optimization time significantly. To do this we apply a technique, partial evaluation [17], where the query is simplified by iteratively evaluating at compile time some predicates until a fix-point is reached. Basically, partial evaluation simplifies SWARD queries substantially by compile time evaluation of i) system functions managing RDF objects and ii) system tables storing meta-data about the wrapped relational database. Compile time calls to the terminology mapping table enables reduction to a simple conjunctive query, so no normalization is needed.

Select, expand, and partially evaluate, SEP

We observe that the reason ENOP scales is that partial evaluation enables the optimizer to prune away all subqueries except one from view expanded query. Partial evaluation then walks through the expanded expression to eliminate most code. However, the expanded union view is large. We avoid expanding the entire triple extent view by selecting at compile time relevant fragments of the triple extent view from pre-generated column extent views stored in SWARD.

Evaluation

The database scalability results are verified by performance evaluations using queries to a triple extent view of a TPC-H based database [26]. We scale both the size of the database and the size of the query. Dramatic performance improvements are shown for query compilation times and many queries cannot be executed at all without the proposed methods.

As initial query language we use RDQL [22] which is widely accepted in the semantic web community. Because queries to triple views are naturally expressed in domain calculus it is used as internal query representation in SWARD. Our approach applies to other proposed semantic web query languages (e.g. [13][24][25]) as well.

The rest of the paper is organized as follows: In Section 2 we describe related work. Then Section 3 defines a running example to illustrate the technique. Section 4 describes how triple extent views are defined and queried. Section 5 describes the strategies for scalable query optimization of RDF queries followed by a

performance evaluation in Section 6. Finally, Section 7 summarizes our results and outlines future work.

2. Related Work

RDF repository systems [5][6][9][19][26] often use relational databases internally. This relational database is fully managed by the repository system and the schema of the relational database is internal. If one wants to make RDF queries to a relational database using such a repository, it requires downloading the database into the repository. This clearly does not scale.

Rather than storing RDF data in dedicated RDF-repositories our work wraps an existing relational database so that it can be used in RDF queries without downloading database tables to a repository. Instead the data necessary for answering a particular query are represented as transient RDF triples streamed through the wrapper.

SWIM [9] and D2RQ [3] provide conversion methods from relational databases to RDF. D2RQ makes some optimizations to speed up the execution-time of RDQL queries. However, none of the works study how to optimize queries over binary RDF views of relational databases.

The typed RDFS-based view specification language RVL [15] is proposed for semantic web integration [9]. It can complement SWARD by allowing the definition of RDF views on top of our triple extent views.

The reference relation by [14] proposes a flexible representation of a relational database as a four-column table. This enables very general queries combining schema and data. Our triple extent views also provide the same flexibility and, in addition, support RDF mappings.

Optimizing disjunctive queries in general was studied by, e.g., [7] without any use of partial evaluation and without paying attention to query optimization time.

The purpose of partial evaluation [17] is to simplify programs by compile time evaluation of functions and expressions. The technique was invented by Futamura [12] in the 70s and has been applied on various programming languages, e.g., Lisp [1] and C [8]. For query optimization it has been used very little, mainly for optimizing mediator queries [18].

To summarize, we are not aware on any other work on optimizing query processing of very large disjunctive queries as is needed for RDF views of relational databases.

3. Example

To illustrate our methods we use an example based on the TPC-H benchmark database [26] stored in a backend commercial relational database system. A company owns a relational database used for decision support. The company decides that they want to expose customers and orders in the following two tables:

customer	Custid	name	mktsegment
	120	'Smith'	'Automobile'

orders	orderid	custid	price	clerk	Comment
	1	120	22000	'Wesson'	'SAAB, on time'
	2	120	20000	'Doe'	'Volvo, late'

Figure 1: Example database.

The column *custid* is foreign key in relation *orders*. The following SWARD statement automatically generates the triple extent view for the exported tables:

```
ExportRDB('udbl.it.uu.se:3050/company/',
          {'orders', 'customers'},
          'orderTerminology',nil)
```

The relational database is identified by the URI *udbl.it.uu.se:3050/company/*[2]. The second argument is the list of tables to export. The third argument is the name of a user provided *terminology mapping table* to map between the *internal* column identifiers systematically generated by SWARD to the corresponding *external* column identifiers used in the RDF queries. The fourth argument is an optional list of columns to be excluded from the triple extent view. In our example we use the following terminology mapping table:

internal column id	External column id
udbl.it.uu.se:3050/company/customer/custid	udbl.it.uu.se/terms#custid
udbl.it.uu.se:3050/company/customer/name	udbl.it.uu.se/terms#name
udbl.it.uu.se:3050/company/customer/mktsegment	udbl.it.uu.se:3050/company/customer/mktsegment
udbl.it.uu.se:3050/company/orders/orderid	udbl.it.uu.se/terms#ordered
udbl.it.uu.se:3050/company/orders/custid	udbl.it.uu.se/terms#o_custid
udbl.it.uu.se:3050/company/orders/price	udbl.it.uu.se/terms#price
udbl.it.uu.se:3050/company/orders/clerk	udbl.it.uu.se/terms#clerk
udbl.it.uu.se:3050/company/orders/comment	udbl.it.uu.se/terms#comment

Table 1: The terminology mapping table used in our example.

Notice that, as the user has not provided any external column identifier for column *mktsegment* in the terminology mapping table, the system makes the internal and external identifiers the same.

With the above terminology table the first rows in table customer and orders will produce the triples in Figure 2. Section 4 explains the rules for how *co:triples* is defined as a view.

[2] It is based on the JDBC naming convention.

co:triples	s	P	v
	udbl.it.uu.se:3050/company/customer/custid/120	Udbl.it.uu.se/terms#custid	120
	udbl.it.uu.se:3050/company/customer/custid/120	Udbl.it.uu.se/terms#name	'Smith'
	udbl.it.uu.se:3050/company/customer/custid/120	Udbl.it.uu.se:3050/company/customer/mktsegment	'AUTOMOBILE'
	udbl.it.uu.se:3050/company/orders/ordered/1	Udbl.it.uu.se/terms#orderid	1
	udbl.it.uu.se:3050/company/orders/ordered/1	Udbl.it.uu.se/terms#o_custid	120
	udbl.it.uu.se:3050/company/orders/ordered/1	Udbl.it.uu.se/terms#price	22000
	udbl.it.uu.se:3050/company/orders/ordered/1	Udbl.it.uu.se/terms#clerk	'Wesson'
	udbl.it.uu.se:3050/company/orders/ordered/1	Udbl.it.uu.se/terms#comment	'SAAB, on time'

Figure 2: Part of triple extent for example database.

The example RDQL query to SWARD in Figure 3 returns all late orders and their comments. We use this example throughout the paper to show how queries are processed in SWARD.

```
SELECT ?o, ?c
FROM <udbl.it.uu.se:3050/company/>
WHERE
(?o,<term:comment>,?c) AND
?c =~ '/late/'
USING term FOR <udbl.it.uu.se/terms#>
```

Figure 3: Example RDQL query Q1.

In RDQL URIs are specified on the form <…> and variables are prefixed with '?'. The SELECT clause specifies the variables to be returned to the user. The FROM clause specifies the universe of RDF triples to query, in our case an identifier for the triple extent view of a relational database. The WHERE clause specifies a selection condition over the RDF triples in the view. The selections are specified using the notation *(s,p,v)* where *s, p,* and *v* are constants or variables. Special infix notation is used for string comparison =~ (i.e. like). The USING clause specifies prefixes (namespaces) used in URIs.

The parser first translates the RDQL query to the domain calculus expression in Figure 4. Triple selections are translated into calls to the triple extent view, *co:triples*. The prefix *co:* identifies our relational database; it is substituted with *udbl.it.uu.se:3050/company/*. For every URIs reference in the query, e.g. <*term:comment*>, the parser creates a SWARD object holding an RDF resource, called a *URI object*. As in RDQL we also use the notation <*xxx*> in our domain calculus expressions to represent a URI object with identifier *xxx*.

```
{o, c |                                                    (1)
 co:triples(o,<udbl.it.uu.se/terms#comment>,c)
                                                  AND (2)
 like(c,'*late*')                                          (3)
}
```

Figure 4: Domain calculus expression of example RDQL query.

The optimizer transforms the domain calculus expression into an execution plan containing SQL calls.

The result from the RDQL query is the tuple[3]:

```
(<udbl.it.uu.se:3050/company/orders/orderid/2>,
 'Volvo, late')
```

Here <*udbl.it.uu.se:3050/company/orders/orderid/2*> is a system generated URI object that identifies the row with key '2' in *orders* and 'Volvo, late' is the value of the column *comment* for the row.

4. Triple extent views

This section describes how triple extent views are represented and automatically generated. The generation is driven by a meta-schema in SWARD, the *SWARD Schema*, representing RDF resources, descriptions of wrapped relational databases, and descriptions of triple extent views. Notice that query results are not permanently stored in SWARD but are dynamically created by SWARD, streamed through the system, and automatically removed by a garbage collector when no longer needed.

4.1 The SWARD schema

Figure 5 shows the SWARD schema. An RDF resource (type *Resource*) represents data that can be described by RDF. It is either a *reference* to some data item identified by a URI (c.f. an OID) or a literal (type *RDFLiteral*). The system needs to distinguish between different kinds of RDF resources. For example, one needs to distinguish between a URI and an RDF literal with the same value as the URI identifier (c.f. separating OIDs from strings). Therefore SWARD represents resources as *objects* where one attribute denotes its value and a flag indicates whether the value is a URI or an RDF Literal. Furthermore, objects also allow representing various other attributes of RDF resources, as well as representation of RDF schema data [20]. URI objects are resources representing URIs. Notice that, following the RDF Schema standard [20] there is no special type representing URI objects. Thus URI objects are those resources not being literals.

To allow more than one data source, different triple extents views can be generated for different data sources, e.g., representing different relational databases. This is

[3] We use the (..) notation for tuples.

achieved by representing each data source as an instance of type *Source*. Each source has an attribute called *TEV,* representing its triple extent view. Each kind of source also has an associated *triple extent view generator*, such as *ExportRDB*, that automatically generates the triple extent view definition. The relationship *triples* associates a set of RDF triples with a data source. This set is not explicitly stored, but implicitly produced by the TEV associated with each data source.

Following the RDF Schema definition [20], the RDF triples themselves are by default not RDF resources and have no URIs. Therefore type *Triple* is not a subtype of *Resource*. However, in RDF Schema the user can explicitly represent statements as being resources with an associated URI by defining them as *reified statements* [20]. Reified statements are represented as objects of type *ReifiedTriple*. The relationship *reifies* associates an RDF triple with its corresponding *ReifiedTriple*. Reified statements are outside the scope of this paper.

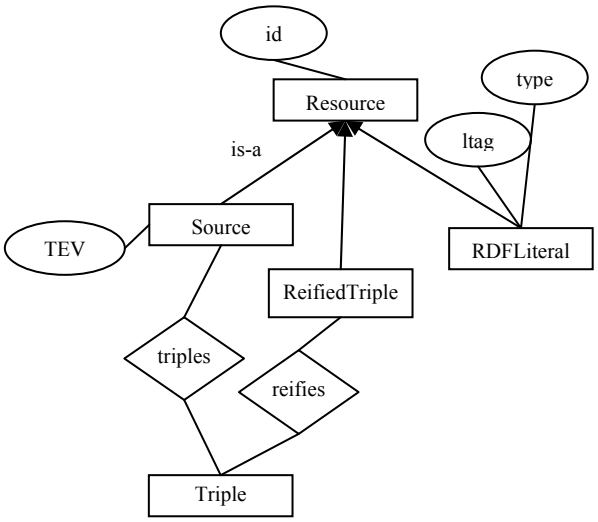

Figure 5: The SWARD schema.

Objects of type *Resource* have one important attribute, *id*. It stores a string *identifying* the resource. For URI objects this is a unique identifier, i.e., it is the key for the resource.

RDF literals, represented as objects of type *RDFLiteral,* do not represent URIs but literal values. They have the following attributes:

- The attribute *id* stores the value of the literal and is *not* a key for literals.
- The attribute *ltag* stores a string identifying the language of the literal.
- The attribute *type* stores for *typed RDF literals* [23] a URI object representing the type of the literal.

Thus an RDF literal is uniquely identified by the combination of the attributes *id*, *ltag* and *type* (i.e. a compound key) while URI objects are identified by the key *id*. For supporting typed literals the attribute *type* has to be taken into account as well. Without loss of generality in the discussion below we ignore attributes *ltag* and *type* for literals; thus we discuss only *plain* literals [23] which are always strings. Two plain literals are the same if they have the same *id*. However, two objects of type *Resource* with the same *id* need not be the same because they can be either RDF literals or URI objects.

A relational database is represented by a URI object identified by a relational database identifier. The identifier is constructed by concatenating the URL for the database server with the database name, e.g. "*udbl.it.uu.se:3050/company/"*. The subjects of RDF triples in the triple extent view of a relational database identify table rows and their URI objects are called *row identifiers*. The triple extent properties identify columns of tables and their URI objects are called *column identifiers*. The triple extent values represent table *column values*. They are always RDF literals.

Column identifiers are constructed by concatenating the relational database identifier with the name of the table owning the column and the name of the column, e.g. *'udbl.it.uu.se:3050/company/orders/orderid'*. Row identifiers are constructed by concatenating a column identifier for the key column with a key value, e.g. '*udbl.it.uu.se:3050/company/orders/orderid/220'*. Compound keys are outside the scope of this paper.

We will proceed by describing how relational data are represented as automatically generated triple extent views in SWARD.

4.2 Defining triple extent views for relational databases

Each RDQL query references one triple extent view in the FROM clause. In our example it is the triple extent view for the database in Figure 1. The query processor accesses the triple extent view definition to translate the query into an algebra expression containing one or several calls to SQL. The triple extent view T_r of the entire relational database identified by r is defined as a union of column extent views T_c generated for each column named c for each table in r, i.e. $T_r = \cup T_c$ for all c in r. Each column extent view is defined as a domain calculus view whose name is identified by the internal column identifier. A column extent view defines the triples produced by that column.

Figure 6 shows the generated triple extent view for the example database as a union in terms of column extent views.

```
co:triples(s, p, v):
 cu:custid(s, p, v)                    OR
 cu:name(s, p, v)                      OR
 cu:mktsegment(s, p, v)                OR
 ord:orderid(s, p, v)                  OR
 ord:custid(s, p, v)                   OR
 ord:price(s, p, v)                    OR
 ord:clerk(s, p, v)                    OR
 ord:comment(s, p, v)
```

Figure 6: Triple extent view definition for example database.

Figure 7 shows a template for definition of the column extent view *ord:comment(s, p, v)*. The system subsitutes *ord:* with *udbl.it.uu.se:3050/company/orders* and *cu:* with *udbl.it.uu.se:3050/company/customer.*

```
ord:comment(s, p, v):                                  (1)
orders(orderid, custid, price, clerk,
       comment)                                AND (2)
 rid = rowid(KCID,orderid)                     AND (3)
 s = uri(rid)                                  AND (4)
 p = uri(externalize(CID,TERMS))               AND (5)
 v = lit(comment)                                   (6)
```

Figure 7: Column extent view template for *comment*

The template in Figure 7 contains some capitalized symbols that the triple extent view generator substitutes to produce the definition in Figure 8. *KCID* on line (3) is substituted with the internal column identifier for the key of table *orders*. The function *rowid* returns a unique internal table row identifier by string concatenation, e.g. *'udbl.it.uu.se:3050/company/orders/orderid/2'*. The function *uri* on line (4) constructs a new URI object as a proxy resource identified by that row identifier. *CID* on line (5) is substituted with the internal column identifier for the column *comment. TERMS* is substituted with the name of the terminology table used in the triple extent view. The function *externalize* returns an external identifier for a given internal column identifier *CID* and a terminology table *TERMS*. Finally, on line (6) the RDF literal proxy object representing a column value is constructed. Figure 8 shows *ord:comment* after all substitutions:

```
ord:comment(s, p, v):                                  (1)
orders(orderid, custid, price, clerk,
       comment)                                AND (2)
rid = rowid(
'udbl.it.uu.se:3050/company/orders/orderid',
 orderid)                                      AND (3)
 s = uri(rid)                                  AND (4)
 p = uri(externalize(
'udbl.it.uu.se:3050/company/orders/comment',
 'orderTerminology'))                          AND (5)
 v = lit(comment)                                   (6)
```

Figure 8: Column extent view.

After view expansion of all column extent views the disjunctive definition in Figure 6 will have eight conjunctive subqueries. The definition is on disjunctive normal form. In the next section we will show how queries to such disjunctive views are processed efficiently. A particular problem is that queries are conjunctive and this makes naïve query processing slow.

Notice that the expression in Figure 4 contains only one reference to the triple extent view (line (3)). However, most real-world queries will contain many self-joins and this will make the expanded expression huge. The next section investigates query processing strategies to handle this complexity.

5. Query processing

Figure 9 illustrates the query processor of SWARD. Applications access SWARD through its *query interface*. When a user executes an RDQL query it is first transformed by the *parser* into a domain calculus expression, e.g. the expression in Figure 4.

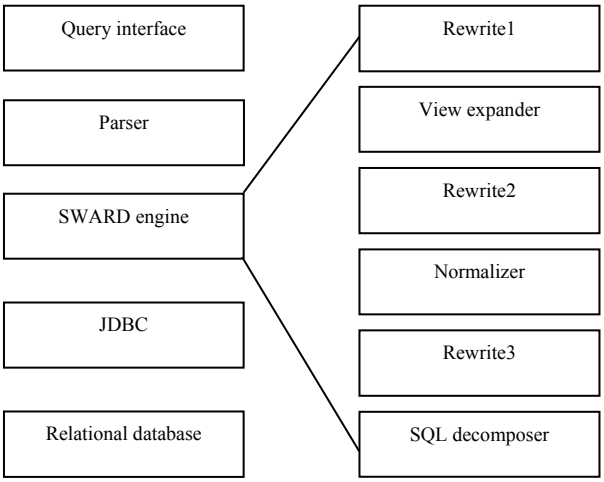

Figure 9: System architecture.

The steps *rewrite1,2,3* perform query simplification such as partial evaluation and elimination of common subexpressions.

The *view expander* substitutes each reference to the triple extent view in the query with its definition. In our example this first produces the expression illustrated by Figure 10. Then the column extent views are expanded which produces a large view partially illustrated by Figure 11. The expanded view becomes very large which slows down query optimization. Therefore we have developed methods to minimize the size of the expanded query based on partial evaluation.

```
{o, c |                                              (1)
(cu:custid(o,<udbl.it.uu.se/terms#comment>,c)
                                        OR           (2)
cu:name(o,<udbl.it.uu.se/terms#comment>,c)
                                        OR           (3)
cu:mktsegment(o,<udbl.it.uu.se/terms#comment>,c)
                                        OR           (4)
ord:orderid(o,<udbl.it.uu.se/terms#comment>,c)
                                        OR           (5)
ord:custid(o,<udbl.it.uu.se/terms#comment>,c)
                                        OR           (6)
ord:price(o,<udbl.it.uu.se/terms#comment>,c)
                                        OR           (7)
ord:clerk(o,<udbl.it.uu.se/terms#comment>,c)
                                        OR           (8)
ord:comment(o,<udbl.it.uu.se/terms#comment>,c)
)                                          AND       (9)
 like(c,'*late*')                                    (10)
}
```

Figure 10: Example query after expanding the triple extent view.

The *normalizer* transforms the simplified query to disjunctive normal form (i.e., a union of conjunctive subqueries). Normalization improves query execution by combining in the same conjunctive subqueries predicates from the query and predicates from column extent view definitions. Disjunctive normal form is used because the triple extent views are already on that format. We will show that normalization produces efficient query execution plans but at a very high query optimization cost. Elimination of normalization is therefore an issue that is addressed in this work.

The normalized expression is simplified once again by *rewriter3*. Finally the SQL decomposer [11] translates each simplified conjunctive subquery into an algebra expression. The algebra expression contains calls to a foreign function sending SQL statements to the *relational database* via *JDBC*. The only SQL statement submitted to the relational database in our example is:

```
select comment
from orders
where comment like '%late%'
```

The algebra expression further contains pre- and post-processing to manage proxy resources. The execution plan is a big union of these algebra expressions. The algebra expression is finally interpreted.

Next we investigate how our query simplification strategies improve query processing.

5.1 ENO, expand-normalize-optimize

The naïve strategy *ENO, expand-normalize-optimize,* does not use partial evaluation at all in the rewrite steps. However, we still employ elimination of common subexpressions; otherwise the performance is completely unreasonable. The view expanded query (Figure 11) is a large conjunction of subqueries on disjunctive normal form. This expression is then normalized before query decomposition in order to push the query selections into the column extent view query fragments so that the selections (i.e. the LIKE condition) are moved into SQL. One major problem here is that the normalized expression becomes huge even for this trivial example, in particular for join queries. Real-world queries contain many joins and therefore generate unreasonably large expressions. A challenge we address here is to limit the query optimization time despite this complexity of the expanded triple extent views.

5.2 ENOP, ENO with partial evaluation

The strategy *ENOP, expand-normalize-and partially evaluate*, uses partial evaluation to improve query optimization time. All three rewrite steps are executed. The functions, *uri, lit,* and *externalize*, are declared to be evaluatable at compile time.

The column extent view reference on line (2) in Figure 4 is expanded with the value of *c1* bound to the proxy resource. After view expansion *rewrite2* will simplify each subquery in the disjunction in Figure 10. Consider the expansion of *ord:clerk(rid, <udbl.it.uu.se/terms#comment>, c)* shown in Figure 12.

```
{o,c |
  (
   Expanded column extent views for
   custid and name here.
                                          OR
    (customer(custid, name,
     mktsegment)                    AND
     rid = rowid(
'udbl.it.uu.se:3050/company/customer/custid',
     custid)                        AND
     o = uri(rid)                   AND
     <udbl.it.uu.se/terms#comment> =
      uri(externalize(
'udbl.it.uu.se:3050/company/customer/mktsegment'
   ,'orderTerminology'))            AND
      c = lit(mktsegment)
     )
                                          OR
   Expanded column extent views for
   mktsegment, ordered, custid, price,
   and clerk here
                                          OR
    (orders(orderid, custid, price, clerk,
     comment)                       AND
     rid = rowid(
'udbl.it.uu.se:3050/company/orders/orderid',
     orderid)                       AND
     o = uri(rid)                   AND
     <udbl.it.uu.se/terms#comment> =
      uri(externalize(
'udbl.it.uu.se:3050/company/orders/comment',
    'orderTerminology'))            AND
      c = lit(comment)
      )
   )                                        AND
 like(c,'*late*')
}
```

Figure 11: Part of example query after full view expansion

Compile time evaluation of *externalize* in clause (4) returns the string *'udbl.it.uu.se/terms#clerk'* by looking up the terminology table. Once *externalize* is evaluated it becomes the argument of *uri* which then is compile time

evaluated to the proxy resource *<udbl.it.uu.se/terms#clerk>*.

```
orders(orderid, custid, price, clerk,
       comment)                                  AND (1)
rid = rowid(
'udbl.it.uu.se:3050/company/orders/orderid',
 orderid)                                        AND (2)
 o = uri(rid)                                    AND (3)
 <udbl.it.uu.se/terms#comment> = uri(externalize(
'udbl.it.uu.se:3050/company/orders/clerk',
 'orderTerminology'))                            AND (4)
 c = lit(clerk)                                      (5)
```

Figure 12: Expanded column extent view *clerk* in query.

Then predicate in line (4) becomes false and so does the entire clause and can be deleted from the disjunction. Similarly partial evaluation will eliminate *all* subqueries in Figure 11 except the one referencing column *comment*. As only one clause remains in the disjunction no expensive normalization is needed. This happens for every reference to the triple extent view in conjunctive database contents RDF queries.

5.3 SEP, Select - Expand – Partially Evaluate

One problem with ENOP is that the triple extent view is expanded before *rewrite2* does the partial evaluation. Real-world databases often have many columns so the full triple extent views often become huge and therefore the partial evaluation itself may take time.

The method *SEP, Select – Expand – Partially Evaluate,* avoids partial evaluation of the expanded triple extent view by selecting for expansion only the relevant subqueries in the triple extent view. This is done by introducing a system table that, for a given internal column identifier *colid*, stores the definition of the corresponding column extent view definition *vd* together with a precompiled execution plan to retrieve the triples of the view:

```
columnExtent(colid, vd)
```

We can now define the triple extent view as in Figure 13 by introducing a system predicate:

```
applyTriples(vd, s, p, v)
```

applyTriples executes a precompiled view definition *vd* returning triples *(s, p, v)*.

```
co:triples(s, p, v):
 columnExtent(colid, vd)                    AND
 p = externalize(colid,'orderTerminology') AND
 applyTriples(vd, s, p, v)
```

Figure 13: Triple extent view definition using *columnExtent*

If we expand *co:triples* in Figure 4 with this definition we get the view expanded query in Figure 14.

```
{o, c |
 columnExtent(colid, vd)                    AND (1)
 <udbl.it.uu.se/terms#comment> =
  externalize(colid,'orderTerminology')  AND (2)
 applyTriples(vd, o,
            <udbl.it.uu.se/terms#comment>, c)
                                            AND (3)
like(c,'*late*')                                (4)
}
```

Figure 14: View expanded query using *columnExtent*

Partial evaluation of this query produces the plan in Figure 15. First, on line (2) the inverse of *externalize* is executed to bind *colid* to the internal column identifier *<udbl.it.uu.se:3050/company/orders/comment>* by looking up the terminology table. Then, *columnExtent* is evaluated to bind *vd* to an object representing the column extent view for column *comment*, which we denote *view:comment*.

```
{o, c |
 applyTriples(view:comment, o,
       <udbl.it.uu.se/terms#comment>, c) AND (1)
 like(c,'*late*')                                (2)
}
```

Figure 15: Partially evaluated view expanded query using *columnExtent*

At this point we introduce another rewrite rule that substitutes (view expands) *applyTriples* if its first argument is a constant. Line (1) is replaced with the definition of the column extent for *comment* in Figure 8, thus producing the query in Figure 16.

```
{o, c |                                             (1)
orders(orderid, custid, price, clerk,
       comment)                                  AND (2)
rid = rowid(
'udbl.it.uu.se:3050/company/orders/orderid',
 orderid)                                        AND (3)
 o = uri(rid)                                    AND (4)
 c = lit(comment)                                AND (5)
 like(c, '*late*')                                   (6)
```

Figure 16: After expansion of view definition for *comment*

This simplified query is sent to query decomposition where lines (2) and (6) create the same SQL statement as before and the rest is post-processing. Our performance evaluation shows the SEP scales excellently with the size of the RDF query, while producing the same optimal execution plan as ENOP.

6. Performance evaluation

Our performance evaluation measures scalability of ENO, ENOP, and SEP over both query and database sizes. As a reference we also test a method *TPD, total push-down*, that stores the triple extent view and the terminology table in the back-end DBMS. The experiments were run on a PC with 2.2 GHz CPU, 512 MB main memory and the Windows XP Professional operating system. We used Microsoft SQL server with 16 MB buffer size as back-end server to store the TPC-H benchmark database.

6.1 Scaling the size of the query

First we evaluate the scalability of our strategies with respect to the size of the query. This is particularly important for RDF queries, as RDF queries will contain many more joins than SQL. Every reference to a column identifier (i.e., every selection) will become a join in RDF queries. Thus, real-world RDF queries will contain many joins and the query processor must be able to handle this.

We increased the size of the RDQL query by introducing more triple joins and measured the optimization time. We use a synthetic example query where we systematically increase the number of joins in the query to reference increasingly more columns. Our synthetic query has the structure shown in Figure 17 where c_1 … c_k are column identifiers of a table *t* and *k* varies from 1 to 21.

```
SELECT ?r, ?v1, ?v2,. . .,?vk
FROM <udbl.it.uu.se:3050/company/>
WHERE
(?r, <c1>, ?v1) AND
(?r, <c2>, ?v2) AND
. . . .
(?r, <ck>, ?vk) AND
USING term FOR <udbl.it.uu.se/terms#>
```

Figure 17: Test query Q2.

We scale the number of joins by including more columns from the triple extent view. The test produces no join as we only use column identifier from table *t*. Scaling the number of joins of back-end tables does not make any difference for the performance of SWARD, as the same rewrite rules apply independently of how many underlying relational tables are involved.

The generated test query Q2 is executed with strategies ENO, ENOP and SEP. Figure 18 compares optimization time for ENO and ENOP where *t* has 9 columns.

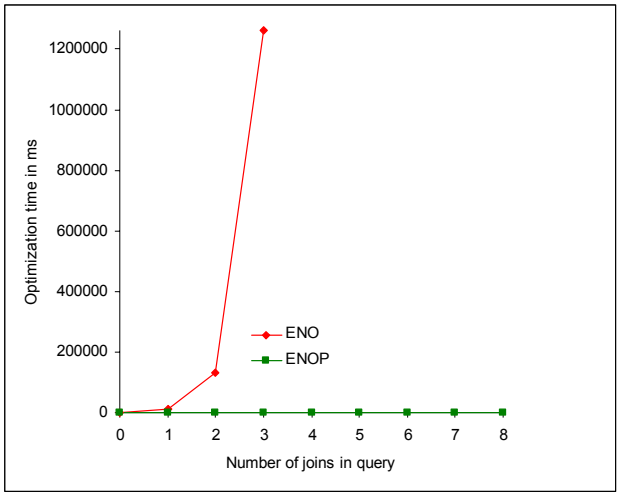

Figure 18: Comparing optimization times of ENO and ENOP.

As expected, the optimization time with ENO deteriorates very fast with the number of joins in the query. This is due to the complexity of the view expanded query sent to the normalizer for translation to disjunctive normal form. ENOP on the other hand scales much better. The reason is that the query is reduced by compile time evaluation of system predicates *uri*, *lit*, and *externalize* in steps *rewrite1* and *rewrite2*. This makes the query be on disjunctive normal form without any normalization. To conclude, Figure 18 clearly shows that the naïve processing strategy of ENO does not work for real-world queries of even modest complexity.

However, reducing the complexity of the expanded query obviously comes with a price in the form of some overhead for the partial evaluation. For strategy ENOP, partial evaluation is applied on the expanded triple extent view in Figure 11. The size of the triple extent view is proportional to the number of columns projected which can be in the hundreds in real-world databases.

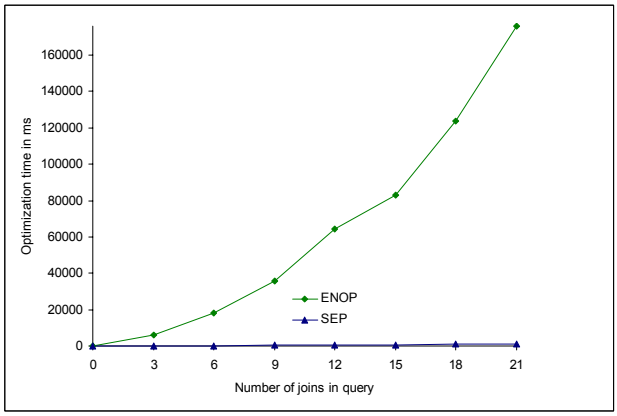

Figure 19: Comparing optimization times of ENOP and SEP.

In Figure 19 we see that SEP scales much better than ENOP when we increase the number of joins in the query. Our results show that SEP can handle large RDF queries.

6.2 Scaling the size of the database

We also measured the quality of the execution plans for our different strategies by executing the query Q3 in Figure 20 while scaling database according to the TPC-H benchmark. Q3 joins the *customer* and *orders* tables and select for each automobile order the clerk who registered the order.

```
SELECT ?clerk
FROM <udbl.it.uu.se:3050/company/>
WHERE
(?oid, <term:clerk>, ?clerk)                AND
(?oid,<term:o_custid>,custid)               AND
(?cid,<term:custid>,custid)                 AND
(?cid,<term:mktsegment>,'AUTOMOBILE')
USING term FOR <udbl.it.uu.se/terms#>
```

Figure 20: Test query Q3.

Figure 21 shows that the execution time with ENOP scales well as the database size grows. The query decomposer generates the optimal SQL statement to retrieve exactly the desired data. The optimal decomposed execution plans will call the following SQL statement:

```
select clerk
from orders o, customer c_
where o.custid = c.custid and
      mktsegment ='AUTOMOBILE'
```

Only one SQL statement is executed as the call to *externalize* (pre-condition) will return false for all subplans except the ones selected by the query. Thus the reduced query will be a single conjunction, as shown above. The decomposed execution plan will also contain post-processing to construct RDF objects from the result of the SQL query. SEP has identical results because the decomposed plan is the same.

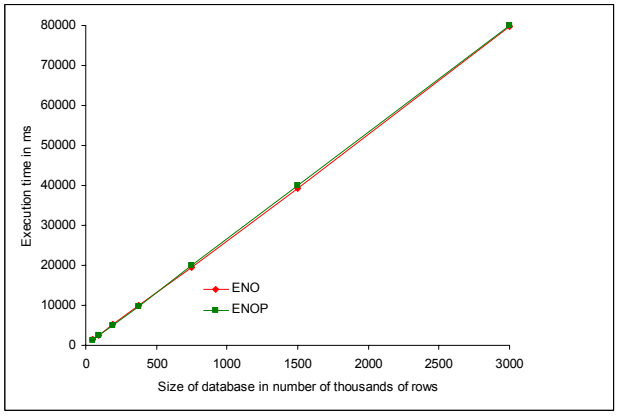

Figure 21: Execution time of Q3

Figure 21 also shows the scalability of the execution plan produced with ENO. Perhaps surprisingly, it scales as well. The reason is that, even though the produced plan in SWARD is very big, only a single SQL query is actually submitted to the database server, the same query as with ENOP or SEP. The big plan is a union of subplans constructed by the normalizer by combining the predicates in the query and the expanded triple extent views into one large union of conjunctive subqueries. Every subquery contains a precondition testing if the subquery applies. The precondition looks up the terminology table to see if the column identifiers for the subquery are those used in the original query. Only one such combination applies, i.e. the one testing exactly the column identifiers in the query. After decomposition it will produce the same SQL query as for ENOP and SEP. Therefore scalability of ENO and ENOP are identical; there is only a small extra overhead with ENO to test the pre-conditions.

In summary, our evaluations show that all our strategies produce scalable decomposed query plans. However, ENOP and in particular SEP radically improve query processing scalability compared to ENO and this is critical for RDF queries.

6.3 Total push-down, TPD

We also investigated whether the triple extent view and the terminology table can be stored as an SQL view entirely in the relational database. We use two tables:

```
t(k, c1, c2)
alias(external, internal)
```

The table *t* stores data. We scale its size by introducing more columns as in Figure 17, up to 10 columns. The table *alias* implements terminology mappings. We made a simplified experiment without any support for URI objects or RDF literals but regard all RDF resources as strings.

```
create view kv(s,p,v)
  as select 'co:k/'+t.k, internal, t.k
     from t, alias a
     where external = 'co#k'

create view c1v(s,p,v)
  as select 'col:c1/'+t.k, internal, t.c1
     from t, alias a
     where external = 'co#c1'

create view c2v(s,p,v)
  as select 'co:c2/ '+t.k, internal, t.c2
     from t, alias a
     where external = 'co#c2')

create view tev(s,p,v)
  as (select * from kv)  union all
     (select * from c1v) union all
     (select * from c2v) union all
     (select * from c3v)
```

Figure 22: Extent views in SQL

Figure 22 shows the three column extent views *c1v*, *c2v*, and *c3v* along with the triple extent view *tev*. We use string concatenation to simulate column and row identifier construction.

The test query expressed in SQL is shown in Figure 23.

```
select t1.v,t2.v,t3.v
from tev t1, tev t2, tev t3
  where t1.p = 'col_k' and
        t2.s = t1.s and t2.p = 'col#c1' and
        t3.s = t1.s and t3.p = 'col#c2'
```

Figure 23: SQL query to triple extent view

We scale the schema and the query by increasing the number of columns in *t*. For each new column we define new column extent views and modify *tev*. We also add to the test query new *tev* aliases and conditions for each added column. The produced execution plans were inspected using the query inspection tool of the DBMS.

It turned out that the DBMS compiled our test queries very fast. However, when the number of columns was larger than 10 the system was unable to execute the query as it required too many joins.

Manual inspections of the query execution plans revealed that no normalization at all had been made so the plans were joins of unions of joins of subplans for each

column extent view. Such an approach clearly does not scale well when the database size is increased.

To investigate scalability over the database size with TPD, we populated the table *t* with one to ten rows and measured the performance of the query in Figure 23. Figure 24 shows the result. For more than nine rows the DBMS gave an error message on lack of memory. Obviously, this is not a good strategy.

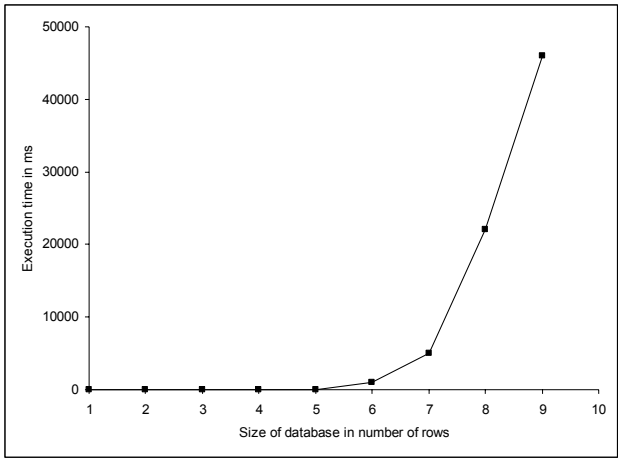

Figure 24: Scalability of total push-down (8 joins)

This experiment shows that the SWARD strategies provide substantially better query execution scalability than a not normalized execution plan on the database server.

7. Summary and conclusions

We have implemented a system named SWARD for scalable querying of RDF-based views of relational databases. We showed how to view the contents of a relational database as a large disjunctive *triple extent view* defined in SWARD using domain calculus. RDF queries expressed in RDQL are translated into domain calculus queries to the triple extent view. The triple extent view is defined as a union of *column extent views* that represent relational columns using domain calculus expressions.

We showed that the more traditional query optimization approach *ENO (expand – normalize – optimize)* based on view expansion followed by normalization and query decomposition does not scale when the size of the RDF query increases. However, the execution plans created by this approach scale well. The reason is that the normalization expands the original query into a disjunctive query where effective query decomposition can be applied on each subquery separately. As normalization is very expensive, the naïve approach is infeasible for real-world RDF queries. RDF queries often produce many self-joins over the triple extent view with many unions of column extent views.

The alternative not to normalize at all makes query optimization very fast. However, the unnormalized execution plans scale very badly for large databases because they perform many joins of unions of column extent views.

To improve the scalability of query processing we investigated two methods based on partial evaluation of queries to triple extent views. It is particularly important to partially evaluate accesses to the terminology mapping table between the ontology used in the RDF query and the automatically generated internal representation of RDF resources in SWARD.

The second method, *ENOP (Expand – Normalize – Partially Evaluate)*, extends ENO with two partial evaluation steps before and after view expansion. This reduces the query to a single conjunctive query after view expansion and no normalization is needed. The performance evaluation showed that this method is indeed superior to ENO when scaling the size of the query. The query execution time is the same.

The third method, SEP, avoids not only normalization but also the view expansion of the large disjunctive triple extent view. Each column extent view and its execution plans are pre-optimized and stored in a system table. The triple extent view is defined to retrieve and execute all pre-optimized column extent views in this system table. Partial evaluation of the original query determines all pre-optimized column extent views that are needed to answer the query. A special re-write expands exactly those column extent views that the partial evaluation produced. SEP creates a single conjunctive query as with ENOP. The difference is that SEP scales much better than ENOP with larger queries and larger triple extent views as the partial evaluation is applied on small expressions.

In this paper we discussed only database contents queries. We are currently investigating the processing of queries that combines schema and database contents data. Partial evaluation can play a critical role here as well, for evaluating meta-data at compile time.

Another interesting use of partial evaluation is for mediation of data from different sources [12]. Mediators are actually view definitions combining and reconciling data from different sources. By partially evaluating mediator meta-data perhaps more logic can be pushed into the sources.

It should also be investigated how to utilize RDF schema in query processing and partial evaluation.

Acknowledgements

This work was partially funded by the IST-STREP project *eGov-Bus, Advanced eGovernment Information Service Bus*, contract number 026727.

References

1. L.Beckman, A.Haraldson, O.Oskarsson, and E.Sandewall. A Partial Evaluator, and Its Use as a

Programming Tool. *Artificial Intelligence*, 7(4):319-357, 1976.
2. T.Berners-Lee, J.Hendler, and O.Lassila: The Semantic Web, *Scientific American*, May 2001.
3. C.Bizer, A.Seaborne: D2RQ -Treating Non-RDF Databases as Virtual RDF Graphs (Poster). *3rd International Semantic Web Conference (ISWC2004)*, 2004, Hiroshima, Japan.
4. D.Brickley and R.V.Guha: RDF Vocabulary Description Language 1.0: RDF-Schema, http://www.w3.org/TR/rdf-schema/, 2004.
5. J. Broekstra, A. Kampman and F. van Harmelen: Sesame: A generic Architecture for Storing and Querying RDF and RDF Schema. *Proc.1st International Semantic Web Conference (ISWC'02)*, Sardinia, Italy. 2002.
6. E.I.Chong, S.Das, G.Eadon and J,Srinivasan: An Efficient SQL-based RDF Querying Scheme, *Proc. 31st Intl. Conf. on Very Large Databases, VLDB2005,* pp1216-1227, Trondheim, Norway, 2005
7. J.Claußen, A.Kemper, G.Moerkotte, K.Peithner, and M.Steinbrunn: Optimization and Evaluation of Disjunctive Queries. *IEEE Trans. Knowl. Data Eng.* 12(2): 238-260 (2000)
8. C.Consel, L.Hornof, J.L.Lawall, R.Marlet, G.Muller, J.Noyé, S.Thibault, and E-N.Volanschi. "Tempo: Specializing Systems Applications and Beyond". *ACM Computing Surveys*, 30(3), September 1998.
9. V.Christophides, G.Karvounarakis, A.Magkanaraki, D.Plexousakis, and V.Tannen: The ICS-FORTH Semantic Web Integration Middleware (SWIM), *IEEE Data Engineering Bulletine*, 26(4), Dec. 2003.
10. Dublin Core Meta-data Initiative, Dublin Core Metadata Element Set, V 1.1, http://dublincore.org/documents/dces/
11. G. Fahl and T. Risch: Query Processing over Object Views of Relational Data, *The VLDB Journal*, 6(4): 261-281, 1997.
12. Y.Futamura: Partial evaluation of computation process – an approach to a compiler-compiler, *Systems, Computer, Controls*, 2(5):45.50, 1971
13. G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis and M. Scholl: RQL: A Declarative Query Language for RDF, *Proc. International World Wide Web Conference (WWW'02)*, Honolulu, Hawaii, USA. 2002.
14. W.Litwin, M.Ketabchi, R.Krishnamurthy: First Order Normal Form for Relational Databases and Multi Databases, *SIGMOD Records*, 20(4), December 1991.
15. A.Magkanaraki, V.Tannen, V.Christophides, and D.Plexousakis: Viewing the Semantic Web Through RVL Lenses, *2nd International Semantic Web Conference (ISWC'03)*, 2003, Sanibel Island, Florida, USA.
16. Open Directory Project, http://dmoz.org/
17. http://partial-eval.org/
18. X.Qian and L.Raschid: Query Interoperation Among Object-Oriented and Relational Databases, *Proc. International Conference on Data Engineering, ICDE 1995*, pp 271-278
19. RDF Gateway - a platform for the semantic web, Intellidimension, http://www.intellidimension.com/.
20. RDF Vocabulary Description Language 1.0: RDF-Schema, http://www.w3.org/TR/rdf-schema/, 2004.
21. RDF Site Summary 1.0, http://web.resource.org/rss/1.0/
22. RDQL - A Query Language for RDF, W3C Member Submission, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/, 2004.
23. Resource Description Framework (RDF): Concepts and Abstract Syntax, http://www.w3.org/TR/rdf-concepts/, 2004.
24. SeRQL, http://www.openrdf.org/doc/sesame/users/ch06.html
25. SPARQL Query Language for RDF, W3C Working Draft, 23 November 2005, http://www.w3.org/TR/rdf-sparql-query/
26. TPC-H benchmark, http://www.tpc.org/tpch/
27. K.Wilkinson, C.Sayers, H.A.Kuno, and D.Reynolds: Efficient RDF Storage and Retrieval in Jena 2, *Proc. VLDB Workshop on Semantic Web and Databases (SWDB'03)*, pp 131-150, September 2003.