Rewriting Queries on SPARQL Views

Wangchao Le[1] Songyun Duan[2] Anastasios Kementsietsidis[2] Feifei Li[1] Min Wang[3]

[1] Florida State University
{le, lifeifei}@cs.fsu.edu

[2] IBM T.J. Watson Research Center
{sduan, akement}@us.ibm.com

[3] HP Labs China
min.wang6@hp.com

ABSTRACT

The problem of answering SPARQL queries over virtual SPARQL views is commonly encountered in a number of settings, including while enforcing security policies to access RDF data, or when integrating RDF data from disparate sources. We approach this problem by rewriting SPARQL queries over the views to equivalent queries over the underlying RDF data, thus avoiding the costs entailed by view materialization and maintenance. We show that SPARQL query rewriting combines the most challenging aspects of rewriting for the relational and XML cases: like the relational case, SPARQL query rewriting requires synthesizing multiple views; like the XML case, the size of the rewritten query is exponential to the size of the query and the views. In this paper, we present the first *native* query rewriting algorithm for SPARQL. For an input SPARQL query over a set of virtual SPARQL views, the rewritten query resembles a union of conjunctive queries and can be of exponential size. We propose optimizations over the basic rewriting algorithm to (i) minimize each conjunctive query in the union; (ii) eliminate conjunctive queries with empty results from evaluation; and (iii) efficiently prune out big portions of the search space of empty rewritings. The experiments, performed on two RDF stores, show that our algorithms are scalable and independent of the underlying RDF stores. Furthermore, our optimizations have order of magnitude improvements over the basic rewriting algorithm in both the rewriting size and evaluation time.

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Database Management—Systems. Subject: Query processing

General Terms: Algorithms

Keywords: Rewriting, SPARQL query, SPARQL views

```
(person0, name, Eric)     (person0, lives, NYC)
(person1, name, Kenny)    (person1, lives, LA)     (person0, friend, person1)
(person2, name, Stan)     (person2, lives, NYC)    (person0, friend, person2)   (person0, related, person3)
(person3, name, Kyle)     (person3, lives, NYC)    (person1, friend, person2)   (person3, related, person9)
(person5, name, Jimmy)    (person5, lives, NYC)    (person1, friend, person5)   (person0, works, person4)
(person6, name, Timmy)    (person6, lives, CHI)    (person2, friend, person6)   (person2, works, person7)
(person9, name, Danny)    (person9, lives, LA)     (person3, friend, person8)
```

(a) Base triples

```
1) View V_F:                              3) View V_R:                               Q_U: SELECT {
CONSTRUCT {                                  CONSTRUCT {                             ?f5, ?r5, ?l5 }
① ?f0 vfriend ?f1,                        ① ?r0 vrelated ?r1,                        WHERE {
② ?f1 vname ?n1, ③ ?f1 vlives ?l1 }       ② ?r1 vname ?n1, ③ ?r1 vlives ?l1 }        ① person0 vfriend ?f5,
WHERE {                                      WHERE {                                 ② ?f5 vlives ?l5,
  ?f0 name ⟨P1⟩, ?f0 friend ?f1,               ?r0 name ⟨P3⟩, ?r0 related ?r1,       ③ person0 vrelated ?r5,
  ?f1 name ?n1, ?f1 lives ?l1 }                ?r1 name ?n1, ?r1 lives ?l1 }         ④ ?r5 vlives ?l5 }

2) View V_FoF:                            4) View V_RoR:
CONSTRUCT {                                  CONSTRUCT {
① ?f2 vfriend ?f4,                        ① ?r2 vrelated ?r4,
②?f4 vname ?n4, ③ ?f4 vlives ?l4 }        ② ?r4 vname ?n4, ③ ?r4 vlives ?l4 }
WHERE {                                      WHERE {
  ?f2 name ⟨P2⟩,                               ?r2 name ⟨P4⟩,
  ?f2 friend ?f3, ?f3 friend ?f4,              ?r2 related ?r3, ?r3 related ?r4,
  ?f4 name ?n4, ?f4 lives ?l4 }                ?r4 name ?n4, ?r4 lives ?l4 }
```

(b) Views and user query

```
(person1, vname, Kenny) [V_F]          (person1, vlives, LA) [V_F]             (person0, vfriend, person1) [V_F]
(person2, vname, Stan) [V_F, V_FoF]    (person2, vlives, NYC) [V_F, V_FoF]     (person0, vfriend, person2) [V_F, V_FoF]
(person3, vname, Kyle) [V_R]           (person3, vlives, NYC) [V_R]            (person0, vrelated, person3) [V_R]
(person5, vname, Jimmy) [V_FoF]        (person5, vlives, NYC) [V_FoF]          (person0, vfriend, person5) [V_FoF]
(person6, vname, Timmy) [V_FoF]        (person6, vlives, CHI) [V_FoF]          (person0, vfriend, person6) [V_FoF]
(person9, vname, Danny) [V_RoR]        (person9, vlives, LA) [V_RoR]           (person0, vrelated, person9) [V_RoR]
```

(c) Materialized triples in V_{Eric}

Figure 1: Motivating example

1. INTRODUCTION

In a number of settings, including access control [13, 14, 24, 26] or data integration [19, 25], users can only access data that are *visible* through a set of views. The views are typically defined using a standard query language (SQL for relational data, XPath/XQuery for XML, SPARQL for RDF) and commonly the same language is used by the users to express the queries over the views. The process of answering these user queries is determined on whether the views are *virtual* or *materialized*. For materialized views, evaluating the user queries is straightforward, but the simplicity in query evaluation comes at a cost, both in terms of the space required to save the views, and in terms of the time needed to maintain the views. Therefore, view materialization is a viable alternative only when (i) there are a small number of views; (ii) the views expose small fragments of base data; and (iii) the base data are infrequently updated. Since most practical scenarios do not meet these requirements, the other alternative is to use virtual view and *rewrite* the queries over the views to equivalent queries over the underlying data. In relational databases, query rewriting over SQL views is *straightforward* as it only requires *view expansion*, *i.e.*, the view mentioned in the user SQL query is replaced by its definition. However, in the case of RDF and SPARQL, view expansion is not possible since expansion requires query nesting, a feature not currently supported by SPARQL. In XML, XPath query rewriting is rather involved and the rewriting is exponential to the size of the query and the view [14]. Query rewriting for RDF/SPARQL is inherently more complex since (i) whereas XML/XPath is used for representing and querying trees, RDF/SPARQL considers generic graphs; and (ii) in SPARQL, the query and view definitions may use different variables to refer to the same entity, thus requiring *variable mappings* when synthesizing multiple views to rewrite a given query. Therefore, query rewriting in RDF/SPARQL raises distinct challenges from those in the relational or XML.

To illustrate these challenges, we use a Facebook-inspired example, and in Figure 1(a) we consider RDF triples modeling common *acquaintances* (e.g., friend, related, and works). In such a setting, we can use views to express access control (privacy) policies over Facebook profiles. For instance, for each person (*e.g.,* person0 with name "Eric") we might have a default policy that exposes from the social network only the person's immediate friends (*e.g.,* for person0, person1 and person2), and relatives (*e.g.,* for person0, person3), along with friends-of-friends (FoF), and relatives-of-relatives (RoR), while not exposing the relatives-of-friends, or the friends-of-relatives. Figure 1(b) shows four views to enforce this policy (variables are prefixed by '?' and constructed view predicates are prefixed with the letter 'v'). The views hide any distinction between immediate friends (or relatives) and those at a distance of two. Like [24], a parameter $\langle P_i \rangle$ specifies the name of the person for whom the policies are enforced. Figure 1(c) shows the result V_{Eric} of materializing all four views for "Eric", with each triple annotated by the generating view(s).

Consider the query Q_U in Figure 1(b) over the triples for "Eric" (shown in Figure 1(c)). Q_U identifies "Eric"'s friends and relatives who live in the same city. Instead of materializing V_{Eric} just to evaluate Q_U, we would like to use the views to rewrite Q_U into a query over the base data in Figure 1(a). *The first challenge is to determine which views can be used in this rewriting.* Finding relevant views requires computing *(variable) mappings* between the body of Q_U (its WHERE clause) and the return values (CONSTRUCT clause) of the views. An example of a mapping between triples (?f_0, vfriend, ?f_1) in V_F and (person0, vfriend, ?f_5) in Q_U, maps ?f_0 to person0 and ?f_1 to ?f_5. The mapping indicates that V_F *can be used* for rewriting Q_U. *How* it will be used, is our next challenge.

In more detail, *the second challenge of the query rewriting is to determine how the views can be combined into a sound and complete rewriting.* Intuitively, soundness guarantees that the rewritten query only returns results that would have been retrieved should the user query have been executed over the materialized view. Completeness guarantees that the rewritten query returns all these results. Addressing the second challenge requires algorithms that (i) meaningfully combine the views identified in the first step of the rewriting; and (ii) consider all such possible combinations of the views. In our example, a sound and complete rewriting results in a union of *64 queries*, with each query being a result of a single view combination, and where each view combination results in by combining 2 possible var. mappings for each of vfriend and vrelated, and 4 possible var. mappings for each instance of vlives. Clearly, there is an (exponential) blowup in the size of the rewritten query, with respect to the size of the input query and views. However, the *blind* view combinations often generate rewritings that have empty results, which provides optimization opportunities by removing the empty rewritings from evaluation. For this particular example, *only four of these combinations* need to be evaluated (the others are either subsumed by these four, or return no results). Therefore, *our third challenge is to optimize the rewriting and evaluate only a subset of the view combinations without sacrificing soundness or completeness.*

Given that relational algebra (and the corresponding SQL fragment) has the same expressive power as SPARQL [8], one

```
VF-SQL:
SELECT F.s, F.o, N'.s, N'.o, L.s, L.o
FROM name N, friend F, name N', lives L
WHERE N.s =F.s AND N.o =⟨P1⟩ AND
      N'.s =F.o AND L.s =F.o
```

```
VR-SQL:
SELECT R.s, R.o, N'.s, N'.o, L.s, L.o
FROM name N, related R, name N', lives L
WHERE N.s =R.s AND N.o =⟨P3⟩ AND
      N'.s =R.o AND L.s =R.o
```

```
VFoF-SQL:
SELECT F.s, F'.o, N'.s, N'.o, L.s, L.o
FROM name N, friend F, friend F', name N', lives L
WHERE N.s =F.s AND N.o =⟨P2⟩ AND
      F.o =F'.s AND N'.s =F'.o AND L.s =F'.o
```

```
VRoR-SQL:
SELECT R.s, R'.o, N'.s, N'.o, L.s, L.o
FROM name N, related R, related R', name N', lives L
WHERE N.s =R.s AND N.o =⟨P4⟩ AND
      R.o =R'.s AND N'.s =R'.o AND L.s =R'.o
```

(a) SQL translation of V_F, V_R, V_{FoF} and V_{RoR}.

```
vfriend:
SELECT fs, fo FROM VF-SQL
UNION
SELECT fs, fo FROM VFoF-SQL

vrelated:
SELECT rs, ro FROM VR-SQL
UNION
SELECT rs, ro FROM VRoR-SQL
```

```
vlives:
SELECT ls, lo FROM VF-SQL
UNION
SELECT ls, lo FROM VFoF-SQL
UNION
SELECT ls, lo FROM VR-SQL
UNION
SELECT ls, lo FROM VRoR-SQL
```

(b) Secure predicate tables definitions

```
QU-SQL:
SELECT F.o, R.o, L.o
FROM vfriend F, vlives L, vlives L', vrelated R
WHERE F.s =person0 AND F.o =L.s AND R.s =person0 AND R.o =L'.s AND L.o =L'.o
```

(c) SQL translation of query Q_U.

Figure 2: Attempting a relational/SQL rewriting

might be tempted to address the SPARQL rewriting problem by considering the corresponding SQL setting and applying the solutions in SQL. Although this seems promising since some RDF stores do use a relational back-end (*e.g.,* Jena SDB [2], Virtuoso [3], C-store [4], *etc*), we show here that for a number of reasons such an approach does *not* reduce the complexity. To translate our setting to the relational case, we use one of the most efficient relational storage strategies for RDF, namely, predicate tables [4] (column-store style storage); our observations are *independent* of this choice. So, we have a database with five tables: name (s, o), lives (s, o), friend (s, o), related (s, o), and works (s, o), whose contents are easily inferred by the corresponding triples in Figure 1(a). In Figures 2(a) and (c), we show the SQL translations of the views and query of Figure 1(b). During this translation, we need to create the corresponding *view* predicate tables of the base database tables. So, as shown in Figure 2(b), we need to create the vfriend table which contains the friend subjects and objects returned by the $V_{F\text{-}SQL}$ and $V_{FoF\text{-}SQL}$ views (similarly for vrelated and vlives). How can we rewrite $Q_{U\text{-}SQL}$ to a query over the base five tables? Since view expansion is supported in SQL, we can replace in $Q_{U\text{-}SQL}$ the vfriend, vrelated, and vlives tables with their definitions in Figure 2(b), and in turn replace $V_{F\text{-}SQL}$, $V_{FoF\text{-}SQL}$, $V_{R\text{-}SQL}$ and $V_{RoR\text{-}SQL}$ with their definitions in Figure 2(a). Finally, it is not hard to see that the rewriting of $Q_{U\text{-}SQL}$ results in a union of 64 queries, the same blow-up in size, as the one observed in SPARQL. So, moving from SPARQL to SQL does not reduce the complexity of the problem (more exposition in Section 5); we will validate this observation in Section 4. Such move is also prohibitive as there is an increasing number of stores (*e.g.,* Jena TDB [2], 4store [1]) using native RDF storage. For these stores, translation to SQL does not work. Therefore, it is necessary to have a *native* and *efficient* SPARQL rewriting algorithm, which has the advantage of being *generic* since it works on any existing RDF store irrespectively of the storage model used.

Summary of our contributions:

1. We study the rewriting of SPARQL queries over virtual SPARQL views, and propose a *native* SPARQL rewriting algorithm (Section 2), and prove that it generates *sound* and *complete* rewritings.

2. We propose several optimizations of the basic rewriting algorithm to reduce the complexity (Section 3.1) and size of the rewritten queries (Sections 3.2 and 3.3), while employing novel optimization techniques customized for our needs.

3. We present extensive experiments on two RDF stores (Section 4) on the scalability and portability of our algorithms. The optimizations result in order of magnitude improvements in rewritten query sizes and evaluation times over our basic rewriting algorithm in SPARQL; the latter is comparable to applying rewriting techniques in SQL after translating SPARQL queries into SQL queries.

We survey the related work in Section 5 and conclude the paper in Section 6.

2. QUERY REWRITING IN SPARQL

SPARQL, a W3C recommendation, is a pattern-matching query language. The most common SPARQL queries have the following form: Q := (SELECT | CONSTRUCT) RD (WHERE GP), where GP are *triple patterns*, *i.e.,* triples involving variables and/or constants, and RD is the *result description.* Given an RDF graph G, a triple pattern on G searches for a set of subgraphs of G, each of which *matches* the pattern (by binding pattern variables to values in the subgraph). For SELECT queries, RD is a subset of variables in the graph pattern, similar to a projection in SQL. This is the case for query $\mathsf{Q_U}$ in Figure 1(b). For CONSTRUCT queries, RD is a set of triple templates that construct a *new* RDF graph by replacing variables in GP with matched values. This is the case for the views in Figure 1(b). Finally, we consider boolean SPARQL queries of the form ASK GP which indicate whether GP exists, or not, in G. Similar to SQL where research considered set before bag semantics, for our non-boolean SPARQL queries we assume set semantics whose importance for SPARQL has already been noted [22].

The central technical problem in this paper is the *rewriting problem* as follows: given a set of views $\mathcal{V} = \{\mathsf{V}_1, \mathsf{V}_2, \ldots, \mathsf{V}_l\}$ over an RDF graph G, and a SPARQL query Q over the vocabulary of the views, compute a SPARQL query Q′ over G such that $\mathsf{Q}'(\mathsf{G}) = \mathsf{Q}(\mathcal{V}(\mathsf{G}))$. Like [26], we consider two criteria on the correctness of a rewriting, namely, *soundness* and *completeness.*

1. The rewriting is sound iff $\mathsf{Q}'(\mathsf{G})$ is contained in $\mathsf{Q}(\mathcal{V}(\mathsf{G}))$, *i.e.,* $\mathsf{Q}'(\mathsf{G}) \subseteq \mathsf{Q}(\mathcal{V}(\mathsf{G}))$
2. The rewriting is complete iff $\mathsf{Q}(\mathcal{V}(\mathsf{G}))$ is contained in $\mathsf{Q}'(\mathsf{G})$, *i.e.,* $\mathsf{Q}(\mathcal{V}(\mathsf{G})) \subseteq \mathsf{Q}'(\mathsf{G})$

Soundness and completeness suffice to show that $\mathsf{Q}(\mathcal{V}(\mathsf{G})) = \mathsf{Q}'(\mathsf{G})$. We will prove our rewriting meet the two criteria.

2.1 Rewriting Algorithm

The first challenge in query rewriting (as mentioned in the introduction) is to determine which views can be used for the rewriting. In SPARQL, the crucial observation to address this challenge is that *if a variable mapping exists between a triple pattern* ($\mathsf{s_v}$, $\mathsf{p_v}$, $\mathsf{o_v}$) *in the result description* $\mathsf{RD}(\mathsf{V}_j)$ *of a view* V_j *and one of the triple patterns* ($\mathsf{s_q}$, $\mathsf{p_q}$, $\mathsf{o_q}$) *in the graph pattern* GP(Q) *of query* Q*, then view* V_j *can be used to rewrite* Q . Using this observation, we present Algorithm 1 (SQR) to perform the rewriting in two steps. In the first step (lines 3-18), the algorithm determines, for each triple pattern $p_i(\bar{X}_i)$ in user query, the set CandV_i of *candidate views* that have a variable mapping to this triple pattern. For ease of presentation, we assume that in our SPARQL queries the predicate in each triple pattern is a constant (the subject and object can either be variables or constants). Even if a triple has a variable in its predicate, we can simply substitute such a triple by a set of triple patterns, each triple in the set binding the predicate variable to a constant predicate from the active domain of predicates in the RDF store.

Computing variable mappings between triple patterns in SQR is similar to computing *substitutions* between conjunctive queries [6]. Formally, a substitution is a mapping between the corresponding elements (subject, predicate, and object) in a pair of triples that maps: (i) a variable in the first triple to another variable or constant in the second triple; or (ii) a constant in the first triple to the same constant in the second triple. Or, conversely, a substitution cannot map a constant in the first triple to a variable in the second, or map two different constants in the triples. For example, a substitution exists from ($?\mathsf{f}_0$, vfriend, $?\mathsf{f}_1$) to (person0, vfriend, $?\mathsf{f}_5$), which maps the variable $?\mathsf{f}_0$ to the constant person0 and the variable $?\mathsf{f}_1$ to the variable $?\mathsf{f}_5$. There is no substitution from the second to the first triple since we cannot map the constant person0 to the variable $?\mathsf{f}_0$.

```
1  Input: Views $\mathcal{V}$, query Q with $\mathsf{GP(Q)}=(\mathsf{s}_1^\mathsf{Q}, \mathsf{p}_1^\mathsf{Q}, \mathsf{o}_1^\mathsf{Q}), \ldots, (\mathsf{s}_n^\mathsf{Q}, \mathsf{p}_n^\mathsf{Q}, \mathsf{o}_n^\mathsf{Q})$
2  Output: a rewriting Q′ as a union of conjunctive queries
3  for each $(\mathsf{s}_i^\mathsf{Q}, \mathsf{p}_i^\mathsf{Q}, \mathsf{o}_i^\mathsf{Q})$, $1 \le i \le n$ do
4      Set $\mathsf{CandV}_i$ to $\emptyset$.
5      for each view $\mathsf{V}_j \in \mathcal{V}$ do
6          Let $\mathsf{RD}(\mathsf{V}_j)=(\mathsf{s}_1^{\mathsf{V_j}}, \mathsf{p}_1^{\mathsf{V_j}}, \mathsf{o}_1^{\mathsf{V_j}}), \ldots, (\mathsf{s}_m^{\mathsf{V_j}}, \mathsf{p}_m^{\mathsf{V_j}}, \mathsf{o}_m^{\mathsf{V_j}})$
7          for each $(\mathsf{s}_k^{\mathsf{V_j}}, \mathsf{p}_k^{\mathsf{V_j}}, \mathsf{o}_k^{\mathsf{V_j}})$, $1 \le k \le m$ do
8              if $\mathsf{p}_i^\mathsf{Q} = \mathsf{p}_k^{\mathsf{V_j}}$ then
9                  Set variable mapping $\Phi_{ijk}$ to undefined
10                 for the pair $(\mathsf{s}_i^\mathsf{Q}, \mathsf{s}_k^{\mathsf{V_j}})$ of subjects (similarly objects $(\mathsf{o}_i^\mathsf{Q}, \mathsf{o}_k^{\mathsf{V_j}})$) do
11                     if var. mapping $\phi : \mathsf{s}_i^\mathsf{Q} \to \mathsf{s}_k^{\mathsf{V_j}}$ exists then
12                         if $\phi$ maps two variables then $\Phi_{ijk}(\mathsf{s}_k^{\mathsf{V_j}}) = s_i^\mathsf{Q}$
13                         else $\Phi_{ijk}(\mathsf{s}_k^{\mathsf{V_j}}) = \mathsf{s}_k^{\mathsf{V_j}}$ ($\mathsf{s}_k^{\mathsf{V_j}}$ is a constant)
14                     if var. mapping $\phi : \mathsf{s}_k^{\mathsf{V_j}} \to \mathsf{s}_i^\mathsf{Q}$ exists then
15                         if $\phi$ maps a variable to a constant then
                           $\Phi_{ijk}(\mathsf{s}_k^{\mathsf{V_j}}) = \mathsf{s}_i^\mathsf{Q}$
16                     if $\Phi_{ijk}$ is defined then
17                         For any variable $v'$ in $\mathsf{RD}(\mathsf{V}_j)$ not in $(\mathsf{s}_k^{\mathsf{V_j}}, \mathsf{p}_k^{\mathsf{V_j}}, \mathsf{o}_k^{\mathsf{V_j}})$,
                           $\Phi_{ijk}$ maps $v'$ to a fresh variable (a new variable)
18                         Add $(\mathsf{V}_j, \Phi_{ijk})$ to $\mathsf{CandV}_i$
19 Set the query rewriting result Q′ to $\emptyset$
20 for each entry in Cartesian product $\mathsf{CandV}_1 \times \ldots \times \mathsf{CandV}_n$ do
21     if $\Phi_{1j_1k_1}, \Phi_{2j_2k_2}, \ldots, \Phi_{nj_nk_n}$ is compatible then
22         $\mathsf{RD}(q') = \mathsf{RD(Q)}$
23         $\mathsf{GP}(q') = \mathsf{GP}(\Phi_{1j_1k_1}(\mathsf{V}_{j_1}), \ldots, \Phi_{nj_nk_n}(\mathsf{V}_{j_n}))$
24         $\mathsf{Q}' = \mathsf{Q}' \cup q'$
25 return Q′
```
Algorithm 1: SPARQL Query Rewriting (SQR) Algorithm

Unlike substitutions that are *directional*, *i.e.,* the mapping is always from one triple to another, the variable mappings computed here are more complex; since for their creation we need to compose the (partial) substitutions that exist between the two triples in *both* directions. As an example, consider the triples (person0, vfriend, $?\mathsf{f}_5$) and ($?\mathsf{f}_6$, vfriend, person1). There is no substitution between the two triples in either of the directions. However, the variable mappings used by our algorithm attempt to compute partial substitutions between the two triples and use those to compute a variable mapping. In our example, our algorithm computes a partial substitution from the first triple to the second that maps $?\mathsf{f}_5$ to constant person1. It also computes a partial substitution from the second triple to the first that maps $?\mathsf{f}_6$ to constant

$(\mathrm{V_F}, \Phi_{111})$: Φ_{111}(?f$_0$, ?f$_1$, ?n$_1$, ?l$_1$)= (person0, ?f$_5$, ?ν_0, ?ν_1)
$(\mathrm{V_{FoF}}, \Phi_{121})$: Φ_{121}(?f$_2$, ?f$_4$, ?n$_4$, ?l$_4$)= (person0, ?f$_5$, ?ν_2, ?ν_3)

(a) CandV$_1$ for triple (person0, vfriend, ?f$_5$)

$(\mathrm{V_F}, \Phi_{213})$: Φ_{213}((?f$_1$,?l$_1$,?f$_0$,?n$_1$))=(?f$_5$,?l$_5$,?ν_4,?ν_5)
$(\mathrm{V_{FoF}}, \Phi_{223})$: Φ_{223}((?f$_4$,?l$_4$,?f$_2$,?n$_4$))=(?f$_5$,?l$_5$,?ν_6,?ν_7)
$(\mathrm{V_R}, \Phi_{233})$: Φ_{233}((?r$_1$,?l$_1$,?r$_0$,?n$_1$))=(?f$_5$,?l$_5$,?ν_8,?ν_9)
$(\mathrm{V_{RoR}}, \Phi_{243})$: Φ_{243}((?r$_4$,?l$_4$,?r$_2$,?n$_4$))=(?f$_5$,?l$_5$,?ν_{10},?ν_{11})

(b) CandV$_2$ for triple (?f$_5$, vlives, ?l$_5$)

GP(q')={ person0 name $\langle P\rangle$, person0 friend ?f$'_3$, ?f$'_3$ friend ?f$_5$, ?f$_5$ name ?ν_2, ?f$_5$ lives ?ν_3, ?ν_8 name $\langle P\rangle$, ?ν_8 related ?f$_5$, ?f$_5$ name ?ν_9, ?f$_5$ lives ?l$_5$ }

(c) Rewritten body of $\mathsf{Q_U}^{\mathrm{part}}$

Table 1: Variable mapping example

person0. The combination of the two partial substitutions constitutes a variable mapping. Eventually, this is used to compute a new triple of the form (person0, friend, person1). The computed triple is such that a substitution exists from each of the initial triples to it.

After the var. mapping computation, Algorithm SQR (lines 19-23) constructs in its second step the rewriting as a union of conjunctive queries. Each query in the union is generated by considering one combination from the Cartesian product of the sets CandV$_i$ ($i \in [1, n]$). While considering each combination, we need to make sure that the corresponding variable mappings from individual predicates are *compatible*, *i.e.,* they do not map one variable in the query Q to two different constants (from the views). For the variables only appearing in GP of the views, they are mapped to fresh (*i.e.,* new) variables by default. For each compatible combination, we generate one query in the union.

To illustrate this, consider triples $\mathsf{t}_1^{\mathsf{QU}}$ = (person0, vfriend, ?f$_5$) and $\mathsf{t}_2^{\mathsf{QU}}$ = (?f$_5$, vlives, ?l$_5$), from Q$_\mathsf{U}$ of Figure 1. For $\mathsf{t}_1^{\mathsf{QU}}$, CandV$_1$ = $\{(\mathrm{V_F}, \Phi_{111}), (\mathrm{V_{FoF}}, \Phi_{121})\}$, where both Φ_{111} and Φ_{121} are shown in Table 1(a) (the subscripts of Φs are defined in Algorithm SQR and labelled in Figure 1(b)). Similarly, Table 1(b) shows CandV$_2$ for $\mathsf{t}_2^{\mathsf{QU}}$. To get Φ_{111}, SQR first considers $\mathsf{t}_1^{\mathsf{QU}}$ with $\mathsf{t}_1^{\mathsf{V_F}}$ = (?f$_0$, vfriend, ?f$_1$) from $\mathrm{V_F}$ (lines 3-8). Then, for the pair of subjects (person0, ?f$_0$) (line 10), a var. mapping ϕ exists (line 14) from ?f$_0$ to person0. Therefore Φ_{111} assigns the constant to the variable (line 15). Next, the pair of objects (?f$_5$, ?f$_1$) is considered (line 10) and as a result Φ_{111} assigns ?f$_1$ to ?f$_5$ (lines 11-12). The remaining variables (?n$_1$ and ?l$_1$) in $\mathrm{V_F}$ are assigned to fresh/new variables respectively (?v$_0$ and ?v$_1$) (line 17). This concludes the computation of Φ_{111}. Other Φ's are computed accordingly. To illustrate, we consider the (partial) query $\mathsf{Q_U}^{\mathrm{part}}$ of Q$_\mathsf{U}$ consisting only of triples $\mathsf{t}_1^{\mathsf{QU}}$ and $\mathsf{t}_2^{\mathsf{QU}}$. Then, there are 8 rewritings of $\mathsf{Q_U}^{\mathrm{part}}$ (lines 20-24), one for each combination of Φ's in CandV$_1$ and CandV$_2$. Table 1(c) shows the rewriting for $\mathsf{Q_U}^{\mathrm{part}}$, using $(\mathrm{V_{FoF}}, \Phi_{121})$ in CandV$_1$ and $(\mathrm{V_R}, \Phi_{233})$ in CandV$_2$.

Theorem 1. *The rewriting* Q$'$ *of* SQR *is sound and complete (see proof in [18]).*

The cost of Algorithm SQR is influenced by the cost of computing variable mappings $\mathcal{O}(|\mathsf{Q}| \times \sum_j |\mathsf{RD}(\mathsf{V}_j)|)$, but is dominated by the generation of rewritings and is thus equal to $\mathcal{O}((\sum_j |\mathsf{V}_j|)^{|\mathsf{Q}|})$, where $|\mathsf{Q}|$ (resp. $|\mathsf{V}_j|$) is the size of Q (resp. V$_j$).

In SQR, as long as a view predicate is mentioned in a query, the view automatically becomes a candidate for rewriting the query (modulo an incompatibility check). The key reason is that the RDF model is, in a sense, *schema-less*. This schema-less nature of the data model is the main reason behind the exponential blow-up of the rewriting. As an example, using SQR to rewrite query Q$_\mathsf{U}$ over the views of Figure 1 results in a rewriting Q$'$ that is a union of 64 queries; all of which must be evaluated in principle for the rewriting to be sound and complete. However, a number of these queries can either be (i) optimized and replaced by more *succinct* and equivalent queries; or (ii) dropped from consideration altogether because they result in an empty set. Going back to our motivating example, remember that actually only 4 queries suffice for the rewriting. Therefore, the challenge we address next is to perform such optimizations without sacrificing soundness or completeness.

3. OPTIMIZING REWRITINGS

3.1 Optimizing Individual Rewritings

In the rewriting of Q$_\mathsf{U}$, each rewriting q' generated by Algorithm SQR joins four views (one view from the CandV of each of the four predicates vfriend, vlives, vrelated, lives in Q$_\mathsf{U}$). One such rewriting involves views V$_\mathsf{F}$ for vfriend, V$_\mathsf{F}$ for vlives, V$_\mathsf{R}$ for vrelated, and V$_\mathsf{R}$ for vlives. That is, the rewriting uses two copies of both V$_\mathsf{F}$ and V$_\mathsf{R}$. Since the join (*e.g.,* vfriend joined with vlives) in Q$_\mathsf{U}$ is done in a similar way as that in the view (correspondingly, V$_\mathsf{F}$), there is redundancy to have two copies of V$_\mathsf{F}$ for this join; the similar situation happens to V$_\mathsf{R}$. The question is whether it is possible to get an *equivalent* rewriting by merging view copies, and thus generate a simpler query to evaluate. Indeed, one copy from each view suffices: the two copies of V$_\mathsf{F}$ are due to predicates vfriend and vlives being joined on variable ?f$_5$ in Q$_\mathsf{U}$. But in the CONSTRUCT of V$_\mathsf{F}$ these two predicates are joined in a similar way. Therefore, one copy of V$_\mathsf{F}$ suffices since it already returns all the triples joinable by the two predicates (*i.e.,* the view self-join is *equivalent* to the view itself).

```
1  Inputs: (V, $\Phi_1$) from CandV$_1$, (V, $\Phi_2$) from CandV$_2$
2  Output: (V, $\Phi_{\mathrm{merge}}$)
3  Continue_merge = $false$;
4  for each triple pattern (s, p, o) in $\Phi_1$(RD(V)) do
5      Let (s', p, o') be the corresponding pattern in $\Phi_2$(RD(V))
6      if $\{s, o\} \cap \{s', o'\} \neq \emptyset$ then Continue_merge = $true$;
7  if Continue_merge == $false$ then return (V, $\emptyset$);
8  for each triple pattern (s, p, o) in $\Phi_1$(RD(V)) do
9      Let (s', p, o') be the corresponding pattern in $\Phi_2$(RD(V))
10     Create corresponding merged pattern (s$_M$, p, o$_M$) for $\Phi_{\mathrm{merge}}$
11     if s is a fresh variable then s$_M$ = s'; goto 14;
12     if s' is a fresh variable then s$_M$ = s; goto 14;
13     if s = s' then s$_M$ = s else return (V, $\emptyset$)
14     if o is a fresh variable then o$_M$ = o'; goto 8;
15     if o' is a fresh variable then o$_M$ = o; goto 8;
16     if o = o' then o$_M$ = o else return (V, $\emptyset$)
17 return (V, $\Phi_{merge}$)
```

Algorithm 2: Candidate View Merging

Algorithm 2 detects such situations by accepting as input two copies of a view V that are used in rewriting a query, one as the candidate view for predicate p_1 and the other for its joinable predicate p_2, with variable mappings Φ_1 and Φ_2, respectively. The algorithm considers the variable mappings between the query and the views and attempts to construct a new mapping Φ_{merge} that merges the two input mappings. If Φ_{merge} exists, the two copies of V can be merged to simplify the rewriting. During merging, should multiple occurrences of the same predicate appear in the same V, they are treated as distinct predicates. A key observation during the construction of Φ_{merge} is that *all the variables and constants appearing in the query are treated as constants* (thus only fresh variables are treated as variables for the purpose of merging the view copies). This ensures that views are merged not only because they are copies of the same view, but also because their predicates are joined in precisely the same way as in the query (lines 4-7). Each time view copies are merged, we must also account for any variable mappings that have been applied to the views, due to their relationships with the views used for rewriting other predicates. Algorithm 2 ensures that the effects of such variable mappings are also merged (lines 8-16). If Φ_{merge} in the output of Algorithm 2 is $\emptyset$, the two copies of V can not be merged.

To illustrate, consider in the rewriting of $\mathsf{Q_U}$ the var. mapping $(\mathsf{V_F}, \Phi_{111})$ for predicate vfriend and $(\mathsf{V_F}, \Phi_{213})$ for predicate vlives. Applying the two mapping functions respectively on $\mathsf{V_F}$ would result in two copies of $\mathsf{V_F}$ joined on $?\mathsf{f_5}$. Since in $\mathsf{V_F}$ the triple patterns of vfriend and vlives are joined in the same way as that in $\mathsf{Q_U}$, Φ_{111} and Φ_{213} can be merged; $\Phi_{\text{merge}}(?\nu_4, ?\mathsf{f_5}, ?\nu_0, ?\nu_1) = (\text{person0}, ?\mathsf{f_5}, ?\nu_5, ?\mathsf{l_5})$. Therefore, the rewriting from Algorithm SQR involving two copies of $\mathsf{V_F}$ can be simplified into a rewriting with one copy.

Theorem 2. *Query q'_{merge} resulting from (i) replacing the two copies of view V in query q' with one; and (ii) applying Φ_{merge} computed by Algorithm 2, in place of Φ_1 and Φ_2; is equivalent to q' (see proof in [18]).*

The cost of Algorithm 2 is $\mathcal{O}(|\mathsf{V}|)$. Since, in the worst case, there can be as many view copies of a view V as the size of the query, optimizing with Algorithm 2 each conjunctive query generated at lines 22-23 of SQR costs $\mathcal{O}(|\mathsf{Q}| \times |\mathsf{V}|)$.

3.2 Pruning Rewritings with Empty Results

Due to the schema-less nature of RDF, a sound and complete rewriting of an input query *requires* that we construct rewritings by considering every possible combination of predicates from the input views, which often results in a certain number of rewritings with empty results. (This observation is unique to RDF/SPARQL, in comparison to the query rewriting results in relational or XML case.) For example, a sound and complete rewriting of query $\mathsf{Q_U}^{\text{part}}$ (see Section 2.1) includes the rewriting q' in Table 1(c). Rewriting q' joins triples from $\mathsf{V_{FoF}}$ and $\mathsf{V_R}$ and essentially looks for persons that are relatives of friends-of-friends of person0. Looking at the triples in Figure 1, it is clear that no current base triples satisfy the constraints of q'. The question is then how can we detect such empty rewritings, and more importantly, how to do this in a light-weight fashion.

Consider a simple case where a rewriting involves a join between two predicates $(?\mathsf{y_1}, \mathsf{p_1}, ?\mathsf{y_2})$ and $(?\mathsf{y_3}, \mathsf{p_2}, ?\mathsf{y_4})$, where the join equates $?\mathsf{y_2}$ and $?\mathsf{y_3}$. Denote the value set of a variable $?\mathsf{x}$ as $A(?\mathsf{x})$. If we store $A(?\mathsf{x})$ for every variable in any triple pattern, this problem is trivial, *i.e.,* we simply check whether $A(?\mathsf{y_2}) \bigcap A(?\mathsf{y_3}) = \emptyset$. Unfortunately, this straightforward solution is expensive space-wise. In general, a negative result exists for the *boolean set intersection* problem, *i.e.,* given two sets A_1 and A_2, checking if A_1 and A_2 intersects requires linear space, even if one is willing to settle to a constant success probability [7,17]. However, we can design a space-efficient heuristic that works well in practice.

The basic idea is to first determine the value set for each distinct variable involved in the rewriting, and then construct a *synopsis* for each value set. In our example, we can estimate the *size of intersection* of $A(?\mathsf{y_2})$ and $A(?\mathsf{y_3})$ based on their synopses. If the intersection size is estimated to be above some preset threshold with a reasonable probability, we consider the predicates as joinable; otherwise we issue an ASK query to verify if the join is actually empty; if it is, we remove this and other rewritings involving predicates $(?\mathsf{y_1}, \mathsf{p_1}, ?\mathsf{y_2})$ and $(?\mathsf{y_3}, \mathsf{p_2}, ?\mathsf{y_4})$. Note that our pruning step does not affect the soundness and completeness of our solution, as before pruning, we always issue an ASK query to make sure that rewriting has an empty result. In general, an ASK query is much cheaper than the corresponding SELECT query especially when the graph pattern is nonselective, and the synopses are used to avoid issuing unnecessary ASK queries (for those rewritings that are very likely to be non-empty).

The synopses should satisfy two key requirements. First, we should be able to estimate the size of intersection of multiple value sets (not just binary intersection) since a rewriting might include a join of m predicates on m variables. Let $?\mathsf{x_1}, ?\mathsf{x_2}, \ldots, ?\mathsf{x_m}$ denote these variables. To simplify notation, we use A_i to denote $A(?\mathsf{x_i})$. Second, the synopses of each variable should be able to estimate the distinct elements in its value set (as well as support distinct elements estimation under the set intersection operator). This requirement comes from the observation that we can estimate the size of an intersection $|A_1 \bigcap A_2|$ by simply estimating the size of $D(A_1 \bigcap A_2)$ where D is the number of distinct elements in A_1 and A_2, respectively. In what follows, we show that the *KMV-synopsis* [9] meets both requirements.

For a set A_1, we denote its KMV-synopsis as $\sigma(A_1)$. The construction of $\sigma(A_1)$ is as follows. Given a collision-resistant hash function h that generates (roughly) uniformly random hash values in its domain $[1, M]$, $\sigma(A_1)$ simply keeps the k smallest hash values from all elements in A_1, *i.e.,* $\sigma(A_1) = \{h(v_1), \ldots, h(v_k)\}$, where $v_i \in A_1$, and $h(v) \geq \max(\sigma(A_1))$ if $v \in A_1$ and $h(v) \notin \sigma(A_1)$. Then, $\widehat{D}(A_1) = \frac{k-1}{\max(\sigma(A_1))/M}$ is an unbiased estimator for $D(A_1)$ [9]. Furthermore, it is also possible to estimate the distinct number of elements in a general compound set (produced based on $A_1, \ldots, A_m$ with set union, intersection and difference operators) [9]. In our case, we are only interested in estimating $D(I)$ where $I = A_1 \bigcap A_2 \cdots \bigcap A_m$. Specifically, inspired by the discussion in [9], we can obtain an unbiased estimator $\widehat{D}(I)$ as follows. Define $\sigma(A_i) \oplus \sigma(A_j)$ as the set consisting of the k smallest values in $\sigma(A_i) \bigcup \sigma(A_j)$, and let $\sigma_{1\ldots m} = \sigma(A_1) \oplus \sigma(A_2) \cdots \oplus \sigma(A_m)$. Furthermore, let:

$$K_I = \left|\sigma_{1\ldots m} \bigcap \sigma(A_1) \bigcap \cdots \bigcap \sigma(A_m)\right| \text{ and,}$$
$$\widehat{D}(I) = \frac{K_I}{k}\left(\frac{k-1}{\max(\sigma_{1\ldots m})/M}\right). \tag{1}$$

We can show that (see proofs in [18]), by extending similar arguments from [9]:

Lemma 1. *For $k > 1$, $\widehat{D}(I)$ in Equation 1 is an unbiased estimator for $D(I)$.*

Lemma 2. *If $D(I) > 0, \epsilon \in (0,1)$ and $k \geq 1$, let $T = kD(I)/j$, it follows:*

$$\Pr\left(\frac{|\widehat{D}(I) - D(I)|}{D(I)} \leq \epsilon \,\middle|\, K_I = j\right) = \Delta(kD(I)/j, k, \epsilon) = \delta, \tag{2}$$
$$\Delta(T,k,\epsilon) = \sum_{i=k}^{T}\binom{T}{i}\left(\frac{k-1}{(1-\epsilon)T}\right)^i\left(1-\frac{k-1}{(1-\epsilon)T}\right)^{T-i}$$
$$-\sum_{i=k}^{T}\binom{T}{i}\left(\frac{k-1}{(1+\epsilon)T}\right)^i\left(1-\frac{k-1}{(1+\epsilon)T}\right)^{T-i}$$

In practice, given the observation of $\widehat{D}(I)$ and K_I, we can set $T = k\widehat{D}(I)/K_I$ and substitute T in Equation 2. Thus, we can obtain the confidence value δ for $\Pr\left(\frac{|\widehat{D}(I)-D(I)|}{D(I)} \leq \epsilon\right)$ for any error value ϵ. That said, our pruning technique works as follows.

We preset a small threshold value $\tau > 1$, a probability threshold $\theta \in [0, 1)$ and a relative error value $\epsilon \in (0, 1)$. For any m value sets of m variables to be joined in a rewriting, we estimate their intersection size as $\widehat{D}(I)$ by Equation 1, and $\delta = \Pr\left(\frac{|\widehat{D}(I)-D(I)|}{D(I)} \leq \epsilon\right)$ as above. Then, we check if $\widehat{D}(I)/(1+\epsilon) > \tau$ and $\delta > \theta$ (*i.e.,* if $D(I)$ is larger than τ with a probability $\geq \theta$). If this check returns false, we issue an ASK query to verify if the corresponding rewriting is empty; if yes, we can safely prune this rewriting. Otherwise (either the check returns true or the ASK returns nonempty), we consider that I is not empty and keep the current rewriting. In practice, we observe that the above procedure can be simplified by just checking if $\widehat{D}(I) \leq \tau$ for a small threshold value $\tau > 1$ (without using δ, θ and ϵ), which performs almost equally well.

To illustrate, consider again the rewriting in Table 1(c) of query $\mathsf{Q_U}^{\text{part}}$. To detect whether the rewriting is empty, we estimate the intersection size of the join in Table 1(c) using Equation 1. For the example, the equation indicates that the intersection is not larger than τ, and therefore we issue an ASK query. The ASK query evaluates the rewriting of Table 1(c) over the triples of Figure 1(a). Since there are no triples for persons that are relatives of friends-of-friends of person0, the ASK query returns false. Thus, $\mathsf{Q_U}^{\text{part}}$ is pruned.

Discussion on synopsis updates. The KMV-synopsis supports insertions (of a new item to the multiset the synopsis was initially built from) but not deletions (hence, it does not support the general update, which can be modeled as a deletion followed by an insertion) [9]. However, we can still use the KMV-synopsis to provide a quick estimation for pruning rewritings with empty results in case of updates to RDF stores, by only updating the synopses with the insertions and ignoring the deletions. Clearly, over the time, this will lead to an overestimation of the intersection size for multiple sets. However, such an overestimation only gives us false positives but not false negatives, i.e., we will not mistakenly prune any rewritings that do not produce an empty result. Of course, as the number of deletions increases, this approach will lead to too many false positives (rewritings that do produce empty results cannot be detected by checking their synopses) Hence, we can periodically rebuild all synopses after seeing enough number of deletions w.r.t. a user-defined threshold.

3.3 Optimizing the Generation of Rewritings

The pruning technique presented in Section 3.2 considers rewritings in isolation, to decide if a rewriting is empty or not. One way to integrate Algorithm SQR with the pruning technique will be: generating all the possible rewritings *in one shot* followed by a pruning step to remove empty rewritings from evaluation. However, such an integration ignores some inherent relationships between the rewritings, *i.e.,* that different rewritings share similar sub-queries. If we can quickly determine a common sub-query (*i.e.,* partial rewriting) is empty, it will save time that otherwise is needed to determine whether the rewritings contained in this sub-query are empty or not. In what follows, we show how one can optimize the rewriting by taking advantage of these common sub-queries. To illustrate, consider our running example and the rewriting of $\mathsf{Q_U}$ over the views in Figure 1(b). One generated rewriting q_1' for $\mathsf{Q_U}$ involves views $\mathsf{V_F}$, $\mathsf{V_R}$, $\mathsf{V_R}$, $\mathsf{V_R}$ with appropriate variable mappings since each view is in the CandV of predicate friend, lives, related and lives, respectively. Similarly, another generated rewriting q_2' involves views $\mathsf{V_F}$, $\mathsf{V_R}$, $\mathsf{V_{RoR}}$, $\mathsf{V_R}$. The key observations here is that (i) both rewritings involve a join of views $\mathsf{V_F}$ and $\mathsf{V_R}$; and (ii) from the optimization of the previous section, the join of views $\mathsf{V_F}$ and $\mathsf{V_R}$ is *empty* since the set of friends of "Eric" (see Figure 1(c)) is disjoint from his relatives. Therefore, both rewritings q_1' and q_2' can safely be removed (and every other rewriting involving a join of the two views over the corresponding predicates). By detecting with a single check the empty join between views $\mathsf{V_F}$ and $\mathsf{V_R}$, the algorithm optimized SQR (OSQR, see Algorithm 3) terminates immediately the *branch* of rewritings (including q_1' and q_2') involving these two views. To remove them from consideration, Algorithm SQR must check each generated individual rewriting independently. Algorithm OSQR addresses this shortcoming by building individual rewritings in a step-wise fashion. This way, OSQR *detects* and *terminates* early any *branch* of rewritings involving views whose join result is empty.

```
1  Input: Views $\mathcal{V}$, query Q with GP(Q)=($s^Q_1$, $p^Q_1$, $o^Q_1$), . . . ,
   ($s^Q_n$, $p^Q_n$, $o^Q_n$)
2  Output: a rewriting Q′ as a union of conjunctive queries
3  Set the query rewriting result Q′ to $\emptyset$.
4  Generate $\mathsf{CandV}_i$ for each triple pattern ($s^Q_i$, $p^Q_i$, $o^Q_i$), $1 \leq i \leq n$.
5  Set SubQ to $\emptyset$; initialize a stack STACK to store view
   combinations for SubQ.
6  Pick a triple pattern ($s^Q_i$, $p^Q_i$, $o^Q_i$), with the smallest size of
   $|\mathsf{CandV}_i|$.
7  Add ($s^Q_i$, $p^Q_i$, $o^Q_i$) into SubQ;
8  push each combination (SubQ, $\{\mathsf{V}, \mathsf{V} \in \mathsf{CandV}_i\}$) into STACK.
9  while STACK is non-empty do
10     Pop a combination $R$ from STACK; extract SubQ from $R$.
11     if SubQ contains all triple patterns in user query then
12         Generate a rewriting $q$ from $R$'s view set (lines 21-23 in
           SQR).
13         Q′ = Q′ $\cup$ $q$; goto line 9.
14     Get all triple patterns that can be joined with SubQ but not
       in SubQ;
15     Pick the triple pattern ($s^Q_j$, $p^Q_j$, $o^Q_j$) with the smallest size of
       $|\mathsf{CandV}_j|$.
16     for each view V in $\mathsf{CandV}_j$ do
17         Create a copy $R'$ of $R$ and a copy SubQ′ of SubQ.
18         if V is redundant with existing views in $R'$ then
19             Merge V with the view set of $R'$ (Sec. 3.1).
20         else  Add V into the view set of $R'$.
21         if the estimated result of a rewriting from $R'$ is empty
           (Sec. 3.2) then
22             Issue an ASK query corresponding to the rewriting.
23             if ASK query confirms the result is empty then  goto
               line 16.
24         Add ($s^Q_j$, $p^Q_j$, $o^Q_j$) in SubQ′ to replace SubQ in $R'$;
25         Push $R'$ in STACK.
```
Algorithm 3: The Optimized SQR (OSQR) Algorithm

In a nutshell, Algorithm OSQR works as follows. The algorithm uses a structure STACK where each element in STACK stores a sub-query SubQ of Q along with a candidate view combination for rewriting SubQ. Initially, STACK and SubQ are empty. The first sub-query considered corresponds to a triple pattern in Q, and we pick the pattern with the smallest size of |CandV| (*i.e.,* the number of views in CandV). Intuitively, this triple pattern is the most *selective* and by considering the most selective predicates in order (in terms of their |CandV|), we maximize the effects of early terminating a branch of rewritings once we detect the rewriting for SubQ results in an empty set (a larger portion of the rewritings for Q that contain this rewriting for SubQ is pruned earlier in this manner). After the first pattern, the algorithm considers one pattern added at each step. The way the pattern is picked (line 14) ensures that it can be joined with the current SubQ at the head of STACK, which increases the chance of optimization with techniques described in Section 3.1 and Section 3.2. Again, when more than one patterns are under consideration, the most selective one is picked. After a pattern is added and a candidate view for the pattern is picked, the view redundant with the existing view set for SubQ will be merged into the view set (lines 18-19). If the current rewriting for SubQ has an empty result (lines 21-23), the rewriting is not extended further and not pushed back into STACK.

We use CandV_1 and CandV_2 in Tables 1(a) and 1(b) to illustrate OSQR. Since $|\mathsf{CandV}_1|$ is smaller in size (line 6) , it first initializes STACK = $\{(\{\text{vfriend }\},\{\mathsf{V}_\mathrm{F}\}),\ (\{\text{vfriend }\},\{\mathsf{V}_\mathrm{FoF}\})\}$ (line 8). OSQR processes CandV_2 next (line 15). It iterates through CandV_2 from $(\mathsf{V}_\mathrm{R}, \Phi_{233})$ and detects that V_F in CandV_1 can not be merged with V_R in CandV_2 (line 18). Therefore, OSQR adds (vlives,V_R) to R' (line 20). Assume OSQR detects an empty result (line 21), (*e.g.,* the join of V_F and V_R for "Eric" is actually empty), OSQR issues an ASK query. If ASK returns negative (*i.e.,* empty), OSQR will skip lines 24-25 to avoid pushing ($\{$vfriend, vlives $\}$,$\{\mathsf{V}_\mathrm{F}$, $\mathsf{V}_\mathrm{R}\}$) into STACK. The above procedure iterates until STACK is empty.

4. EXPERIMENTS

We implemented our rewriting algorithms and optimization components in C++ and evaluated them on two RDF stores, namely, 4store [1] and Jena TDB [2]. Our relational database experiments were conducted using MySQL. For KMV synopsis, we set k=16 and $\tau = 2$ whenever the synopses were used (the simplified version of the checking procedure from Section 3.2 was adopted).

Here, we report the experimental results that compare the basic SPARQL query rewriting (SQR) algorithm with the optimized SQR (OSQR) algorithm, with detailed evaluation of the impact of individual optimization components. We used two key performance metrics, *i.e.,* the number of rewritings generated through query rewriting and the end-to-end evaluation time, including query rewriting and execution. Also we studied the scalability of our algorithms along multiple dimensions, *i.e.,* the size of query $|\mathsf{Q}|$, views $|\mathcal{V}|$ and $|\mathsf{CandV}|$. In experiments, we used the popular RDF benchmark LUBM [15] (which considers a setting in the university domain that involve students, departments, professors, etc.) to generate a dataset of 10M triples as the base data, over which views are defined using SPARQL queries. We run all experiments on a 64-bit Linux machine with a 2GHz Intel Xeon(R) CPU and 4GB of memory.

4.1 Experimental Results with 4Store

Native SPARQL rewriting vs. SQL expansion: In the introduction, we claim that translating SPARQL queries/views to SQL does not resolve the challenges addressed by our work. Here, we illustrate experimentally this is indeed the case. For the experiment we use the setup shown in Figure 3. In more detail, we use the seven view templates to instantiate 56 different views. Specifically, we create 14 views using template V_1 (each view with a different parameter in P_1), 12

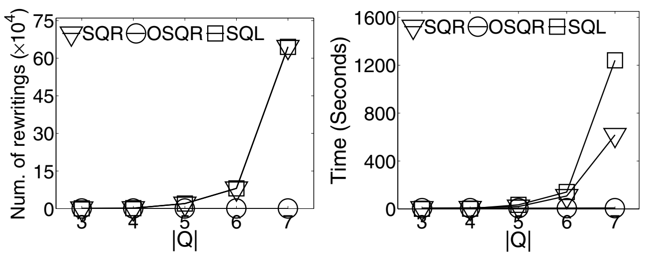
V_1:CONSTRUCT { ?x_1 name ?n_1 } WHERE { ?x_1 name ?n_1, ?x_1 worksFor $\langle P_1 \rangle$}
V_2:CONSTRUCT { ?x_2 email ?e_2 } WHERE { ?x_2 email ?e_2, ?x_2 worksFor $\langle P_2 \rangle$}
V_3:CONSTRUCT { ?x_3 degreeFrom ?d_3 } WHERE { ?x_3 degreeFrom ?d_3, ?x_3 worksFor $\langle P_3 \rangle$}
V_4:CONSTRUCT { ?x_4 phone ?p_4 } WHERE { ?x_4 phone ?p_4, ?x_4 worksFor $\langle P_4 \rangle$}
V_5:CONSTRUCT { ?x_5 teacherOf ?c_5 } WHERE { ?x_5 teacherOf ?c_5, ?x_5 worksFor $\langle P_5 \rangle$}
V_6:CONSTRUCT { ?x_6 interest ?i_6 } WHERE { ?x_6 teacherOf ?i_6, ?x_6 worksFor $\langle P_6 \rangle$}
V_7:CONSTRUCT { ?x_7 worksFor ?w_7 } WHERE { ?x_7 worksFor ?w_7, ?x_7 worksFor $\langle P_7 \rangle$}

(a) Views templates

Q:SELECT { ① ?x, ① ?n, ① ?e, ① ?d, ② ?p, ③ ?c, ④ ?i, ⑤ ?w }
WHERE { ① ?x name ?n, ① ?x email ?e, ① ?x degreeFrom ?d,
② ?x phone ?p, ③ ?x teacherOf ?c, ④ ?x interest ?i, ⑤ ?x worksFor ?w }

(b) Query template

Figure 3: Experimental Setup 1



(a) Rewritten queries over query size (b) Eval. time over query size

Figure 4: SPARQL rewriting vs. SQL expansion

views using template V_2 (using the same first 12 of the 14 parameters used for V_1), 10 views using template V_3 (using the same first 10 of the parameters used for V_1 and V_2), 8 views using template V_4 (using the same first 8 of the parameters used for V_1, V_2, and V_3), 6 views using template V_5 (using the same first 6 of the parameters used for V_1, V_2, V_3 and V_4), 4 views using template V_6 (using the same first 4 of the parameters used for V_1, V_2, V_3, V_4, and V_5), and 2 views using template V_7 (using the same first 2 of the parameters used for all the other views). Each view exposes some aspect of a student's data (*e.g.,* name, email). In terms of the query, we execute a different query in each iteration of the experiment. In iteration i, the query involves all the predicates in Figure 3(b) with an annotation $j \leq i$. So, the query initially has 3 predicates, and in each iteration we add one more predicate, up to a size of 7. Given the above setup, it is not hard to see that (i) the CandV for predicate name has 14 views, that for predicate email has 12, and finally for predicate worksFor has only 2 views; and (ii) for any two predicates p_i and p_j there are $min(|\mathsf{CandV}_i|, |\mathsf{CandV}_j|)$ non-empty joins between the two candidate views.

We also translate the SPARQL queries/views and the underlying RDF data to SQL and relational data. For the relational representation of RDF data we use (fully-indexed) predicate tables [4], which provide one of the most efficient representations of RDF in terms of query performance. Then, we compare algorithms SQR and OSQR as well as the corresponding relational/SQL-based representation (denoted as SQL in our figures). Figure 4 shows the comparison results. As the size of the input query increases, Algorithm OSQR results in between *one and four orders of magnitude* less queries as part of the rewriting process, while both algorithms SQR and the SQL view expansion result in the same number of queries. Meanwhile, Algorithm OSQR is up to *two orders of magnitude* faster than both SQR and SQL, in terms of the evaluation times for query rewriting and execution.

To illustrate that the above result holds for different queries and views we perform the same experiment with an alternative setup. In this setting, a query has three predicates

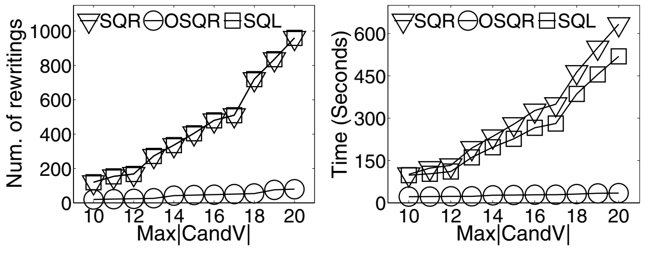
V$_1$:CONSTRUCT { ?x$_1$ email ?e$_1$, ?x$_1$ course ?c$_1$ }
    WHERE { ?x$_1$ email ?e$_1$, ?x$_1$ course ?c$_1$, ?x$_1$ member ?u$_1$, ?u$_1$ subOrg $\langle P_1 \rangle$}
V$_2$:CONSTRUCT { ?x$_2$ phone ?p$_2$, ?x$_2$ course ?c$_2$ }
    WHERE { ?x$_2$ phone ?p$_2$, ?x$_2$ course ?c$_2$, ?x$_2$ member ?u$_2$, ?u$_2$ subOrg $\langle P_2 \rangle$}
V$_3$:CONSTRUCT { ?x$_3$ degree ?d$_3$, ?x$_3$ course ?c$_3$ }
    WHERE { ?x$_3$ degree ?p$_3$, ?x$_3$ course ?c$_3$, ?x$_3$ member ?u$_3$, ?u$_3$ subOrg $\langle P_3 \rangle$}
V$_4$:CONSTRUCT { ?x$_4$ email ?e$_4$, ?x$_4$ course ?c$_4$ }
    WHERE { ?x$_4$ email ?e$_4$, ?x$_4$ course ?c$_4$, ?x$_4$ member ?u$_4$, ?u$_4$ subOrg $\langle P_3 \rangle$}
V$_5$:CONSTRUCT { ?x$_5$ email ?e$_5$, ?x$_5$ course ?c$_5$ }
    WHERE { ?x$_5$ email ?e$_5$, ?x$_5$ course ?c$_5$, ?x$_5$ member ?u$_5$, ?u$_5$ subOrg $\langle P_5 \rangle$}

(a) Views templates

Q: SELECT { ?x, ?e, ?c, ?d } WHERE { ?x email ?e, ?x course ?c, ?x degreeFrom ?d }

(b) Query template

Figure 5: Experimental Setup 2



(a) Rewritten queries over max CandV (b) Eval. time over max CandV

Figure 6: SPARQL rewriting vs. SQL expansion

and retrieves the email, degree, and all the courses taken by each student (see Figure 5(b)). The query is evaluated over views that have one of five view templates, denoted by V$_i$, $1 \leq i \leq 5$ (shown in Figure 5(a)). The templates are defined so that CandV$_{\text{courses}}$ = {V$_1$, V$_2$, V$_3$, V$_4$, V$_5$}, CandV$_{\text{degree}}$ = {V$_3$}, and CandV$_{\text{email}}$ = {V$_1$, V$_4$, V$_5$}. Notice that if each template is instantiated only once, SQR results in 15 rewritings. Normally, one expects that only a few of the rewritings are non-empty and hence we make 2 of the 15 rewritings non-empty, those involving templates V$_3$ and V$_4$. To do this, we make sure that the same variable P_3 is used for both view templates V$_3$ and V$_4$ and thus both templates are instantiated from the same university. Notice that definition-wise, view templates V$_1$, V$_4$ and V$_5$ are identical. However, we make sure that the three templates are instantiated from different universities so that they are non-overlapping in their contents. We create multiple instances of view templates using students from different departments, and by always populating pairs of instances of templates V$_3$ and V$_4$ from the same department, we make sure they join. Figure 6 shows the number of rewritings and evaluation times for SQR, OSQR and the corresponding relational/SQL setting. In the experiment, we start by instantiating each template twice (10 views in total), and proceed by picking a template and adding view instances in a way that linearly increases the cardinality of CandV$_{\text{courses}}$ (the largest CandV set). Figure 6 shows that as the size of the largest CandV set increases, OSQR generates up to *an order of magnitude* less rewritings than SQR and the SQL view expansion, resulting in up to *an order of magnitude* savings in evaluation times.

Optimizing Individual Rewritings: In Section 3 we introduced three orthogonal optimizations and in algorithm OSQR we incorporated all of them into a single algorithm. It is interesting to see what are the effects of each optimization in isolation, to the size of the rewriting and the evaluation time of the rewritten query. In the next three experiments we investigate exactly this, starting here with an experiment that studies the effects of optimizing individual

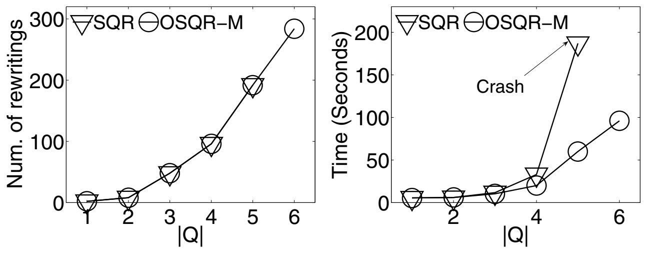
V$_1$:CONSTRUCT { ?x$_1$ name ?n$_1$, ?x$_1$ email ?e$_1$, ?x$_1$ takes ?c$_1$ }
    WHERE { ?x$_1$ name ?n$_1$, ?x$_1$ email ?e$_1$, ?x$_1$ takes ?c$_1$ }
V$_2$:CONSTRUCT { ?x$_2$ phone ?p$_2$, ?x$_2$ course ?c$_2$, ?x$_2$ member ?u$_2$ }
    WHERE { ?x$_2$ phone ?p$_2$, ?x$_2$ course ?c$_2$, ?x$_2$ member ?d$_2$ }
V$_3$:CONSTRUCT { ?x$_3$ phone ?p$_3$, ?x$_3$ course ?c$_3$, ?x$_3$ degree ?d$_3$ }
    WHERE { ?x$_3$ phone ?p$_3$, ?x$_3$ course ?c$_3$, ?x$_3$ degree ?d$_3$ }
V$_4$:CONSTRUCT { ?x$_4$ name ?n$_4$, ?x$_4$ email ?e$_4$, ?x$_4$ takes ?c$_4$ }
    WHERE { ?x$_4$ name ?n$_4$, ?x$_4$ email ?e$_4$, ?x$_4$ takes ?c$_4$ }
V$_5$:CONSTRUCT { ?x$_5$ phone ?p$_5$, ?x$_5$ course ?c$_5$, ?x$_5$ member ?u$_5$ }
    WHERE { ?x$_5$ phone ?p$_5$, ?x$_5$ course ?c$_5$, ?x$_5$ member ?u$_5$ }
V$_6$:CONSTRUCT { ?x$_6$ phone ?p$_6$, ?x$_6$ course ?c$_6$, ?x$_6$ degree ?d$_6$ }
    WHERE { ?x$_6$ phone ?p$_6$, ?x$_6$ course ?c$_6$, ?x$_6$ degree ?d$_6$ }

(a) Views templates

Q:SELECT { ?x, ① ?e, ② ?p, ③ ?c, ④ ?n, ⑤ ?u, ⑥ ?u′ }
  WHERE { ① ?x email ?e, ② ?x phone ?p, ③ ?x takes ?c,
④ ?x name ?n, ⑤ ?x member ?u, ⑥ ?x degree ?u′ }

(b) Query template

Figure 7: Experimental Setup 3



(a) Rewritten queries over query size (b) Eval. time over query size

Figure 8: Optimizing Individual Rewritings

rewritings (presented in Section 3.1). To this end, we *switch off* in OSQR *all* other optimizations but merging views (denoted as OSQR-M) and compare it with SQR. In terms of the experimental setup, this is shown in Figure 7. We define 6 views over our base data, with each view exposing some aspect of a student's data (*e.g.,* email, phone). As for the queries, we execute 6 different queries, with each query increasingly bringing together data from the views. The return values and predicates of the query executed in iteration i are marked appropriately in Figure 7(b). Figure 8 shows the results of the comparison between SQR and OSQR-M, as the input query size increases. Figure 8(a) shows that both algorithms result in the same number of rewritings; note that merging does not influence the number of generated rewritings (this is the focus of the other optimizations). Merging optimizes each individual rewriting, and this becomes apparent in the evaluation time of the rewritings (see Figure 8(b)). As the size of query |Q| increases, so is the potential for merging views (the same view might appear in the candidate view set of more predicates), which is confirmed in Figure 8(b) — savings in evaluation time of OSQR-M, compared to SQR, start from 10% to 70% for queries with 2 to 5 predicates. As |Q| increases, so is the size of each rewriting (since the rewriting ultimately integrates the where clauses of candidate views). In our experiments, when a (rewritten) query has approximately 16 predicates, the engine of 4store crashes, therefore, it is impossible to execute a rewriting from SQR when $|Q| \geq 6$. Since merging results in smaller rewritings, OSQR-M can handle larger input queries.

Pruning Rewritings with Empty Results: As before, we switch off in OSQR all other optimizations but pruning empty rewritings (denoted as OSQR-P) and compare it with SQR. The experimental setup used here is shown in Figure 9. Using the view template in Figure 9(a), we generate 10 views, where each view has a different value for the vari-

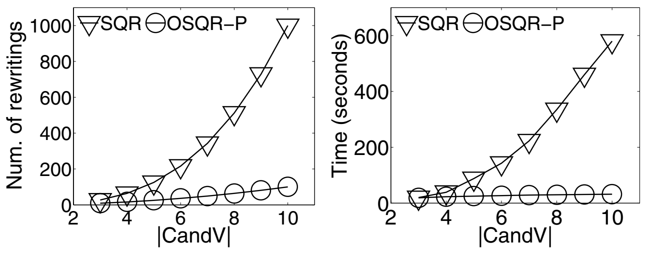
```
V:CONSTRUCT {?x1 name ?n1,
    ?x1 email ?e1,?x1 course ?c1 }
  WHERE { ?x1 name ?n1, ?x1 course ?c1,
    ?x1 email ?e1, ?x1 member ⟨P⟩ }
```

(a) Views templates

```
Q:SELECT { ?x, ?n, ?e, ?c }
  WHERE { ?x name ?n, ?x email ?e,
          ?x course ?c }
```

(b) Query template

Figure 9: Experimental Setup 4



(a) Rewritten queries over max CandV (b) Eval. time over max CandV

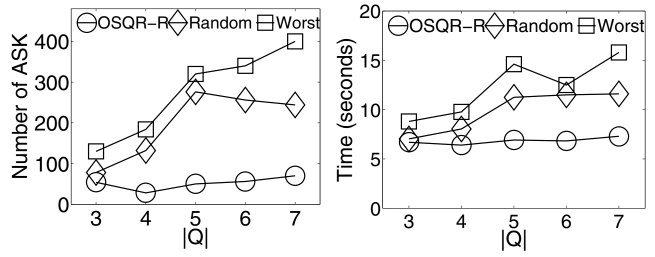
Figure 10: Pruning Empty Rewritings

able $\langle P \rangle$. Our instantiation is such that we use ten different departments from the same university as the values for variable $\langle P \rangle$. In this manner, we make sure that the views are non-overlapping. The experiment has 8 iterations. The same query Q (shown in Figure 9(b)) is evaluated across all iterations over a set of $i + 2$ views at iteration i. Notice that the CONSTRUCT statements of all views are identical to the graph pattern of the Q. It is not hard to see that for SQR, the CandV for each predicate of Q (name, email, course) contains all the views. Therefore, SQR will create $(i + 2)^3$ rewritings at iteration i. Contrarily, OSQR-P does not generate rewritings involving different views since these lead to empty results; synopses and ASK queries, which are less expensive, are executed to detect these empty results, and therefore in each iteration i essentially only $i+2$ queries need to be executed by OSQR-P. Figure 10 shows the comparison. Through synopses and ASK queries, OSQR-P produces an *order of magnitude* less rewritings than SQR, resulting in an *order of magnitude* faster evaluation times for query Q.

Optimizing the Generation of Rewritings: Here, we investigate the influence of sub-query (*i.e.,* triple pattern) ordering to OSQR. Since the objective of ordering is to improve the effectiveness of pruning, in OSQR we *only* switch off merging views; the algorithm is denoted as OSQR-R. We consider the same experimental setup with the one used in our first experiment, shown in Figure 3. For this setup, Figure 11 compares the performance of OSQR-R using 3 different reordering strategies. The figure shows the number of ASK queries issued during query rewriting (to detect empty rewritings), and the evaluation time of the rewritten query. Note that all the three reordering strategies result in the same number of nonempty rewritings, and only the numbers of ASK queries issued during rewriting are different; the latter affects the evaluation times, as shown in Figure 10. Using the proposed ordering on the size of CandV, OSQR-R detects the optimal ordering (which considers $p_1, p_2, \ldots, p_7$ in order) and generates up to an *order of magnitude* less ASK queries than either a random or the worst $(p_7, p_6, \ldots, p_1)$ ordering, resulting in near 60% savings in evaluation times.

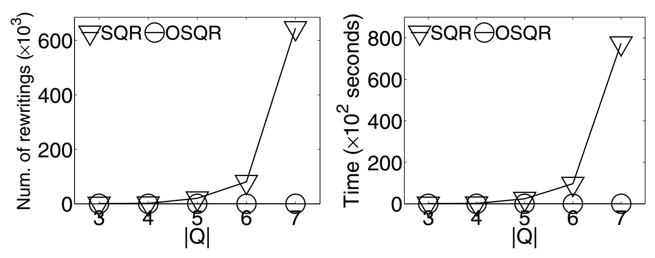
4.2 Experimental Results from Jena TDB

Using the same query and view definitions, we have run the same set of experiments on Jena TDB, to demonstrate the flexibility and the store-independent property of our algorithms. In general, the results from Jena TDB are highly



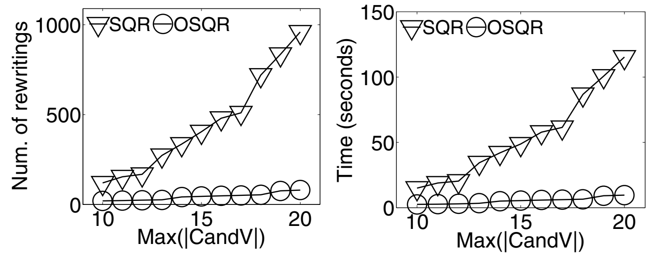
(a) ASK queries over query size (b) Eval. time over query size

Figure 11: Optimizing Rewriting Generation



(a) Rewritten queries over query size (b) Eval. time over query size

Figure 12: SQR vs. OSQR on Jena TDB



(a) Rewritten queries over max CandV (b) Eval. time over max CandV

Figure 13: SQR vs. OSQR on Jena TDB

consistent with our observations from 4store. As is evident from Figure 12, the overall performance in Jena TDB of OSQR is several orders of magnitude better than the SQR in the first experiment using the setup in Figure 3. The situation is similar when using the experimental setup of Figure 5 and the results are shown in Figure 13. These trends are highly consistent with what we have observed from their comparison in 4store (Figures 4 and 6, respectively).

4.3 Concluding remarks

Our experiments clearly illustrate the advantages of OSQR over SQR. In [18], we show, in a full set of experiments, that these results are not limited to 4store but carry over to Jena. Our experiments show that: we have realized the *first* practical rewriting solution (OSQR) which provides, *in real time*, sound and complete access of RDF data, *independent* of underlying RDF stores, with good efficiency in practice (to rewrite and evaluate a query over tens to hundred of views) and without the need to materialize intermediate data.

5. RELATED WORK

Query rewriting over views, motivated by a view-based approach to access control, has been well studied in relational (*e.g.,* [24]) and XML (*e.g.,* [13,14]) database. However, to the best of our knowledge, our work is the first on *native* query rewriting in SPARQL. SPARQL query rewriting combines the challenges that arise in the relational and XML settings: like the relational case, SPARQL query rewriting needs to synthesize multiple views; like the XML case, SPARQL query rewriting generates a query of exponential size. Previous work on rewriting SPARQL queries typically adopted a rule-based approach. In [12], the authors perform rewritings using predefined rewriting rules, whereas our rewriting techniques can dynamically compose the right views to rewrite a user query. Similarly in [10], the authors identify a set of tightest restrictions under which an XPath query can be rewritten over multiple views in PTIME. Such restrictions are expressed as rules during the rewriting, therefore this approach is rule-based as well. Reference [11] presents theoretical results for rewriting a query over multiple data sources; the authors studied the rewriting problem in the presence of embedded constraints from up to infinite data sources, and focused on the problem of deciding the *right* data sources that satisfy integrity constraints (*i.e.,* the expressibility and the support for the sources). Unlike our work, the rewriting algorithm in [11] does not guarantee completeness, and the optimization issue was not addressed.

Although our proposed SPARQL query rewriting techniques face similar challenges as the classical techniques for answering queries using views [16] and rewriting queries on semi-structured data [21], the actual rewriting steps differ significantly. In particular, relational techniques surveyed in [16] can not efficiently address the problem in SPARQL. For example, the pruning power of MiniCon [23] vanishes due to the fact that all the variables in SQL-translated views (see Figure 2(b)) are *distinguished* variables [23]. Furthermore, our computation of variable mappings and selection of candidate views are distinct from the query containment techniques discussed in [23]. The exponential size of the rewriting is also unique to our setting, which forces us to address new challenges not found in [16]. To address those challenges, we propose novel optimization techniques to remove empty rewritings from execution.

Existing works on *general* query rewriting in RDF store [5, 20] specify view definition in customized high-level languages, and perform query rewriting in an ad-hoc manner. In contrast, our work defines views in SPARQL, thus having more expressive power and wider applicability; furthermore, our SPARQL rewriting techniques are principled and independent of the underlying RDF stores.

6. CONCLUSION

We studied the classical problem of query rewriting over views in the context of SPARQL and RDF data. We proposed the first *sound* and *complete* query rewriting algorithm for SPARQL, with novel optimizations that (i) simplify individual rewritings by removing redundant triple patterns coming from the same view; (ii) eliminate rewritings with empty results based on a light-weight synopsis construction and efficient value-set intersection computation to estimate the size of joined triple patterns; and (iii) prune out big portions of the search space of rewritings (that lead to empty results) by optimizing the sequence of sub-query rewriting. Evaluation of our rewriting algorithm over two RDF stores showed its portability and its scalability in terms of query and view size. This work opens the gate to several interesting directions in future research, such as how to efficiently deal with variable predicates (instead of enumerating all predicates in the data to replace them) in query and view definition, how to partially materialize the views with the query rewriting in SPARQL to further improve the efficiency, and also, how to include other SPARQL features such as FILTER and OPTIONAL into the algorithm.

7. ACKNOWLEDGMENT

This work was done in part when Wangchao Le (intern) and Min Wang were with IBM. Wangchao Le and Feifei Li were partially supported by NSF Grant CNS-0831278.

8. REFERENCES

[1] 4store - scalable RDF storage. http://4store.org/.

[2] Jena semantic web framework. http://jena.sourceforge.net.

[3] Virtuoso universal server. http://virtuoso.openlinksw.com.

[4] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[5] F. Abel and et al. Enabling advanced and context dependent access control in RDF stores. In *ISWC*, 2007.

[6] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[7] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *STOC*, 1996.

[8] R. Angles and C. Gutierrez. The expressive power of SPARQL. In *ISWC*, pages 114–129, 2008.

[9] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *SIGMOD*, 2007.

[10] B. Cautis, A. Deutsch, and N. Onose. Xpath rewriting using multiple views: Achieving completeness and efficiency. In *WebDB*, 2008.

[11] B. Cautis, A. Deutsch, and N. Onose. Querying data sources that export infinite sets of views. In *ICDT*, 2009.

[12] G. Correndo, M. Salvadores, I. Millard, H. Glaser, and N. Shadbolt. SPARQL query rewriting for implementing data integration over linked data. In *EDBT*, 2010.

[13] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *SIGMOD*, 2004.

[14] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Rewriting regular XPath queries on XML views. In *ICDE*, pages 666–675, 2007.

[15] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 2005.

[16] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[17] B. Kalyanasundaram and G. Schintger. The probabilistic communication complexity of set intersection. *SIAM J. Discret. Math.*, 5(4):545–557, 1992.

[18] W. Le, S. Duan, A. Kementsietsidis, F. Li, and M. Wang. Query rewriting over SPARQL views. Technical report. http://ww2.cs.fsu.edu/~le/rdfview.pdf.

[19] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[20] G. Manjunath and et al. Semantic views for controlled access to the semantic web. In *Tech. Rep. HPL-08-15*, 2008.

[21] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *SIGMOD*, pages 455–466, 1999.

[22] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):1–45, 2009.

[23] R. Pottinger and A. Halevy. MiniCon: A scalable algorithm for answering queries using views. *VLDB J.*, 2001.

[24] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, pages 551–562, 2004.

[25] J. D. Ullman. Information integration using logical views. In *ICDT*, pages 19–40, 1997.

[26] Q. Wang and et al. On the correctness criteria of fine-grained access control in relational databases. In *VLDB*, 2007.