

Rewriting Queries on SPARQL Views

Wangchao Le¹ Songyun Duan² Anastasios Kementsietsidis² Feifei Li¹ Min Wang³

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 terms of both the rewriting size and evaluation time.

1. INTRODUCTION

In a number of settings, including access control [13, 14, 22, 24] or data integration [19, 23], 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, 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 evalu-

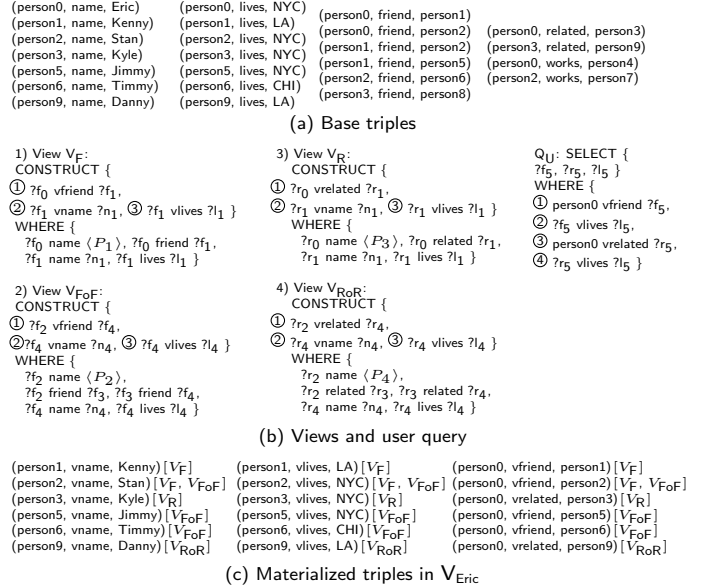


Figure 1: Motivating example

ation 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 case.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

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 the above 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 [22], 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”. Each triple in V_{Eric} is 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 [16], one

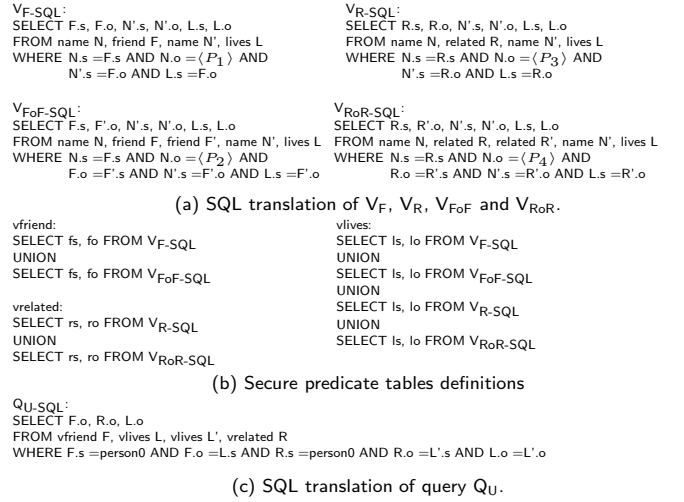


Figure 2: Attempting a relational/SQL rewriting

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], Sesame [9], C-store [4]), we show here that for a number of reasons *such an approach does not work.* 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 -SQL and V_{FoF} -SQL views (similarly for vrelated and vlives). How can we rewrite Q_U -SQL to a query over the base five tables? Since view expansion is supported in SQL, we can replace in Q_U -SQL the vfriend, vrelated, and vlives tables with their definitions in Figure 2(b), and in turn replace V_F -SQL, V_{FoF} -SQL, V_R -SQL and V_{RoR} -SQL with their definitions in Figure 2(a). Finally, it is not hard to see that the rewriting of Q_U -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; 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, and it is necessary to have a native 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: $Q := (\text{SELECT} \mid \text{CONSTRUCT}) \text{RD} (\text{WHERE GP})$, 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 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 [21].

The central technical problem in this paper is the *rewriting problem* as follows: given a set of views $\mathcal{V} = \{V_1, V_2, \dots, V_l\}$ 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 $Q'(G) = Q(\mathcal{V}(G))$. Like [24], we consider two criteria on the correctness of a rewriting, namely, *soundness* and *completeness*.

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

Soundness and completeness suffice to show that $Q(\mathcal{V}(G)) = Q'(G)$. 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 (s_v, p_v, o_v) in the result description $\text{RD}(V_j)$ of a view V_j and one of the triple patterns (s_q, p_q, o_q) in the graph pattern $\text{GP}(Q)$ 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.

```

1 Input: Views  $\mathcal{V}$ , query Q with  $\text{GP}(Q) = (s_1^Q, p_1^Q, o_1^Q), \dots, (s_n^Q, p_n^Q, o_n^Q)$ 
2 Output: a rewriting  $Q'$  as a union of conjunctive queries
3 for each  $(s_i^Q, p_i^Q, o_i^Q)$ ,  $1 \leq i \leq n$  do
4   Set  $\text{CandV}_i$  to  $\emptyset$ .
5   for each view  $V_j \in \mathcal{V}$  do
6     Let  $\text{RD}(V_j) = (s_1^{V_j}, p_1^{V_j}, o_1^{V_j}), \dots, (s_m^{V_j}, p_m^{V_j}, o_m^{V_j})$ 
7     for each  $(s_k^{V_j}, p_k^{V_j}, o_k^{V_j})$ ,  $1 \leq k \leq m$  do
8       if  $p_i^Q = p_k^{V_j}$  then
9         Set variable mapping  $\Phi_{ijk}$  to undefined
10        for the pair  $(s_i^Q, s_k^{V_j})$  of subjects (similarly objects  $(o_i^Q, o_k^{V_j})$ ) do
11          if var. mapping  $\phi : s_i^Q \rightarrow s_k^{V_j}$  exists then
12            if  $\phi$  maps two variables then  $\Phi_{ijk}(s_k^{V_j}) = s_i^Q$ 
13            else  $\Phi_{ijk}(s_k^{V_j}) = s_k^{V_j}$  ( $s_k^{V_j}$  is a constant)
14          if var. mapping  $\phi : s_k^{V_j} \rightarrow s_i^Q$  exists then
15            if  $\phi$  maps a variable to a constant then
16               $\Phi_{ijk}(s_k^{V_j}) = s_i^Q$ 
17            if  $\Phi_{ijk}$  is defined then
18              For any variable  $v'$  in  $\text{RD}(V_j)$  not in  $(s_k^{V_j}, p_k^{V_j}, o_k^{V_j})$ ,
19               $\Phi_{ijk}$  maps  $v'$  to a fresh variable (a new variable)
20              Add  $(V_j, \Phi_{ijk})$  to  $\text{CandV}_i$ 
21          Set the query rewriting result  $Q'$  to  $\emptyset$ 
22          for each entry in Cartesian product  $\text{CandV}_1 \times \dots \times \text{CandV}_n$  do
23            if  $\Phi_{1j_1k_1}, \Phi_{2j_2k_2}, \dots, \Phi_{nj_nk_n}$  is compatible then
24               $\text{RD}(q') = \text{RD}(Q)$ 
25               $\text{GP}(q') = \text{GP}(\Phi_{1j_1k_1}(V_{j_1}), \dots, \Phi_{nj_nk_n}(V_{j_n}))$ 
26               $Q' = Q' \cup q'$ 
27          return  $Q'$ 

```

Algorithm 1: SPARQL Query Rewriting (SQR) Algorithm

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 $(?f_0, \text{vfriend}, ?f_1)$ to $(\text{person0}, \text{vfriend}, ?f_5)$, which maps the variable $?f_0$ to the constant person0 and the variable $?f_1$ to the variable $?f_5$. There is no substitution from the second to the first triple since we cannot map the constant person0 to the variable $?f_0$.

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 $(\text{person0}, \text{vfriend}, ?f_5)$ and $(?f_6, \text{vfriend}, \text{person1})$. 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 $?f_5$ to constant person1 . It also computes a partial substitution from the second triple to the first that maps $?f_6$ to constant

$(V_F, \Phi_{111}) : \Phi_{111}(\?f_0, \?f_1, \?n_1, \?l_1) = (\text{person0}, \?f_5, \?v_0, \?v_1)$
$(V_{FoF}, \Phi_{121}) : \Phi_{121}(\?f_2, \?f_4, \?n_4, \?l_4) = (\text{person0}, \?f_5, \?v_2, \?v_3)$
(a) CandV ₁ for triple (person0, vfriend, ?f ₅)
$(V_F, \Phi_{213}) : \Phi_{213}(\?f_1, \?l_1, \?r_0, \?n_1) = (\?f_5, \?l_5, \?v_4, \?v_5)$
$(V_{FoF}, \Phi_{223}) : \Phi_{223}(\?f_4, \?l_4, \?r_2, \?n_4) = (\?f_5, \?l_5, \?v_6, \?v_7)$
$(V_R, \Phi_{233}) : \Phi_{233}(\?r_1, \?l_1, \?r_0, \?n_1) = (\?f_5, \?l_5, \?v_8, \?v_9)$
$(V_{RoR}, \Phi_{243}) : \Phi_{243}(\?r_4, \?l_4, \?r_2, \?n_4) = (\?f_5, \?l_5, \?v_{10}, \?v_{11})$
(b) CandV ₂ for triple (?f ₅ , vlives, ?l ₅)
GP(q') = { person0 name (P), person0 friend ?f ₅ , ?f ₅ friend ?f ₅ , ?f ₅ name ?v ₂ , ?f ₅ lives ?v ₃ , ?v ₈ name (P), ?v ₈ related ?f ₅ , ?f ₅ name ?v ₉ , ?f ₅ lives ?l ₅ }
(c) Rewritten body of Q _U ^{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 variables by default. For each compatible combination, we generate one query in the union.

To illustrate this, consider triples $t_1^{\text{QU}} = (\text{person0}, \text{vfriend}, \?f_5)$ and $t_2^{\text{QU}} = (\?f_5, \text{vlives}, \?l_5)$, from Q_U of Figure 1. For t_1^{QU} , CandV₁ = {(V_F, Φ₁₁₁), (V_{FoF}, Φ₁₂₁)}, where both Φ₁₁₁ and Φ₁₂₁ 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 CandV₂ for t_2^{QU} . To get Φ₁₁₁, SQR first considers t_1^{QU} with $t_1^{\text{VF}} = (\?f_0, \text{vfriend}, \?f_1)$ from V_F (lines 3-8). Then, for the pair of subjects (person0, ?f₀) (line 10), a var. mapping ϕ exists (line 14) from ?f₀ to person0. Therefore Φ₁₁₁ assigns the constant to the variable (line 15). Next, the pair of objects (?f₅, ?f₁) is considered (line 10) and as a result Φ₁₁₁ assigns ?f₁ to ?f₅ (lines 11-12). The remaining variables (?n₁ and ?l₁) in V_F are assigned to fresh variables respectively (?v₀ and ?v₁) (line 17). This concludes the computation of Φ₁₁₁. Other Φ's are computed accordingly. To illustrate, we consider the (partial) query Q_U^{part} of Q_U consisting only of triples t_1^{QU} and t_2^{QU} . Then, there are 8 rewritings of Q_U^{part} (lines 20-24), one for each combination of Φ's in CandV₁ and CandV₂. Table 1(c) shows the rewriting for Q_U^{part}, using (V_{FoF}, Φ₁₂₁) in CandV₁ and (V_R, Φ₂₃₃) in CandV₂.

Theorem 1. *The rewriting Q' of SQR is sound and complete (see proof in Appendix A).*

The cost of Algorithm SQR is influenced by the cost of computing variable mappings $\mathcal{O}(|Q| \times \sum_j |\text{RD}(V_j)|)$, but is dominated by the generation of rewritings and is thus equal to $\mathcal{O}((\sum_j |V_j|)^{|Q|})$, where |Q| (resp. |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_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_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_U). One such rewriting involves views V_F for vfriend, V_F for vlives, V_R for vrelated, and V_R for vlives. That is, the rewriting uses two copies of both V_F and V_R. Since the join (e.g., vfriend joined with vlives) in Q_U is done in a similar way as that in the view (correspondingly, V_F), there is redundancy to have two copies of V_F for this join; the similar situation happens to V_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_F are due to predicates vfriend and vlives being joined on variable ?f₅ in Q_U. But in the CONSTRUCT of V_F these two predicates are joined in a similar way. Therefore, one copy of V_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, Φ1) from CandV1, (V, Φ2) from CandV2
2 Output: (V, Φmerge)
3 Continue_merge = false;
4 for each triple pattern (s, p, o) in Φ1(RD(V)) do
5   Let (s', p, o') be the corresponding pattern in Φ2(RD(V))
6   if {s, o} ∩ {s', o'} ≠ ∅ then Continue_merge = true;
7 if Continue_merge == false then return (V, ∅);
8 for each triple pattern (s, p, o) in Φ1(RD(V)) do
9   Let (s', p, o') be the corresponding pattern in Φ2(RD(V))
10  Create corresponding merged pattern (sM, p, oM) for Φmerge
11  if s is a fresh variable then sM = s'; goto 14;
12  if s' is a fresh variable then sM = s; goto 14;
13  if s = s' then sM = s else return (V, ∅)
14  if o is a fresh variable then oM = o'; goto 8;
15  if o' is a fresh variable then oM = o; goto 8;
16  if o = o' then oM = o else return (V, ∅)
17 return (V, Φmerge)

```

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 p₁ and the other for its joinable predicate p₂, with variable mappings Φ₁ and Φ₂, 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. 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 corresponding to other predicates. Algorithm 2 ensures that the effects of these variable mappings are also merged. 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 Q_U the var. mapping (V_F, Φ_{111}) for predicate vfriend and (V_F, Φ_{213}) for predicate vives . Applying the two mapping functions respectively on V_F would result in two copies of V_F joined on $?f_5$. Since in V_F the triple patterns of vfriend and vives are joined in the same way as that in Q_U , Φ_{111} and Φ_{213} can be merged; $\Phi_{\text{merge}}(?v_4, ?f_5, ?v_0, ?v_1) = (\text{person0}, ?f_5, ?v_5, ?l_5)$. Therefore, the rewriting from Algorithm SQR involving two copies of 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 Appendix B).*

The cost of Algorithm 2 is $\mathcal{O}(|V|)$. 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 $\mathcal{O}(|Q| \times |V|)$.

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 Q_U^{part} (see Section 2.1) includes the rewriting q' in Table 1(c). Rewriting q' joins triples from V_{F_0F} and 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 $(?y_1, p_1, ?y_2)$ and $(?y_3, p_2, ?y_4)$, where the join equates $?y_2$ and $?y_3$. Denote the value set of a variable $?x$ as $A(?x)$. If we store $A(?x)$ for every variable in any triple pattern, this problem is trivial, *i.e.*, we simply check whether $A(?y_2) \cap A(?y_3) = \emptyset$. 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, 18]. However, we can design a space-efficient heuristic that works well in practice.

The basic idea of our solution 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 ex-

ample, we can estimate the *size of intersection* of $A(?y_2)$ and $A(?y_3)$ 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 $(?y_1, p_1, ?y_2)$ and $(?y_3, p_2, ?y_4)$. It is important to note that our pruning step does not affect the soundness and completeness of our solution, as before pruning a rewriting, we always issue an ASK query to make sure that rewriting has an empty result. Intuitively, an ASK query is cheaper than the corresponding SELECT query, 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 $?x_1, ?x_2, \dots, ?x_m$ denote these variables. To simplify notation, we use A_i to denote $A(?x_i)$. 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 $|A_1 \cap A_2|$ by simply estimating the size of $D(A_1 \cap A_2)$ where D is the number of distinct elements in A_1 and A_2 , respectively. In what follows, we show that the *KMV-synopsis* [8] 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.*, $\sigma(A_1) = \{h(v_1), \dots, h(v_k)\}$, 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 $D(A_1)$ [8]. Furthermore, it is also possible to estimate the distinct number of elements in a general compound set (produced based on A_1, \dots, A_m with set union, intersection and difference operators) [8]. In our case, we are only interested in estimating $D(I)$ where $I = A_1 \cap A_2 \cdots \cap A_m$. Specifically, inspired by the discussion in [8], 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 $\sigma(A_i) \cup \sigma(A_j)$, and let $\sigma_{1\dots m} = \sigma(A_1) \oplus \sigma(A_2) \cdots \oplus \sigma(A_m)$. Furthermore, let:

$$K_I = \left| \sigma_{1\dots m} \cap \sigma(A_1) \cap \cdots \cap \sigma(A_m) \right| \text{ and,} \quad (1)$$

$$\widehat{D}(I) = \frac{K_I}{k} \left(\frac{k-1}{\max(\sigma_{1\dots m})/M} \right).$$

We can show that (see proofs in Appendix C and D), by extending similar arguments from [8]:

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 $T = kD(I)/j$:*

$$\Pr \left(\frac{|\widehat{D}(I) - D(I)|}{D(I)} \leq \epsilon \mid K_I = j \right) = \Delta(kD(I)/j, k, \epsilon) = \delta, \quad (2)$$

$$\Delta(T, k, \epsilon) = \sum_{i=k}^T \binom{T}{i} \left(\frac{k-1}{(1-\epsilon)T} \right)^i \left(1 - \frac{k-1}{(1-\epsilon)T} \right)^{T-i}$$

$$- \sum_{i=k}^T \binom{T}{i} \left(\frac{k-1}{(1+\epsilon)T} \right)^i \left(1 - \frac{k-1}{(1+\epsilon)T} \right)^{T-i}$$

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 Q_U^{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.

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) [8]. 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. For more expositions on synopsis updates, please refer to Appendix F.

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 Q_U over the views in Figure 1(b). One generated rewriting q'_1 for Q_U involves views V_F, V_R, V_R, 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 V_F, V_R, V_{RoR}, V_R . The key observations here is that (i) both rewritings involve a join of views V_F and V_R ; and (ii) from the optimization of the previous section, the join of views V_F and 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 V_F and 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  $GP(Q)=(s_1^Q, p_1^Q, o_1^Q), \dots,$ 
    $(s_n^Q, p_n^Q, o_n^Q)$ 
2 Output: a rewriting  $Q'$  as a union of conjunctive queries
3 Set the query rewriting result  $Q'$  to  $\emptyset$ .
4 Generate  $CandV_i$  for each triple pattern  $(s_i^Q, p_i^Q, o_i^Q)$ ,  $1 \leq i \leq n$ .
5 Set SubQ to  $\emptyset$ ; initialize a stack STACK to store view
   combinations for SubQ.
6 Pick a triple pattern  $(s_i^Q, p_i^Q, o_i^Q)$ , with the smallest size of
    $|CandV_i|$ .
7 Add  $(s_i^Q, p_i^Q, o_i^Q)$  into SubQ;
8 push each combination  $(SubQ, \{V, V \in CandV_i\})$  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  $(s_j^Q, p_j^Q, o_j^Q)$  with the smallest size of
    $|CandV_j|$ .
16 for each view  $V$  in  $CandV_j$  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  $(s_j^Q, p_j^Q, o_j^Q)$  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 $|\text{CandV}|$), 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 $|\text{CandV}_1|$ is smaller in size (line 6), it first initializes $\text{STACK} = \{(\{\text{vfriend}\}, \{\text{V}_F\}), (\{\text{vfriend}\}, \{\text{V}_{\text{FoF}}\})\}$ (line 8). OSQR processes CandV_2 next (line 15). It iterates through CandV_2 from (V_R, Φ_{233}) and detects that V_F in CandV_1 can not be merged with V_R in CandV_2 (line 18). Therefore, OSQR adds $(\text{vives}, \text{V}_R)$ 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 $(\{\text{vfriend}, \text{vives}\}, \{\text{V}_F, \text{V}_R\})$ 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 $|Q|$, views $|\mathcal{V}|$ and $|\text{CandV}|$. 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 instanti-

```

V1:CONSTRUCT { ?x1 name ?n1 } WHERE { ?x1 name ?n1, ?x1 worksFor {P1} }
V2:CONSTRUCT { ?x2 email ?e2 } WHERE { ?x2 email ?e2, ?x2 worksFor {P2} }
V3:CONSTRUCT { ?x3 degreeFrom ?d3 } WHERE { ?x3 degreeFrom ?d3, ?x3 worksFor {P3} }
V4:CONSTRUCT { ?x4 phone ?p4 } WHERE { ?x4 phone ?p4, ?x4 worksFor {P4} }
V5:CONSTRUCT { ?x5 teacherOf ?c5 } WHERE { ?x5 teacherOf ?c5, ?x5 worksFor {P5} }
V6:CONSTRUCT { ?x6 interest ?i6 } WHERE { ?x6 teacherOf ?i6, ?x6 worksFor {P6} }
V7:CONSTRUCT { ?x7 worksFor ?w7 } WHERE { ?x7 worksFor ?w7, ?x7 worksFor {P7} }

```

(a) Views templates

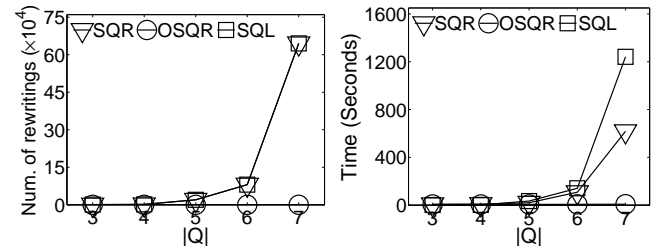
```

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

ate 56 different views. Specifically, we create 14 views using template V_1 (each view with a different parameter in P_1), 12 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 $\min(|\text{CandV}_i|, |\text{CandV}_j|)$ 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 (refer to Appendix G.1 for the SQL translation in detail). Then, we compare algorithms SQR and OSQR as well as the corresponding relational/SQL-based representation. 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. At the same time, Algorithm OSQR is up to *two orders of magnitude* faster than both SQR and SQL, in terms of the evaluation times for query

```

V1:CONSTRUCT { ?x1 email ?e1, ?x1 course ?c1 }
  WHERE { ?x1 email ?e1, ?x1 course ?c1, ?x1 member ?u1, ?u1 subOrg (P1) }
V2:CONSTRUCT { ?x2 phone ?p2, ?x2 course ?c2 }
  WHERE { ?x2 phone ?p2, ?x2 course ?c2, ?x2 member ?u2, ?u2 subOrg (P2) }
V3:CONSTRUCT { ?x3 degree ?d3, ?x3 course ?c3 }
  WHERE { ?x3 degree ?p3, ?x3 course ?c3, ?x3 member ?u3, ?u3 subOrg (P3) }
V4:CONSTRUCT { ?x4 email ?e4, ?x4 course ?c4 }
  WHERE { ?x4 email ?e4, ?x4 course ?c4, ?x4 member ?u4, ?u4 subOrg (P3) }
V5:CONSTRUCT { ?x5 email ?e5, ?x5 course ?c5 }
  WHERE { ?x5 email ?e5, ?x5 course ?c5, ?x5 member ?u5, ?u5 subOrg (P5) }

```

(a) Views templates

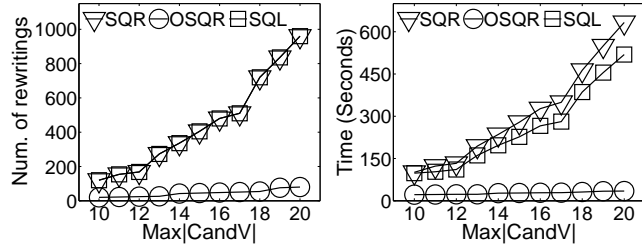
```

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

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 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 $\text{CandV}_{\text{courses}} = \{V_1, V_2, V_3, V_4, V_5\}$, $\text{CandV}_{\text{degree}} = \{V_3\}$, and $\text{CandV}_{\text{email}} = \{V_1, V_4, V_5\}$. 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 $\text{CandV}_{\text{courses}}$ (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.

```

V1:CONSTRUCT { ?x1 name ?n1, ?x1 email ?e1, ?x1 takes ?c1 }
  WHERE { ?x1 name ?n1, ?x1 email ?e1, ?x1 takes ?c1 }
V2:CONSTRUCT { ?x2 phone ?p2, ?x2 course ?c2, ?x2 member ?u2 }
  WHERE { ?x2 phone ?p2, ?x2 course ?c2, ?x2 member ?u2 }
V3:CONSTRUCT { ?x3 phone ?p3, ?x3 course ?c3, ?x3 degree ?d3 }
  WHERE { ?x3 phone ?p3, ?x3 course ?c3, ?x3 degree ?d3 }
V4:CONSTRUCT { ?x4 name ?n4, ?x4 email ?e4, ?x4 takes ?c4 }
  WHERE { ?x4 name ?n4, ?x4 email ?e4, ?x4 takes ?c4 }
V5:CONSTRUCT { ?x5 phone ?p5, ?x5 course ?c5, ?x5 member ?u5 }
  WHERE { ?x5 phone ?p5, ?x5 course ?c5, ?x5 member ?u5 }
V6:CONSTRUCT { ?x6 phone ?p6, ?x6 course ?c6, ?x6 degree ?d6 }
  WHERE { ?x6 phone ?p6, ?x6 course ?c6, ?x6 degree ?d6 }

```

(a) Views templates

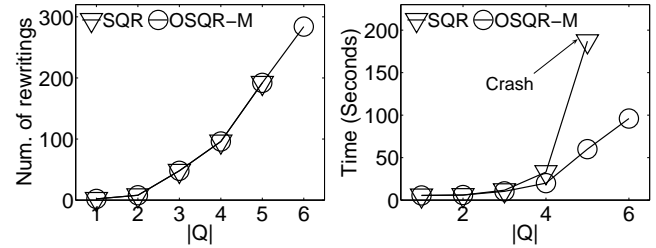
```

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

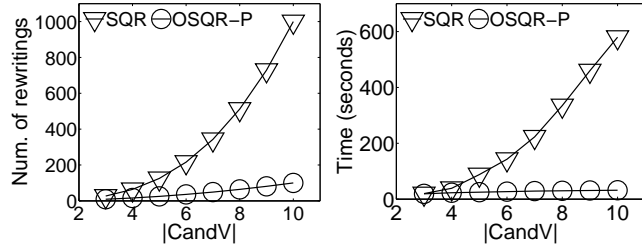
It is interesting to see what are the effects of each optimization in isolation, to the size of the rewriting and the evaluation time of the rewritten query. In the next three experiments we investigate exactly this, starting here with an experiment that studies the effects of optimizing individual rewritings (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.

V:CONSTRUCT { ?x₁ name ?n₁,
?x₁ email ?e₁, ?x₁ course ?c₁ }
WHERE { ?x₁ name ?n₁, ?x₁ course ?c₁,
?x₁ email ?e₁, ?x₁ member (P) }

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

(a) Views templates

(b) Query template

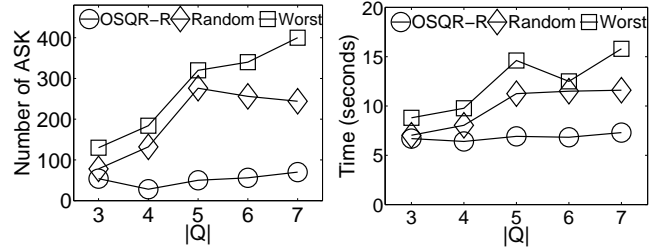
Figure 9: Experimental Setup 4

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

Figure 10: Pruning Empty Rewritings

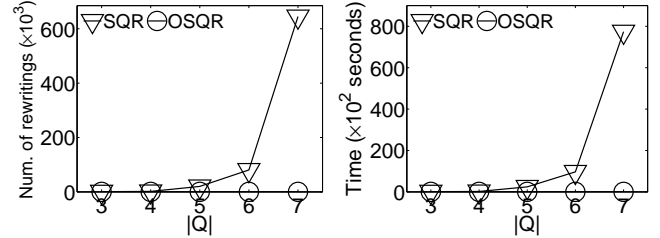
Pruning Rewritings with Empty Results: As before, we switch off in OSQR all other optimizations but pruning empty rewritings (denoted as OSQR-P) and compare it with SQR. The experimental setup used here is shown in Figure 9. Using the view template in Figure 9(a), we generate 10 views, where each view has a different value for the variable $\langle P \rangle$. 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, \dots, 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, \dots, p_1) or-



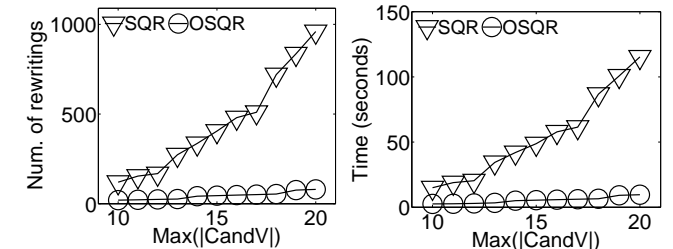
(a) ASK queries over query size

(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

dering, 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 exactly 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 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 already observed from their comparison in 4store (Figures 4 and 6, respectively). In Appendix G.2, we report results from the whole set of experiments on Jena TDB.

4.3 Concluding remarks

Our experiments clearly illustrate the advantages of OSQR over SQR. In Appendix G.2, we show 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 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.*, [22]) 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 pre-defined 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 with the classical technique for answering queries using views [17], the actual rewriting steps differ significantly. Our computation of variable mappings is distinct from the query containment techniques used there, as is our determination of candidate views. The exponential size of the rewriting is also unique to our setting, a problem which forces us to address new challenges not found in [17]. 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. Experiments of our rewriting algorithm over two RDF stores showed

both its portability and its scalability in terms of query and view size.

This work opens the gate for several interesting directions to future research, such as how to partially materialize the views and combine it with the query rewriting in SPARQL to further improve the efficiency, how to incrementally maintain the query rewriting when new views are added or existing views are dropped, etc.

7. REFERENCES

- [1] 4store - scalable RDF storage. <http://4store.org/>.
- [2] Jena: a semantic web framework for java. <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] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *SIGMOD*, 2007.
- [9] J. Broekstra and et al. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC*, 2002.
- [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] C. Gutierrez. Foundations of RDF databases. In *ESWC*, 2008.
- [17] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.
- [18] B. Kalyanasundaram and G. Schintger. The probabilistic communication complexity of set intersection. *SIAM J. Discret. Math.*, 5(4):545–557, 1992.
- [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-2008-15*, 2008.
- [21] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):1–45, 2009.
- [22] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, pages 551–562, 2004.
- [23] J. D. Ullman. Information integration using logical views. In *ICDT*, pages 19–40, 1997.
- [24] Q. Wang and et al. On the correctness criteria of fine-grained access control in relational databases. In *VLDB*, 2007.

APPENDIX

A. PROOF OF THEOREM 1

Soundness: Algorithm 1 generates a rewriting Q' of the input query $q(\bar{X}) :- p_1(\bar{X}_1), \dots, p_n(\bar{X}_n)$ as a union of conjunctive queries, *i.e.*, $Q' = \cup q'$, with one query q' for each entry in the Cartesian product of $\text{CandV}_1 \times \dots \times \text{CandV}_n$ (lines 20-24 of the algorithm). To prove soundness, it suffices to show that that $q'(\mathcal{G}) \subseteq q(\mathcal{V}(\mathcal{G}))$, for each q' of Q' .

Let q' be the query corresponding to the $(\Phi_{1j_1k_1}(\mathcal{V}_{j_1}), \dots, \Phi_{nj_nk_n}(\mathcal{V}_{j_n}))$ entry of the Cartesian product, and let $\mathcal{V}'_{j_i} = \Phi_{ij_ik_i}(\mathcal{V}_{j_i})$, $1 \leq i \leq n$, that is, \mathcal{V}'_{j_i} is the view that results in by applying the variable mapping $\Phi_{ij_ik_i}$ to view \mathcal{V}_{j_i} . It is not hard to see that $\mathcal{V}'_{j_i}(\mathcal{G}) \subseteq \mathcal{V}_{j_i}(\mathcal{G})$. But then, $\cup_i \mathcal{V}'_{j_i}(\mathcal{G}) \subseteq \cup_i \mathcal{V}_{j_i}(\mathcal{G})$, and therefore $\cup_i \mathcal{V}'_{j_i}(\mathcal{G}) \subseteq \mathcal{V}(\mathcal{G})$. Applying query q in both sides of the containment relation, we get $q(\cup_i \mathcal{V}'_{j_i}(\mathcal{G})) \subseteq q(\cup_i \mathcal{V}_{j_i}(\mathcal{G})) \subseteq q(\mathcal{V}(\mathcal{G}))$. By construction, query q' considers one possible way of evaluating query q over $\cup_i \mathcal{V}'_{j_i}(\mathcal{G})$, the one that considers the predicates $p_i(\bar{X}_i)$ in each $\mathcal{V}'_{j_i}(\mathcal{G})$ (remember that predicate $p_i(\bar{X}_i)$ appears in the head of \mathcal{V}'_{j_i} due to $\Phi_{ij_ik_i}$). Thus, $q'(\mathcal{G}) \subseteq q(\cup_i \mathcal{V}'_{j_i}(\mathcal{G}))$, which implies that $q'(\mathcal{G}) \subseteq q(\mathcal{V}(\mathcal{G}))$.

Completeness: To prove completeness, it suffices to show that $Q(\mathcal{V}(\mathcal{G})) \subseteq Q'(\mathcal{G})$. Consider $q(\bar{X}) :- p_1(\bar{X}_1), \dots, p_n(\bar{X}_n)$ and let $\mathcal{A}(p_i(\bar{X}_i), \mathcal{V}(\mathcal{G})) = \{\mathcal{T} = (s, p, o) \mid \mathcal{T} \in \mathcal{V}(\mathcal{G}) \text{ and there exists a valuation } \phi_i \text{ such that } \phi_i(p_i(\bar{X}_i)) = \mathcal{T}\}$, that is, $\mathcal{A}(p_i(\bar{X}_i), \mathcal{V}(\mathcal{G}))$ contains all the triples in $\mathcal{V}(\mathcal{G})$ satisfying $p_i(\bar{X}_i)$.

Now, consider a set of triples $\mathcal{T}_1, \dots, \mathcal{T}_n$ such that (i) $\mathcal{T}_i \in \mathcal{A}(p_i(\bar{X}_i), \mathcal{V}(\mathcal{G}))$; and (ii) there exists a valuation ϕ (the composition of valuations ϕ_i , for each i) that maps the body $\text{BD}(q)$ of query q to the set of \mathcal{T}_i triples. If no such valuation exists for any set of triples, then $Q(\mathcal{V}(\mathcal{G})) = \emptyset$ and completeness trivially holds. Now, each triple \mathcal{T}_i is generated by a predicate t_{k_i} of some view \mathcal{V}_{j_i} . This implies the existence of a valuation Ψ_{j_i} of view \mathcal{V}_{j_i} over \mathcal{G} such that $\Psi_{j_i}(t_{k_i}) = \mathcal{T}_i$. It is not hard to see that due to the existence of valuations ϕ_i between $p_i(\bar{X}_i)$ and \mathcal{T}_i , and Ψ_{j_i} between \mathcal{V}_{j_i} and \mathcal{T}_i , Algorithm 1 constructs (in lines 9-17) a variable mapping $\Phi_{ij_ik_i}$ for \mathcal{V}_{j_i} . We show that $\mathcal{T}_i \in (\Phi_{ij_ik_i}(\mathcal{V}_{j_i}))(\mathcal{G})$. This is proven by contradiction. If $\mathcal{T}_i \notin (\Phi_{ij_ik_i}(\mathcal{V}_{j_i}))(\mathcal{G})$ then variable mapping $\Phi_{ij_ik_i}$ assigns a variable v in \mathcal{V}_{j_i} (and in particular in t_{k_i}) to a constant c (mappings between variables do not affect the evaluation of \mathcal{V}_{j_i}) which causes the exclusion of \mathcal{T}_i from the results (c is not one of the constants in \mathcal{T}_i), a contradiction. Notice that in Algorithm 1 (lines 10-15), any binding of variables in $\Phi_{ij_ik_i}$ is using constants from query q (and therefore from \mathcal{T}_i).

Consider now query q' constructed by considering all the $\Phi_{ij_ik_i}(\mathcal{V}_{j_i})$ corresponding to triples \mathcal{T}_i . Query q' is constructed by Algorithm 1 since (a) the $\Phi_{ij_ik_i}(\mathcal{V}_{j_i})$ are compatible, due to the existence of valuation ϕ ; and (b) Algorithm 1 is exhaustive and considers all possible combinations of variable mappings, and therefore it will consider the above combination. It is not hard to see then that for the triples \mathcal{T}_i , evaluating q over the triples is equivalent to evaluating $q'(\mathcal{G})$. That is, for any set of triples satisfying q , there exists a corresponding query q' that produces the same result. By considering (the union of) all possible sets of triples that satisfy q , we can infer that $q(\mathcal{V}(\mathcal{G})) \subseteq \cup q'(\mathcal{G})$, *i.e.*, $Q(\mathcal{V}(\mathcal{G})) \subseteq Q'(\mathcal{G})$.

B. PROOF OF THEOREM 2

Without loss of generality, suppose user query Q_U is composed of two triple patterns $p_1(\bar{X}_1)$ and $p_2(\bar{X}_2)$, and one rewriting q from Algorithm 1 is: $\text{HD}(q) = p_1(\bar{X}_1), p_2(\bar{X}_2)$ and $\text{BD}(q) = \text{BD}(\Phi_1(\mathcal{V})) \wedge \text{BD}(\Phi_2(\mathcal{V}))$. Recall that Algorithm 2 tries to simplify q by removing redundant triple patterns in q . If Φ_{merge} returned from Algorithm 2 is *null*, the rewriting q is retained (thus the soundness and completeness of Algorithm 1 is not affected); otherwise, q is replaced with a simplified rewriting q' : $\text{HD}(q') = p_1(\bar{X}_1), p_2(\bar{X}_2)$ and $\text{BD}(q') = \text{BD}(\Phi_{\text{merge}}(\mathcal{V}))$. Since both $\Phi_1(\mathcal{V})$ and $\Phi_2(\mathcal{V})$ are conjunctive queries and are joined (lines 4 ~ 6 of Algorithm 2), q is also a conjunctive query. Note that only fresh variables in q and q' are treated as variables when constructing the variable mapping function Φ_{merge} ; constants and variables appearing in the head of q are treated as constants.

To prove that q' is equivalent to q , we first prove q' is contained in q , *i.e.*, $q' \subseteq q$. The way we construct Φ_{merge} (lines 11 ~ 16) guarantees that there is a homomorphism φ from q to q' : $\varphi(s) = s_M$, $\varphi(s') = s_M$, $\varphi(o) = o_M$, and $\varphi(o') = o_M$, as s and s' (correspondingly, o and o') are either fresh variables or identical to s_M (correspondingly, o_M). Since $t(s, o)$ and $t(s', o')$ have the same predicate as $t(s_M, o_M)$, it is easy to see that $\varphi(q) = q'$. Therefore, q' is contained in q , *i.e.*, $q' \subseteq q$.

Next, we prove that $q \subseteq q'$. For any triple pattern $t(s_M, o_M)$ in q' , in which both s_M and o_M are not fresh variables (lines 13 and 16), at least one of the triple patterns $t(s, o)$ and $t(s', o')$ is identical to $t(s_M, o_M)$. For other triple patterns in q' that involve fresh variables, there must exist (at least one) isomorphism between these triple patterns and the corresponding triple patterns in q . Therefore, q' is equivalent to a subquery of q . Thus, we have $q \subseteq q'$.

As both $q' \subseteq q$ and $q \subseteq q'$ hold, we have proved that q is equivalent to q' .

C. PROOF OF LEMMA 1

First of all, by Theorem 5 in [8] and discussion therein, we know that $\sigma_{1\dots m}$ is the size- k KMV-synopsis for the union of sets A_1, A_2, \dots, A_m , *i.e.*, $\sigma_{1\dots m} = \sigma(A_1 \cup A_2 \dots \cup A_m) = \sigma(U)$, where $U = A_1 \cup A_2 \dots \cup A_m$. Hence, $\frac{k-1}{\max(\sigma_{1\dots m})/M}$ is an unbiased estimator for $D(U)$ by the basic property of the KMV-synopsis (see Section 4.1 in [8]).

Next, we simply extend the similar arguments from Section 5.2 in [8] for the intersection over two sets to the intersection over multiple sets. Let V_U be the set of k values from the original value sets A_1 to A_m that correspond to the hash values in $\sigma_{1\dots m}$, then, by a trivial extension of Lemma 1 in [8], we can show that for each $v \in V_U$, we have $v \in A_i$ if and only if $h(v) \in \sigma(A_i)$. This implies that $v \in I$, where $I = A_1 \cap A_2 \dots \cap A_m$, if and only if $h(v) \in \sigma(A_1) \cap \sigma(A_2) \dots \cap \sigma(A_m)$. Hence:

$$K_{\cap} = |v \in V_U : v \in A_1 \cap A_2 \dots \cap A_m| = K_I, \quad (3)$$

where K_I is given by Equation 1.

Clearly, V_U could be viewed as a random sample of k elements from $\mathcal{D}(U)$, where $\mathcal{D}(\cdot)$ represents the set of distinct elements from an input. Hence, $\rho = D(I)/D(U)$ can be estimated by $\hat{\rho} = K_{\cap}/k$, the fraction of sample elements in V_U that also belong to $\mathcal{D}(I)$. As a result, $\hat{\rho}$ gives an unbiased estimator for ρ .

Finally, since $D(I) = \rho D(U)$, then $E(\hat{\rho} \cdot \hat{D}(U)) = D(I)$ if

$\widehat{D}(U)$ is an unbiased estimator of $D(U)$. Observe that $\widehat{\rho} = K_{\cap}/k = K_I/k$ (by Equation 3) and $\widehat{D}(U) = \frac{k-1}{\max(s_1, \dots, m)/M}$ is an unbiased estimator of $D(U)$ (from the first paragraph in this Section), we complete the proof.

D. PROOF OF LEMMA 2

The key observation is that for the intersection over multiple sets (more than 2), the following arguments made for the intersection over two sets in Section 5.2 from [8] still hold. V_U is a uniform random sample of size k drawn from $\mathcal{D}(U)$, and K_{\cap} is a random variable representing the number of elements in V_U that also belongs to $\mathcal{D}(I)$. Hence, similar to Equation 9 in [8], for intersection over m sets, we also have:

$$\Pr(K_{\cap} = j) = \binom{D(I)}{j} \binom{D(U) - D(I)}{k - j} / \binom{D(U)}{k}, \quad (4)$$

where K_{\cap} is given in Equation 3. Hence, K_{\cap} is a random variable with a hypergeometric distribution, just as the same random variable defined for the intersection over two sets in Section 5.2 from [8]. The rest of the proof for $\Pr\left(\frac{|\widehat{D}(I) - D(I)|}{D(I)} \leq \epsilon \mid K_I = j\right)$ given in Equation 2 of Lemma 2 follows exactly the same fashion as the proof of Theorem 6 in [8]. We omit the detailed derivations (simple substitution of parameters for several functions defined in [8] in the proof of Theorem 6) that eventually lead to our Equation 2. This completes the proof.

E. OTHER REWRITING OPTIMIZATIONS

Triple Patterns with Variable Predicates. It requires an extra step to figure out what are the candidate predicates that match a variable predicate. The information to use is the joined triple patterns with constant predicates, and we need to analyze the base data to see what are the predicates that are joined with the constant predicates.

F. SYNOPSIS UPDATE

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) [8]. 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. By our pruning method shown in Section 3.2, 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, with more and more deletions, this approach will lead to too many false positives (rewritings that do produce empty results cannot be detected by checking their synopses, and an ASK query has to be issued). Hence, we can periodically rebuild all synopses after seeing enough number of deletions w.r.t. a user-defined threshold.

Another solution to support flexible updates to RDF stores is to use the AKMV-synopsis, an extension of the KMV-synopsis that supports the deletion (in addition to the in-

sertion) [8]. In this case, we need to derive new estimation formulas based on the AKMV-synopsis for our purpose, which can be done in a similar fashion as what has been shown for the KMV-synopsis in Section 3.2. The drawback is that the AKMV-synopsis takes more space than the KMV-synopsis [8]. We omitted the details of this approach.

G. EXPERIMENTS

G.1 Translation to SQL

Corresponding to the distinct predicates in LUBM, we materialized 16 predicate tables in MySQL, which were summarized in figure 14. In addition, B-tree indexes were built on the subject and the object respectively in each table. Figure 15 showed a sample SQL query resulted from view expansion and its equivalent SPARQL query.

doctoralDegreeFrom (Sub, Obj)	worksFor (Sub, Obj)
headOf (Sub, Obj)	mastersDegreeFrom (Sub, Obj)
teacherOf (Sub, Obj)	telephone (Sub, Obj)
researchInterest (Sub, Obj)	name (Sub, Obj)
advisor (Sub, Obj)	undergradDegFrom (Sub, Obj)
takesCourse (Sub, Obj)	memberOf (Sub, Obj)
teachingAssistantOf (Sub, Obj)	publicationAuthor (Sub, Obj)
subOrganizationOf (Sub, Obj)	emailAddress (Sub, Obj)

Figure 14: Predicate Tables

```
SELECT name0.Sub FROM name name0, undergrad-
DegFrom UGDegFrom1, teacherOf teacherOf2, worksFor
worksFor3 WHERE UGDegFrom1.Sub = name0.Sub AND
teacherOf2.Sub = name0.Sub AND worksFor3.Sub
= name0.Sub AND worksFor3.Obj = '<Depart-
ment0.University0>'
```

(a) Sample SQL query resulted from view expansion.

```
SELECT ?x WHERE{ ?x <name> ?n. ?x <worksFor> <De-
partment0.University0>. ?x <teacherOf> ?c. ?x <un-
dergradDegFrom> ?d.}
```

(b) The equivalent SPARQL query.

Figure 15: An example in SQL rewriting.

G.2 Experimental Results from Jena TDB

Using the same query and view definitions, we have run exactly the same set of experiments in 4store from Section 4 on Jena TDB, to demonstrate the flexibility and the store-independent property of our algorithms. In general, the results from Jena TDB are highly consistent with our observations from 4store, that the various optimization techniques we introduced in OSQR have significantly improved the efficiency of the baseline approach, Algorithm SQR. In particular, the overall performance of OSQR is several orders of magnitude better than the SQR in the “Combining all optimizations” experiment in Jena TDB as well, as evident from Figure 16. In the natural setting with the more representative query and view definitions as discussed in the last experiment in Section 4, OSQR also consistently outperforms SQR in Jena TDB by more than one order of magnitude (Figure 17). These trends are highly consistent with what we have

already observed from their comparison in 4store (Figures 4 and 6 in Section 4).

The comparison for the individual optimization techniques (SQR vs. OSQR-M with merger in Figure 18, SQR vs. OSQR-P with pruning of empty results in Figure 19, and SQR vs. OSQR-R with reordering in Figure 20) in Jena TDB are also consistent with the observations from 4store, showing the benefit of the individual optimization components we have designed. The only exception is the running time comparison between SQR and OSQR-M in Jena TDB in Figure 18(b). We do not observe a significant difference between the two, in comparison to the result from 4store in Figure 8. We articulate that Jena TDB query engine implements some form of merging (which is not available in 4store) and therefore the performance of SQR and OSQR-M is similar in this case. Nevertheless, Jena TDB has neither detection of empty rewritings, nor does it re-order predicates for efficient pruning of empty rewritings, and therefore the remaining figures show identical trends with 4store.

The pleasant surprise by the difference between 4store and Jena TDB with regard to the merger optimization reinforces the importance of our store-independent design principle.

In summary, our ultimate solution for SPARQL query rewriting over multiple views, Algorithm OSQR, exhibits superior efficiency and robustness in practice. Our extensive experimental results from two popular RDF stores convincingly and consistently show that OSQR has good efficiency in practice, from rewriting to the end of query execution, over tens to hundred of views in RDF stores with 10 million triples.

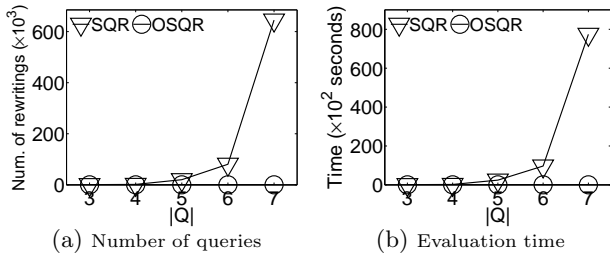


Figure 16: Combining All Optimizations, Jena TDB

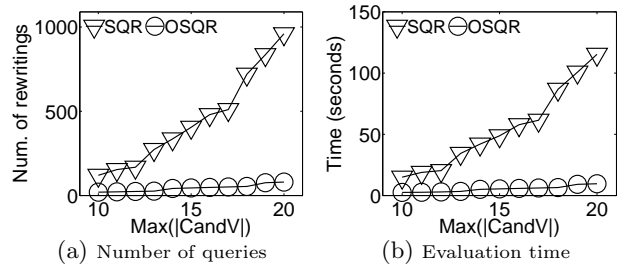


Figure 17: Performance in natural Settings, Jena TDB

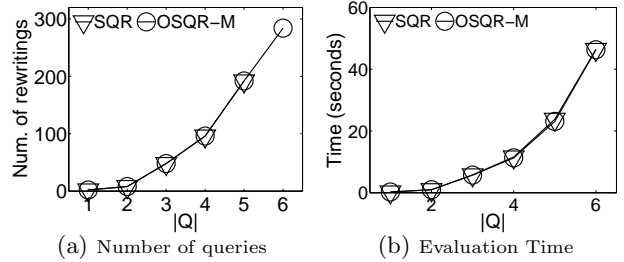


Figure 18: Optimizing Individual Rewritings, Jena TDB

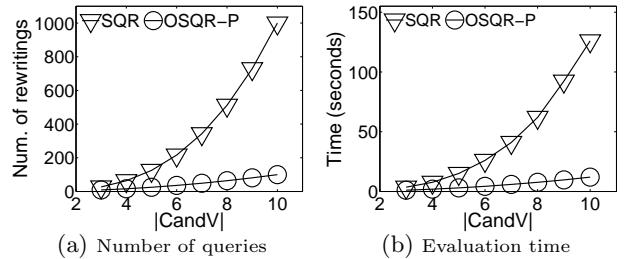


Figure 19: Pruning Rewritings with Empty Results, Jena TDB

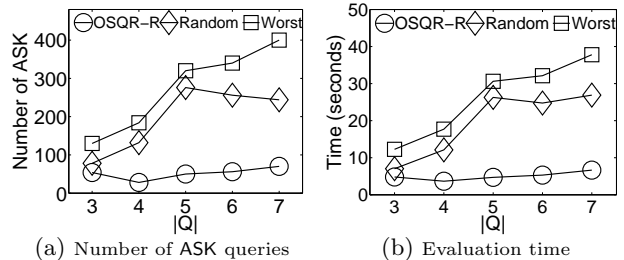


Figure 20: Optimizing the Generation of Rewritings, Jena TDB