Materialized View-Based Processing of RDF Queries

François Goasdoué[1] Konstantinos Karanasos[1] Julien Leblay[1,2]
Ioana Manolescu[1]

[1]Leo team, INRIA Saclay and LRI, Université Paris-Sud 11

[2] Université Paris 6

firstname.lastname@inria.fr

Abstract

The increasing interest in the RDF data model has turned the efficient processing of queries over RDF datasets to a challenging and crucial task. Indeed, the triple format of the RDF data model, along with the lack of structure that characterizes it, raise new challenges in data management both in terms of performance and scalability. In this paper, we consider improving the performance of RDF query evaluation by using materialized views. Starting from a workload of queries, we describe the space of possible views to materialize, we introduce ways for assessing the optimality of each view set and we propose practical algorithms for exploring the search space. When an RDF Schema is available, our algorithm takes advantage of it to guarantee the completeness of answering queries. We evaluate the efficiency of our search algorithms and demonstrate the benefits of the materialized views recommended by our algorithms on a fully implemented RDF querying platform.

Keywords RDF data management, view selection, materialized views, query optimization, RDFS.

1 Introduction

The growing interest in Semantic Web technologies and their many applications have attracted attention from the database research community, as witnessed by many highly visible recent works [1, 12, 13, 15, 18]. One of the core database problems considered in this area is the efficient evaluation of queries over RDF repositories [19]. The SPARQL query language [21] has been proposed as a W3C standard for querying RDF data, and is widely supported by RDF data management tools.

At a first look, querying RDF data bears some similarities with querying relational data using a language such as Datalog or SQL. Indeed, the core of SPARQL is made of conjunctive relational-style querying primitives, and SPARQL queries can be translated to SQL [7, 6]. However, there is a profound difference at the level of the data model, since an RDF dataset is fundamentally a collection of tuples of the form *(subject, property, object),* belonging conceptually to a single *triple* relation. This complicates querying, given that the *triple* table is as big as the whole dataset and therefore optimization is difficult, and has lead to the proposal of competing RDF indexing and storage strategies [1, 12, 13, 15, 18].

Another significant difference is that rich schema information can be attached to an RDF dataset, by means of an RDF Schema [20]. Notably, taking into account such extra information, when available, is crucial for the completeness of query evaluation.

In this work, we consider the efficient evaluation of conjunctive SPARQL queries against an RDF database, in the presence of an RDF Schema. More specifically, we investigate the usage of materialized views to speed up the processing of RDF queries. We make the following contributions:

1. We present an approach for enumerating all possible candidate view sets, based on an existing proposal for relational data and queries [17], and adapted to the particular setting of RDF.
2. Since the complexity of this approach is very high, we present a set of optimizations which reduce the time and memory needs, while preserving completeness. Moreover, we present a set of heuristics which lead to finding a candidate view set fast, without traversing the whole search space.
3. We show how our approach can exploit the presence of an RDF Schema to ensure the completeness of query evaluation.
4. We have fully implemented all our algorithms and experimented with RDF data and queries, using a relational database as a back-end. We assess the efficiency of our approach and demonstrate its practical interest in terms of speeding up query evaluation.

This paper is organized as follows. Section 2 presents our model for RDF queries and views. Section 3 presents the view selection problem as a search problem in a space of states among which one can move via transitions, studies its complexity, and describes some practical search strategies. Section 4 discusses the impact of an RDF Schema on the view selection problem. Section 5 presents our experimental evaluation. Section 6 discusses related works, then we conclude.

2 Problem Statement

RDF data consists of triples of the form *subject, property, object*, or (s, p, o) in short, where s stands for the *subject*, p for the *property* and o for the *object*. In accordance with the RDF data model, in each triple, s and p are URIs, while o can either be an URI or a literal. For simplicity, we use *constant* to refer to either an URI or a literal. Accordingly, at the logical level, we view an RDF dataset as a *triple table* t with three attributes, denoted $t(s, p, o)$.

We consider the conjunctive subset of SPARQL, which is noted here using the Datalog notation without any loss of information. Notably, we view RDF queries (and views) as a special case of conjunctive queries, i.e., conjunctions of atoms, the terms of which are either free variables (a.k.a. head variables), existential variables, or constants.

Definition 1 (RDF queries and views)**.** *An RDF query (or view) is a conjunctive query such that all atoms in the query body use the relation t, i.e., the triple table.*

Without loss of generality, we consider that our queries are *cartesian product-free*, that is, each triple shares at least one variable (i.e., joins) with another triple; a query featuring a cartesian product can be represented by the set of its independent sub-queries. Finally, we assume queries are *minimal*, i.e., the only containment mapping that can be found from the query to itself is the identity [4].

As a running example, we consider an RDF database about painters, their masterpieces, and the museums in which the latter are exposed. The following query q_1 returns the titles of the paintings of all painters, somehow related to Post-Impressionism, that are exposed in MoMA, as well as the painters' names (for the sake of readability namespaces are omitted):

$$\begin{aligned} q_1(X_2, X_5) :- \quad & t(X_1, hasName, X_2), t(X_1, X_3, postImpres), t(X_1, hasPainted, X_4), \\ & t(X_4, hasTitle, X_5), t(X_4, isExpIn, X_6), t(X_6, isNamed, moma) \end{aligned}$$

Similarly, the query q_2 returns the titles of the paintings made by French Post-Impressionism painters that are exposed in museums across Europe, as well as the names of these museums:

$$\begin{aligned} q_2(Y_3, Y_5) :- \quad & t(Y_1, hasCountry, france), t(Y_1, belongsTo, postImpres), \\ & t(Y_1, hasPainted, Y_2), t(Y_2, hasTitle, Y_3), t(Y_2, isExpIn, Y_4), \\ & t(Y_4, isNamed, Y_5), t(Y_4, isLocatIn, europe) \end{aligned}$$

We can now define a rewriting in our context.

Definition 2 (Rewriting)**.** *Let q be an RDF query and $v_1, v_2, \ldots, v_k$ be a set of RDF views. A rewriting of q based on $v_1, v_2, \ldots, v_k$ is a conjunctive query (i) equivalent to q (i.e., it yields the same answers for any dataset) and (ii) whose body only refers to the relations $v_1, v_2, \ldots, v_k$.*

We consider a *quality function* qf which, for every rewriting, returns a quantitative measure of how good the rewriting is. Typical quality functions take into account the effort to evaluate the body of the rewriting, the total space occupancy of the views, the costs to maintain the views as data changes, etc.

We require quality functions to *favor compact views and cheap rewritings*, which we formalize as follows. Let r' and r'' be two rewritings of the same query q.

1. If r' and r'' incur the same effort in answering q, but r' requires less space to store the views, then $\mathrm{qf}(r') > \mathrm{qf}(r'')$.
2. If r' and r'' require the same space to store the views, but the effort to evaluate r' is smaller than the effort to evaluate r'', then $\mathrm{qf}(r') > \mathrm{qf}(r'')$.

We are now ready to define our view selection problem. For ease of exposition, we do it *without* taking into account RDF schemas. Section 4 will show how to integrate them in our setting via a preliminary reasoning step.

Definition 3 (Candidate view set)**.** *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of RDF queries. A candidate view set for Q is a pair (V, R) such that:*

- *V is a set of RDF views $\{v_1, v_2, \ldots, v_k\}$ and*
- *R is and a set of rewritings $\{r_1, r_2, \ldots, r_n\}$, where for each $1 \leq i \leq n$, r_i is an equivalent rewriting of q_i based on the views in V.*

Definition 4 ((Schema-agnostic) view enumeration and selection problems)**.** *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of RDF queries and $W = \{w_1, w_2, \ldots, w_n\}$ be a set of weights associated to the queries in Q. Let qf be a quality function.*

- *The view enumeration problem consists of finding all candidate view sets for Q.*
- *The view selection problem consists of finding a candidate view set (V, R) for Q such that, for any other candidate view set (V', R') for Q, $\Sigma_{r \in R}(w_i \cdot \mathrm{qf}(r_i)) \geq \Sigma_{r' \in R'}(w_i \cdot \mathrm{qf}(r'_i))$.*

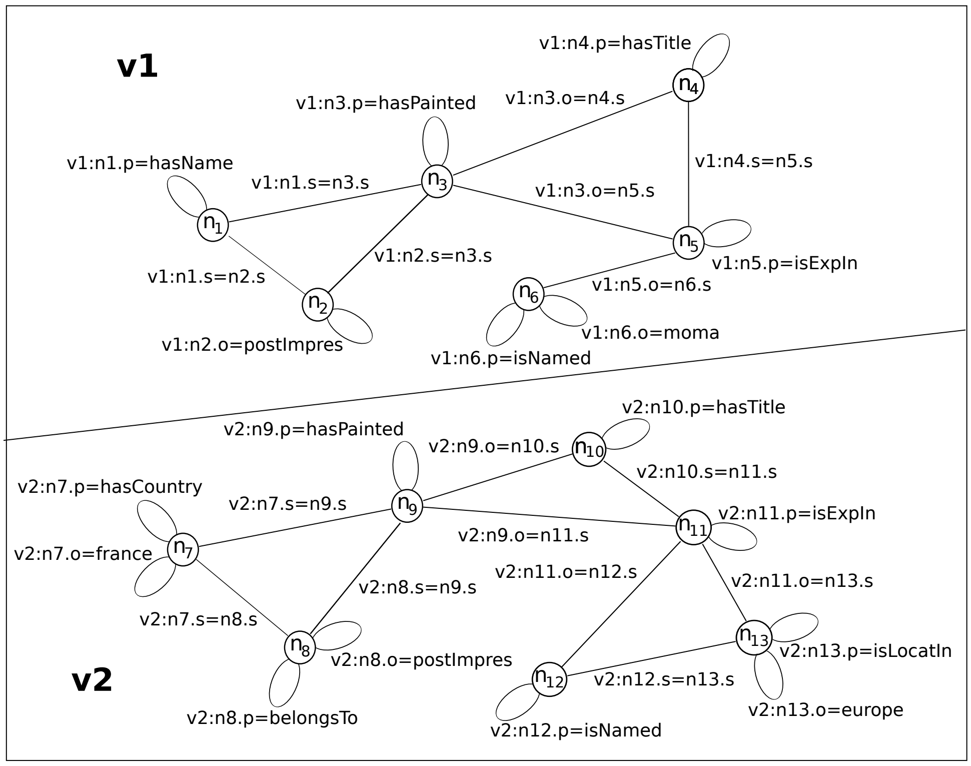

Figure 1: Sample graph of a state.

3 The View Selection Search Problem

This Section describes our approach for solving the RDF view selection and enumeration problems. Section 3.1 describes our modeling of the problem in terms of states and transitions. Section 3.2 characterizes search strategies, the size of the search space, and search complexity. The complexity of the problem is high, and may become prohibitive for large workloads. Therefore, Section 3.3 presents a set of interesting strategies, along with heuristics which allow to limit the search.

3.1 Model: States and Transitions

Solving the view selection problem requires finding both a set of views and the corresponding rewritings of the workload queries based on these views. Among the solutions considered in the literature for view selection in relational databases, we adapt the approach of [17] to our RDF context. More specifically, we model view set selection as a search problem. Each point in the search space, called a *state*, represents a set of views and a set of associated equivalent rewritings for all the workload queries.

Definition 5 (State). *Given the query set $Q = \{q_1, q_2, \dots, q_n\}$, a state is a 3-tuple $S_i(Q) = \langle V_i, G_i, R_i \rangle$ such that:*

- *V_i is a set of views proposed for materialization,*
- *$G_i = (N_i, E_i)$ is an undirected multigraph, whose node set is N_i and whose edge set is E_i, and*
- *R_i is a set of rewritings showing how to compute each query from Q based on the view set V_i.*

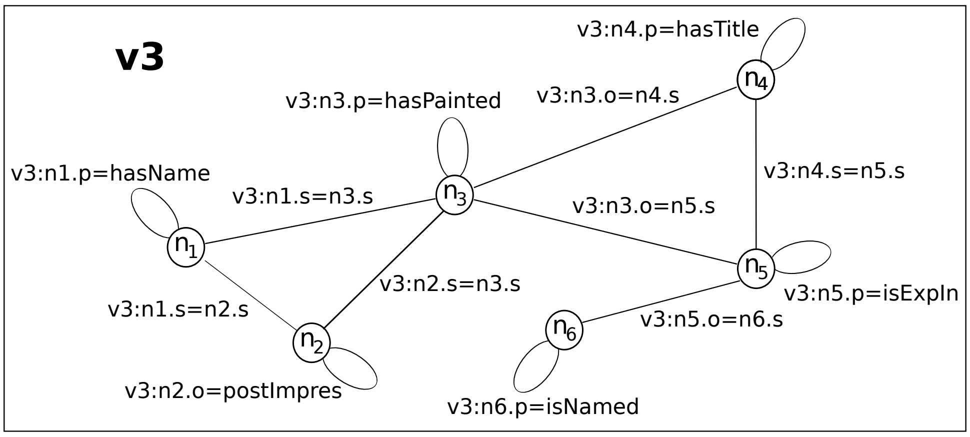

Figure 2: State graph after a selection cut.

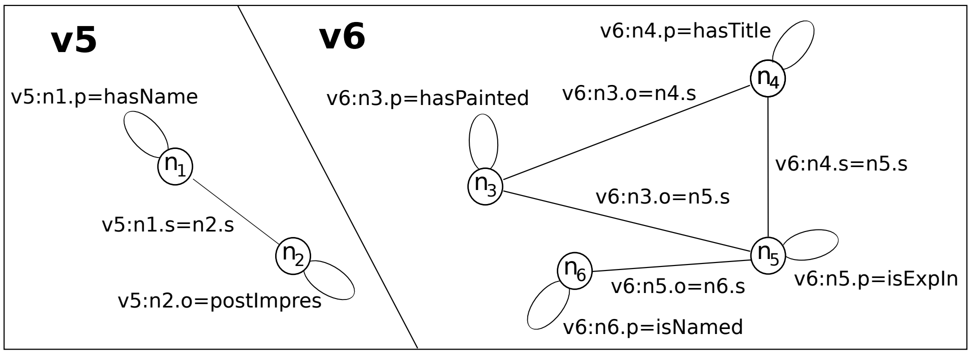

Figure 3: State graph after two join cuts.

Within G_i, each triple t_i appearing in a view $v \in V_i$ is represented by a node $n_i \in N_i$.

Moreover, let t_i and t_j be two triples in a view $v \in V_i$, and a join on their attributes $t_i.a_i$ and $t_j.a_j$ (where $a_i, a_j \in \{s, p, o\}$). For each such join, there is an edge $e_i \in E_i$ connecting the respective nodes $n_i, n_j \in N_i$ and labeled $v{:}n_i.a_i = n_j.a_j$. We call e_j a join edge.

Finally, let t_i be a triple in a view $v \in V_i$ and $n_i \in N_i$ be its corresponding node. For every constant, having a value c_i, that appears in the attribute $a_i \in \{s, p, o\}$ of t_i, an edge connecting n_i to itself is present, labeled $v{:}n_i.a_i = c_i$. These are called selection edges*, as they represent a selection over the triple table t.*

We now make the following observation: *ignoring views with cartesian products does not compromise optimality*. The proof can be sketched as follows. Let v be a view with a cartesian product, having two connected sub-queries, $q \in Q$ be a query and r be a rewriting of q based on v. Since q is cartesian product-free, r must enforce a selection on the two connected sub-queries of v in order to join them. Instead of materializing v, we can materialize v' which is the same as v but having the join imposed by the query in the place of the cartesian product of v. Using v' is at least as quick as using v, because a scan over a possibly smaller table is sufficient to answer q. As for the space needed, though, if many other rewritings for the queries of the workload use v, each imposing a different join, one could think that materializing v would save space, as it will be used in many rewritings. However, we could again materialize the views (like v') that have the join needed for each query, instead of v. In the worst case, we will materialize all possible joins for v. Even in this case, the total space occupied by all these views will be at most equal to the space occupied by the materialization of the cartesian product of v.

We define the *graph of* v as the connected component of G_i corresponding to v. Observe that in a view, two nodes may be connected by a multiple (join) edge if their corresponding

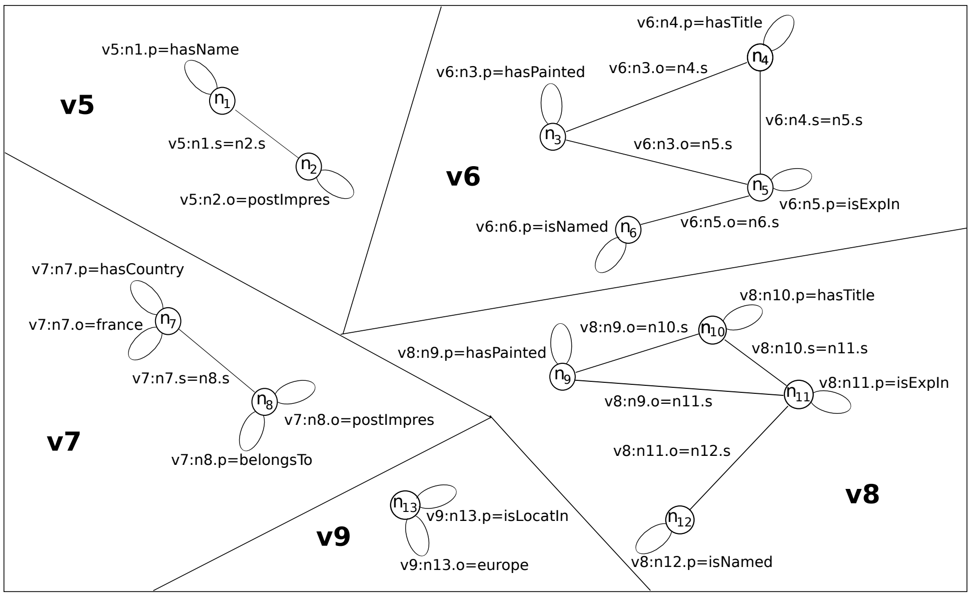

Figure 4: State graph after some selection and join cuts.

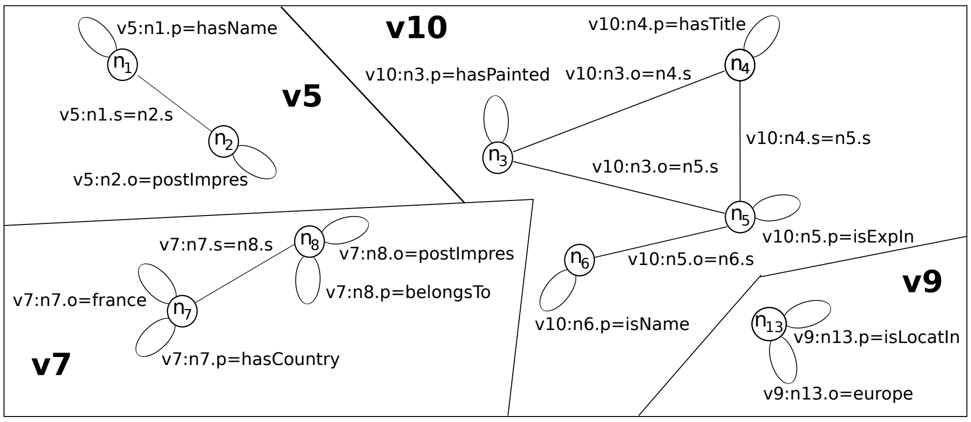

Figure 5: State graph after a view fusion.

patterns are connected by more than one join predicates.

As an example, we consider $S_0(Q) = \langle\{v_1, v_2\}, G_0, R_0\rangle$, where $Q = \{q_1, q_2\}$ are the two sample queries previously introduced, $v_1 = q_1$ and $v_2 = q_2$. Thus, the rewriting set R consists of the trivial rewritings $\{q_1 = v_1, q_2 = v_2\}$. The graph G_0 is depicted in Figure 1. The graph of v_1 is the connected component in the upper half of the Figure.

We define the *initial state* of the search as $S_0(Q) = \langle V_0, G_0, R_0\rangle$ for a given set of queries. In S_0, $V_0 = Q$, i.e., the set of views is exactly the set of queries. The rewritings in R_0 are all trivial (view scans), and the initial graph G_0 corresponds to the queries in Q. Clearly, the rewriting costs associated to the initial state are very low, since for each query it suffices to scan the pre-computed results. However, this solution is typically not the best-quality one, since its space consumption or view maintenance costs may be quite high.

We now introduce a set of three elementary state transformations, adapted from [17]. In the following, we will use v:e to denote the edge e belonging to the view v in a state graph. Moreover, we will use σ_e to denote a selection on the condition attached to the edge e in the graph. Since the query set Q remains the same across all transformations, we omit it for readability.

Definition 6 (Selection Cut (**SC**)). *Let $\langle V, G, R\rangle$ be a state and v:e be a selection edge in G. A* selection cut *on e yields a state $\langle V', G', R'\rangle$ such that:*

- *V' is obtained from V by replacing v with a new view v', in which the constant of the selection has been replaced with a fresh head variable (i.e., is returned by v', along with the variables returned by v),*
- *G' is obtained from G by erasing the edge e, and*
- *R' is obtained from R by replacing all occurrences of v with the expression $\pi_{head(v)}\sigma_e(v')$.*

For example, consider the initial state $S_0 = \langle\{v_1, v_2\}, G_0, \{q_1 = v_1, q_2 = v_2\}\rangle$ where v_1 and v_2 are views identical to the queries q_1 and q_2 introduced in the previous Section. G_0 is the graph depicted in Figure 1. We apply a selection cut on the edge labeled v_1:$n_6.o = moma$ and obtain the query graph G_1, in which v_1 is replaced by a new view v_3, the graph of which is depicted in Figure 2. We don't show the whole G_1 in the Figure, as the graph of v_2 remains unchanged. The resulting state is:

$$S_1 = \langle\{v_2, v_3\}, G_1, \{q_1 = \pi_{head(v_1)}(\sigma_{n6.o=moma}(v_3)), q_2 = v_2\}\rangle.$$

Definition 7 (Join Cut (**JC**)). *Let $\langle V, G, R\rangle$ be a state and v:e be a join edge in G. A* join cut *on e yields a state $\langle V', G', R'\rangle$, obtained as follows:*

1. *If the graph of v is still connected after the cut, V' is obtained from V by replacing v with a new symbol v' in which the variable corresponding to the join edge e becomes a head variable. Moreover, in every rewriting, the symbol v is replaced by $\pi_{head(v)}(\sigma_e(v'))$.*
2. *If the graph of v is split in two independent components, V' is obtained from V by replacing v with two new symbols v'_1 and v'_2, each corresponding to one component. In each of v'_1 and v'_2, the join variable of e becomes a head variable. The new rewriting set R' is obtained from R by replacing v by $\pi_{head(v)}(v'_1 \bowtie_e v'_2)$.*

The new graph G' is obtained from G by erasing the edge v:e.

For example, consider cutting the join edge v_3:$n_1.s = n_3.s$ on the result (shown in Figure 2) of the previous selection cut. This operation does not disconnect the graph of v_3, but replaces v_3 with a new view v_4 such that $v_3 = \pi_{head(v_3)}(\sigma_{n_1.s=n_3.s}(v_4))$. The resulting state is:

$$S_2 = \langle\{v_2, v_4\}, G_2, \{q_1 = \pi_{head(v_1)}(\sigma_{n6.o=moma}(\pi_{head(v_3)}(\sigma_{n_1.s=n_3.s}(v_4)))), q_2 = v_2\}\rangle$$

If we now cut the join edge labeled v_4:$n_2.s = n_3.s$, the graph of v_4 becomes disconnected, resulting in the introduction of the views v_5 and v_6, depicted in Figure 3. The view symbol v_4 is replaced in the rewritings by the expression $\pi_{head(v_4)}(v_5 \bowtie_{n_2.s=n_3.s} v_6)$. Denoting the resulting graph by G_3, the resulting state is:

$$S_3 = \langle\{v_2, v_5, v_6\}, G_3, \{q_1 = \pi_{head(v_1)}(\sigma_{n6.o=moma}(\pi_{head(v_3)}(\sigma_{n_1.s=n_3.s}(\pi_{head(v_4)}(v_5 \bowtie_{n_2.s=n_3.s} v_6))))), q_2 = v_2\}\rangle.$$

Selection and join cuts tend to make the views less specific and more voluminous, and rewritings potentially more expensive. Thus, their interest lies not in the states they produce, but in the factorization transformations which can be applied after some edge cuts. Such factorization is achieved by the following transformation.

$Q = \{q\}$

$q(Y, Z) :-$
$t(X, Y, c_1),$
$t(X, Z, c_2)$

$$\begin{tikzcd}
 & S_2 \arrow[r, "\text{JC}"] \arrow[dr, "\text{SC}(c_2)"] & S_5 \arrow[dr, "\text{SC}(c_2)"] & & \\
S_0 \arrow[ur, "\text{SC}(c_1)"] \arrow[r, "\text{JC}"] \arrow[dr, "\text{SC}(c_2)"'] & S_1 \arrow[ur, "\text{SC}(c_1)"] \arrow[dr, "\text{SC}(c_2)"] & S_4 \arrow[r, "\text{JC}"] & S_7 \arrow[r, "\text{VF}"] & S_8 \\
 & S_3 \arrow[r, "\text{JC}"] \arrow[ur, "\text{SC}(c_1)"] & S_6 \arrow[ur, "\text{SC}(c_1)"] & &
\end{tikzcd}$$

V_0	$\{q(Y, Z) :- t(X, Y, c_1), t(X, Z, c_2)\}$
V_1	$\{q_1(X_1, Y) :- t(X_1, Y, c_1); \quad q_2(X_2, Z) :- t(X_2, Z, c_2)\}$
V_2	$\{q(Y, Z, W_1) :- t(X, Y, W_1), t(X, Z, c_2)\}$
V_3	$\{q(Y, Z, W_2) :- t(X, Y, c_1), t(X, Z, W_2)\}$
V_4	$\{q(Y, Z, W_1, W_2) :- t(X, Y, W_1), t(X, Z, W_2)\}$
V_5	$\{q_1(X_1, Z, W_1) :- t(X_1, Z, W_1); \quad q_2(X_2, Z) :- t(X_2, Z, c_2)\}$
V_6	$\{q_1(X_1, Z) :- t(X_1, Z, c_1); \quad q_2(X_2, Z, W_2) :- t(X_2, Z, W_2)\}$
V_7	$\{q_1(X_1, Z, W_1) :- t(X_1, Z, W_1); \quad q_2(X_2, Z, W_2) :- t(X_2, Z, W_2)\}$
V_8	$\{q(X, Y, Z) :- t(X, Y, Z)\}$

Figure 6: Sample complete search space, and view sets corresponding to each state.

$Q = \{q_1, q_2\}$
$q_1(X) :- (X, Y, c)$
$q_2(X) :- (X, Z, c)$

$$\begin{tikzcd}
 & S_1 \arrow[dr, "\text{SC}"] & & \\
S_0 \arrow[ur, "\text{SC}"] \arrow[r, "\text{VF}"] \arrow[dr, "\text{SC}"'] & S_3 \arrow[rr, bend left, "\text{SC}"] & S_4 \arrow[r, "\text{VF}"] & S_5 \\
 & S_2 \arrow[ur, "\text{SC}"'] & &
\end{tikzcd}$$

Figure 7: Sample complete search space for a simple workload of two queries.

Definition 8 (View Fusion (**VF**)). *Let $\langle V, G, R\rangle$ be a state and v_1, v_2 be two views in V such that their respective graphs (thus, the bodies of the expressions defining the views) are isomorphic. Let v_3 be a copy of v_1, whose head variables are those of v_1* plus *those of v_2 (up to the isomorphism). Fusing v_1 and v_2 leads to a new state $\langle V', G', R'\rangle$ obtained as follows:*

- $V' = (V \setminus \{v_1, v_2\}) \cup \{v_3\}$,
- *G' is obtained from G by removing the graphs of v_1 and v_2 and adding that of v_3, and*
- *R' is obtained from R by replacing any occurrence of v_1 (respectively v_2), with $\pi_{head(v_1)}(v_3)$ (respectively, $\pi_{head(v_2)}(v_3)$).*

Performing a set of join cuts in graph G_3 presented above, we can arrive at state containing the graph depicted in Figure 4. The graphs of v_6 and v_8 are isomorphic, enabling us to fuse them. The resulting graph is depicted in Figure 5.

Sample complete search spaces are shown in Figure 6 for a workload of one query, and Figure 7 for a workload of two queries.

3.2 Search Space and Strategies

We denote by $S \xrightarrow{\tau} S'$ the application of the transformation $\tau \in \{$**SC**,**JC**,**VF**$\}$ on a state S, which leads to the state S'.

Definition 9. *(Strategy) For a given view selection problem, a search strategy Σ is a sequence of transformations of the form:*

$$\Sigma = (S_{i_1} \xrightarrow{\tau_{i_1}} S'_{i_1}), (S_{i_2} \xrightarrow{\tau_{i_2}} S'_{i_2}), \ldots, (S_{i_{k-1}} \xrightarrow{\tau_{i_{k-1}}} S'_{i_{k-1}}), (S_{i_k} \xrightarrow{\tau_{i_k}} S'_{i_k})$$

*where $S_{i_1} = S_0$, for every $j \in [1..k]$ $\tau_{i_j} \in \{$**SC,JC,VF**$\}$, and for every $j \in [2..k]$ there exists $l < j$ such that $S'_{i_l} = S_{i_j}$ (i.e., each state must be obtained before it is transformed).*

For example, on the problem illustrated in Figure 6, two possible strategies are:

$$\Sigma_1 = (S_0 \xrightarrow{SC(c_1)} S_2), (S_2 \xrightarrow{SC(c_2)} S_4), (S_0 \xrightarrow{SC(c_2)} S_3), (S_3 \xrightarrow{SC(c_1)} S_4), (S_0 \xrightarrow{JC} S_1)$$
$$\Sigma_2 = (S_0 \xrightarrow{SC(c_1)} S_2), (S_2 \xrightarrow{JC} S_5), (S_0 \xrightarrow{SC(c_2)} S_3), (S_3 \xrightarrow{JC} S_6)$$

A strategy is *exhaustive* if it reaches all candidate view sets. A set of simple exhaustive strategies can be immediately defined as follows. An **ex-Naïve** (exhaustive naïve) strategy starts by applying a transformation to S_0, leading to a new state S'_0. Then, **ex-Naïve** randomly picks one of the existing states and one transformation which applies to the state, performs the transformation, and adds the new state to the existing ones. **Ex-Naïve** stops when no new states can be found.

Theorem 1 (Completeness)**.** *Any **ex-Naïve** strategy enumerates all the candidate view sets.*

The proof follows from its counterpart in [17]. A sample **ex-Naïve** strategy on the example in Figure 6 is:

$$\Sigma_3 = (S_0, S_2), (S_0, S_1), (S_0, S_3), (S_2, S_5), (S_2, S_4), (S_1, S_5), (S_1, S_6), (S_3, S_6), (S_3, S_4), (S_4, S_7), (S_5, S_7), (S_6, S_7), (S_7, S_8)$$

where for readability, we omitted the transition details. Observe that **ex-Naïve** may lead to reaching the same state several times, e.g., S_5 is reached twice by Σ_3.

As reaching the same state more than once through different search *paths* is a source of inefficiency, we focus on a particular class of *stratified strategies* which suffer less from this shortcoming.

Definition 10. *(Paths to a state) Let Σ be a strategy, S be a state in Σ, and j be an integer such that $(S_{i_j}, S'_{i_j}) \in \Sigma$ and $S'_{i_j} = S$. The set of paths leading to S in Σ, denoted $\hookrightarrow S$, is defined as: $\hookrightarrow S = \{\tau_{i_1}\}$ if $j = 1$, and $\hookrightarrow S = \{p\,\tau_{i_j} \mid p \in \hookrightarrow S_{i_j}\}$ otherwise.*

Definition 11. *(Stratified strategy) A strategy Σ is stratified iff for any state $S \in \Sigma$ and any path $p \in \hookrightarrow S$, p belongs to the regular language:* ***SC* JC* VF****.

Intuitively, in a stratified strategy, *on a given path*, selection cuts precede all join cuts, which precede all view fusions. For instance, on the sample workload previously used in Figure 6, a possible stratified strategy is:

$$\Sigma_4 = (S_0, S_2), (S_0, S_1), (S_0, S_3), (S_2, S_5), (S_2, S_4), (S_3, S_6), (S_3, S_4), (S_4, S_7), (S_7, S_8).$$

Observe that in a stratified strategy, for instance, selection cuts may still appear after join edge cuts, as long as they are *on different search paths*.

Figure 8 depicts the search space corresponding to the workload in Figure 6, just with the transitions of Σ_4. Note that the **ex-Naïve** strategy Σ_3 and the stratified Σ_4 reach the same states.

The following result states that stratified strategies can mimic any strategy.

$Q = \{q\}$

$q(Y,Z) :-$
$t(X,Y,c_1),$
$t(X,Z,c_2)$

$$\begin{tikzcd}
 & S_2 \arrow[r, "\text{JC}"] \arrow[dr, "\text{SC}(c_2)"'] & S_5 & & \\
S_0 \arrow[ur, "\text{SC}(c_1)"] \arrow[r, "\text{JC}"] \arrow[dr, "\text{SC}(c_2)"'] & S_1 & S_4 \arrow[r, "\text{JC}"] & S_7 \arrow[r, "\text{VF}"] & S_8 \\
 & S_3 \arrow[r, "\text{JC}"] \arrow[ur, "\text{SC}(c_1)"] & S_6 & &
\end{tikzcd}$$

Figure 8: Stratified search strategy for the example in Figure 6.

Theorem 2. *(Stratified equivalent strategy) For any strategy Σ, there exists a stratified strategy, denoted Σ^{str}, producing the same states.*

Due to space limitations, the proof is delegated to our technical report, available at http://www-rocq.inria.fr/~karanaso/files/RDFVS-TecReport.pdf.

We focus now on a stratified subset of the **ex-Naïve** strategies, namely the **ex-STR** (exhaustive stratified) strategies, such that:

- For any two transitions $(S_j \xrightarrow{\textbf{SC}} S'_j)$, $(S_l \xrightarrow{\textbf{JC}} S'_l) \in \Sigma$, we have $j < l$. In other words, *any* selection cut is made before *all* join cuts.
- For any two transitions $(S_j \xrightarrow{\textbf{JC}} S'_j)$, $(S_l \xrightarrow{\textbf{VF}} S'_l) \in \Sigma$, we have $j < l$. In other words, *any* join cut is made before *all* view fusions.

Strictly speaking, for a given problem, there are many **ex-STR** strategies (depending on relative transition order in the strategy), but we will simply use **ex-STR** to refer to any of them.

A corollary of Theorems 1 and 2 is the completeness of **ex-STR**, since it can mimic **ex-Naïve**, thus enumerates all the candidate view sets. As a result, from now on and without loss of generality, we will focus on stratified strategies only. We quantify the size of our search space by counting the states produced by **ex-STR**.

Search space size. Let $S_0(Q) = \langle V_0, G_0, R_0 \rangle$ be an initial state, where $G_0 = (N_0, E_0)$. Let $|N_0| = n$, $|E_0| = m$ be the number of nodes and edges of G_0 respectively. Let $|V_0| = |Q| = k$ be the number of views in S_0, m_s be the number of selection edges and m_j be the number of join edges. Obviously, $m_s + m_j = m$.

1. From S_0, repeatedly applying **SC** can lead to exactly 2^{m_s} states.
2. Let S_i be one of these states. Repeatedly applying **JC** on S_i can lead to creating 2^{m_j} states $\{S_i^0, S_i^1, \ldots, S_i^{2^{m_j}}\}$, which are all distinct in the worst case. Thus, selection cuts followed by join cuts leads to creating at most $2^{m_s} * 2^{m_j} = 2^m$ distinct states.
3. Each of the states thus obtained has at least k and at most $k + m_j$ views. The state has k views if no **JC** has been applied or if none of the **JC** disconnected a view. Otherwise, each **JC** may have added one extra view and the maximum number of **JC** is m_j.
4. We now consider one of the states S_j obtained by edge cuts and evaluate the number of states which may result from S_j by repeatedly applying **VF**. In the worst case, any subset of the views in V_j consists of isomorphic views. Thus, the number of states resulting

from S_j by applying **VF** is bound by the number of partitions of the set V_j, whose size is at most $k + m_j$. Denoting by B_x the Bell number (the number of partitions of a set of size x), an upper bound for the total number of distinct states is:

$$N_S = 2^m \cdot B_{k+m_j}.$$

In the particular case of RDF queries, the number of selections is between 0 and $3n$, and the number of joins m_j is between 0 and $\frac{n(n-1)}{2}$ (the latter occurs when G_0 is the complete graph). Replacing these bounds in the above, we obtain:

$$N_s = 2^{3n+n(n-1)/2} \cdot B_{k+n(n-1)/2}.$$

Time complexity. The time complexity of performing an exhaustive search can be similarly derived, based on the number of states created by each transformation and the time complexity of the transformation:

- The cost of a selection or join cut is linear in the size of the largest view, which is bound by $3n$.
- View fusion requires checking query equivalence, which is in $O(2^n)$ [4].

The complexity is well above exponential, making exhaustive search unfeasible for large workloads. This highlights the need for robust implementations and efficient heuristics.

3.3 Optimizations and heuristics

We now discuss a set of search strategies with interesting properties, as well as a set of pruning heuristics which may be used to trade off completeness for efficiency of the search. For our discussion, we assume all strategies use the following simple data structures

CS the *candidate state set*. This set is initially $\{S_0\}$. As new states are created, they are added to CS.

ES the set of *explored* states. ES is disjoint from CS and is initially empty. An explored state S is a state such that for any transformation $\tau \in \{$**SC**, **JC**, **VF**$\}$, the state $S' = \tau(S)$ obtained by applying τ on S already belongs either to CS or to ES.

S_b is at any point in the search, the *best state* explored so far. S_b is initially set to S_0 and is changed to S whenever the newly found state S satisfies $\mathrm{qf}(S) \geq \mathrm{qf}(S_b)$.

DFS (depth-first search) strategies A (stratified) strategy Σ is depth-first iff the order of Σ's transitions satisfies the following constraint. Let S be a state reached by a path p of the form **SC***. Immediately after reaching S for the first time, Σ enumerates all transitions recursively attainable from S by **JC** only. After these transitions, Σ immediately enumerates all transitions recursively attainable from the states previously obtained, using **VF** only. For instance, on the example in Figure 6, the following strategy Σ_5 is DFS:

$$\Sigma_5 = (S_0, S_2), (S_2, S_5), (S_0, S_3), (S_3, S_6)$$

An advantage of DFS strategies is that they fully explore (apply all possible **SC** and **JC** on) each state that is reached. *A fully explored state does not need to be kept (in memory) and can be discarded without compromising the search* (this amounts to not maintaining the ES set altogether, and just discarding any explored state that is not S_b). Thus, DFS reduces the memory footprint of the search.

Aggressive fusion (or eager-fuse). This technique can be included in many strategies: when a new state S is reached, it is substituted with the state attained after repeatedly applying on S all possible **VF**. The motivation here is that **VF** is guaranteed to improve the estimated quality of the state (whereas **SC** and **JC** decrease it): instead of materializing two views, we materialize only one (i.e., we save space) and, at the same time, the rewritings (thus the query evaluation efforts) remain unchanged. Moreover, the number of explored states is reduced, because when we repeatedly apply **VF** we keep only the finally reached state and discard the intermediate ones, which are worse w.r.t., quality. For instance, on the example in Figure 7, eager-fuse would first go from S_0 to S_3, and only afterward explore the other states. S_3 is the best state, and eager-fuse helps find it fast. Thus, even if the search is stopped early, the best solution S_b is likely to be quite good.

Stop conditions. We use some *stop conditions* to limit the search by declaring that some states are not promising, thus should not be explored. Clearly, stop conditions lead to non-exhaustive search. We have considered the following stop conditions for a state S.

- $stop_t(S)$: true if a view in S is the full triple table t
- $stop_{var}(S)$: true if a view in S has only variables. The idea is that we reject S since we consider its space occupancy to be too high. In general, this condition may be satisfied even by the initial state S_0, if one of the queries mentions no URI and no constants, and in this case, $stop_{var}$ would prevent *any* search. However, no query in the frequently-used RDF benchmarks satisfies this condition, therefore, $stop_{var}$ can be used in many settings to restrict the search while leaving many meaningful options.
- $stop_{NV}(S)$: true if the number of views in S exceeds a bound NV. This tends to discourage over-aggressive join edge cuts, based on the idea that joins are expensive operations and if they can be pre-computed by the views, that results in significant effort savings.

"Pull&push constants" technique This technique attempts to "smartly guess" which selection edges can be cut and which should be preserved. It orders all constants from the workload, according to their number of occurrences. The more frequently a constant appears, the more likely it is to appear in the selected view state, because it represents a selective, shared condition. Thus, prior to any search, we start by cutting *all* selection edges corresponding to constants appearing one or a few times ("pull constants" part). If this pre-processing removes l selection edges, this diminishes the search space by a significant factor of 2^l, given that the subsequent search (regardless of its strategy) will be applied on an initial CS of just one state (that obtained from S_0 by the l successive **SC**s). *After* the search has finished, however, we may need to "push" back some of the selections cut in the "pull" stage. This is the case if, for some recommended view v, *all* rewritings using v apply the same selection on v, corresponding to a constant eagerly removed by the "pull".

The interest of this technique is to reduce the size of the search space by "betting against some selection edges". However, it may compromise optimality, given that the comparisons performed during the search ignore the fact that some selections may be brought back by the

Semantic relationship	RDFS notation	FOL semantics
Class inclusion	C_1 rdfs : $\mathrm{subClassOf}$ C_2	$\forall X(C_1(X) \Rightarrow C_2(X))$
Property inclusion	P_1 rdfs : $\mathrm{subPropertyOf}$ P_2	$\forall X \forall Y(P_1(X,Y) \Rightarrow P_2(X,Y))$
Domain typing of a property	P rdfs : domain C	$\forall X \forall Y(P(X,Y) \Rightarrow C(X))$
Range typing of a property	P rdfs : range C	$\forall X \forall Y(P(X,Y) \Rightarrow C(Y))$

Figure 9: Semantic relationships allowed between classes and properties.

post-processing. We have implemented this technique parameterized by w, the maximum number of appearances of constants eliminated by the "pull" stage. However, we found that for w values greater than 1, the loss of optimality may be significant, while for $w = 1$ it tends to be much smaller. Thus, we applied this technique with $w = 1$.

4 RDF Schemas

RDF data is first and foremost a format for *semantic* Web data, and an RDF Schema [20] (RDFS, for short) can be used to enhance the meaning of an RDF data set. RDFS allows defining semantic relationships between the classes and properties used for resource descriptions. Such relations are stated using the RDFS standard properties $\mathrm{subClassOf}$, $\mathrm{subPropertyOf}$, domain, and range. The first order logic semantics of those properties is given in Figure 9.

A direct consequence of using the above properties is the need for extra reasoning capabilities in order to answer queries. Indeed, an RDF dataset together with an RDF Schema entail *implicit* triples that are not explicitly present in that dataset. As a result, standard (database-style) query evaluation techniques may be incomplete on an RDF dataset, if they don't take into account the implicit triples. Two main approaches have been devised for this problem.

A first solution is to compile the knowledge of the schema into the dataset. This is done by a closure mechanism that adds to the dataset all the implicit triples. Such a mechanism is described in the RDF recommendation and has been implemented in various RDF frameworks like Jena[1]. Standard query evaluation techniques can then apply on the resulting closure, but that approach consumes space and is not robust to further updates on the original dataset.

The second solution is to compile the knowledge of the schema into the queries. This is done by query *reformulation*, which transforms a (conjunctive) query into a union of (conjunctive) queries, all the answers of which can be obtained by standard query evaluation techniques for plain RDF [2]. While that approach does not modify the original dataset, it introduces at query time a query reformulation overhead. Algorithm 1 outlines a (simplified) query reformulation algorithm. It is a (syntactic) adaptation of the one described in [2] to our conjunctive (triple) queries. We assume that the $\mathrm{evaluate}$ and $\mathrm{closure}$ functions used in Algorithm 1 provide respectively the evaluation of a query in plain RDF and of the closure of a dataset with respect to a schema, as defined in the RDF recommendation [19]. Moreover, $q_{[g/g']}$ denotes the result of replacing the atom g of the query q by the atom g'.

View selection in the presence of schemas We now explain how to extend our view selection algorithms, in order to attain the completeness guarantees of the reformulation-based approach. To that effect, we extend our rewriting language to that of *unions* of conjunctive queries and the definition of our *initial* state. More precisely, given a set of queries $Q = \{q_1, \ldots, q_n\}$, and assuming that $\mathrm{rewrite}(q_i, \mathcal{S}) = \{q_i^1, \ldots, q_i^{n_i}\}$, it is sufficient to define $S_0(Q) = \langle V_0, G_0, R_0 \rangle$

[1]http://jena.sourceforge.net/

Algorithm 1: the $\mathbf{Reformulate}$ algorithm
$\mathbf{Reformulate}(q, \mathcal{S})$
Input: an RDF schema $\mathcal{S}$ and a conjunctive query q over $\mathcal{S}$
Output: a union of conjunctive queries ucq such that for any dataset D $\text{evaluate}(q, \text{closure}(D, \mathcal{S})) = \text{evaluate}(ucq, D)$

```
(1)  $ucq \leftarrow \{q\}, ucq' \leftarrow \emptyset$
(2)  while $ucq \neq ucq'$
(3)     $ucq' \leftarrow ucq$
(4)     foreach rewriting $r \in ucq'$
(5)        foreach atom $g$ in $r$
(6)           if $g = t(s, rdf:type, C_1)$ and $C_2$ $\text{rdfs:subClassOf}$ $C_1 \in \mathcal{S}$
(7)              $ucq \leftarrow ucq \cup \{r_{[g/t(s,rdf:type,C_2)]}\}$
(8)           else if $g = t(s, rdf:type, C)$ and $P$ $\text{rdfs:domain}$ $C \in \mathcal{S}$
(9)              $ucq \leftarrow ucq \cup \{r_{[g/\exists y\ t(s,P,y)]}\}$
(10)          else if $g = t(s, rdf:type, C)$ and $P$ $\text{rdfs:range}$ $C \in \mathcal{S}$
(11)             $ucq \leftarrow ucq \cup \{r_{[g/\exists y\ t(y,P,s)]}\}$
(12)          else if $g = t(s, P_1, o)$ and $P_2$ $\text{rdfs:subPropertyOf}$ $P_1 \in \mathcal{S}$
(13)             $ucq \leftarrow ucq \cup \{r_{[g/t(s,P_2,o)]}\}$
(14) return $ucq$
```

as the set of conjunctive views $V_0 = \bigcup_{i=1}^{n}\{q_i^1, \dots, q_i^{n_i}\}$, the graph G_0 of V_0, and the set of rewritings $R_0 = \bigcup_{i=1}^{n}\{q_i = q_i^1 \cup \dots \cup q_i^{n_i}\}$.

5 Experimental Evaluation

We present a set of experiments we conducted to evaluate our approach. The view selection software is a standalone Java module (100 classes, 13.500 lines). It takes its input under the form of a set of conjunctive RDF queries, a set of query weights, and an RDF-S schema, and produces as output the set of recommended views and corresponding rewritings. To study the benefits of the recommended views, we coupled our software with a database back-end which stores both the original RDF data and the views. As a back-end, we have chosen PostgreSQL, both for its reputation of a (free) efficient platform, and because it was used in several previous RDF data management works [1, 12, 13, 15, 18]. The PostgreSQL version we used is 8.4.3.

For our experiments, we used the Barton data[2], one of the most widely used in previous RDF data management works [1, 12, 18]). The initial data set consists of about 50 million triples. After some cleaning of the data (removing formatting errors, eliminating duplicates etc.) we kept around 35 million distinct triples. We loaded them in PostgreSQL under the form of the triple table $t(s, p, o)$. The space occupied by the table within PostgreSQL was 15.9 GB. View set enumeration and selection was performed on 2.40GHz Intel Xeon machine with 4GB RAM (2GB for the JVM), running Mandriva Linux. The PostgreSQL server ran on a separate 2,13GHz Intel Xeon machine with 8GB RAM.

In the following, Section 5.1 reports on experiments with various search strategies. Section 5.2 studies the impact of the quality estimation functions on the search, while Section 5.3 demonstrates the interest of using the selected view set, to speed up query processing.

[2]http://simile.mit.edu/wiki/Dataset:_Barton

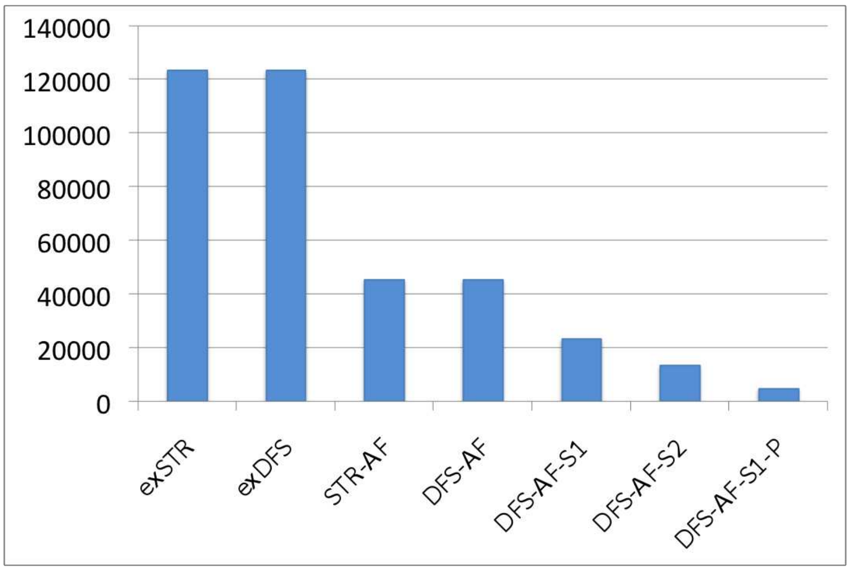

Figure 10: The number of explored states for workload 1.

5.1 Search Strategies Evaluation

We compare five variants of the Stratified (*STR*) and Depth-First (*DFS*) strategies described in Sections 3.2, respectively, 3.3. The variants are: (i) exhaustive (*ex*); (ii) with aggressive fusion enabled (*AF*); (iii) with aggressive fusion and the $stop_t$ stop condition (*AF-S1*); (iv) with aggressive fusion and the $stop_{var}$ stop condition (*AF-S2*); finally, (v) with aggressive fusion, the $stop_t$ stop condition and the "pull&push constants" technique (*AF-S1-P*). The quality function we used was the one that experimentally gave us the best results (further details on this are given in Section 5.2).

We employed four query workloads of increasing complexity (with respect to their numbers of queries, constants, and joins). Due to space limitations, here we report only on the two most complex ones (*workload 1* and *2*), which best highlight the differences between strategies. Workload 1 included 10 constants and 9 variables in total, whereas workload 2 included 15 constants and 14 variables.

Figure 10 shows the number of states explored by each alternative strategy (for workload 1), a measure having a strong impact on the memory and time needs of the strategy. As expected, for each variant, the STR and DFS have the same number of explored states (they explore the same part of the search space). Moreover, applying aggressive fusion reduces the number of states to almost one third. This number is reduced even more by the stop conditions, as they prune part of the search space (notice that $stop_t$ is less *aggressive* than $stop_{var}$). The biggest reduction of the set of explored states was achieved by combining the above techniques with the "pull&push constants".

Figure 11 depicts the memory needs of each strategy for both workloads. A first observation is that, for each given variant, STR consumes more memory than the DFS. This confirms our expectations, since STR needs to keep in memory more states for a longer time than DFS (which explores them in depth and then is able to discard them). Furthermore, Figure 11 shows that neither of the exhaustive and the aggressive fusion STR did manage to complete their execution with the given amount of memory. The most interesting point though is the significant reduction in memory needed when we apply our heuristics. Again, the best results appear when using the "pull&push constants" technique.

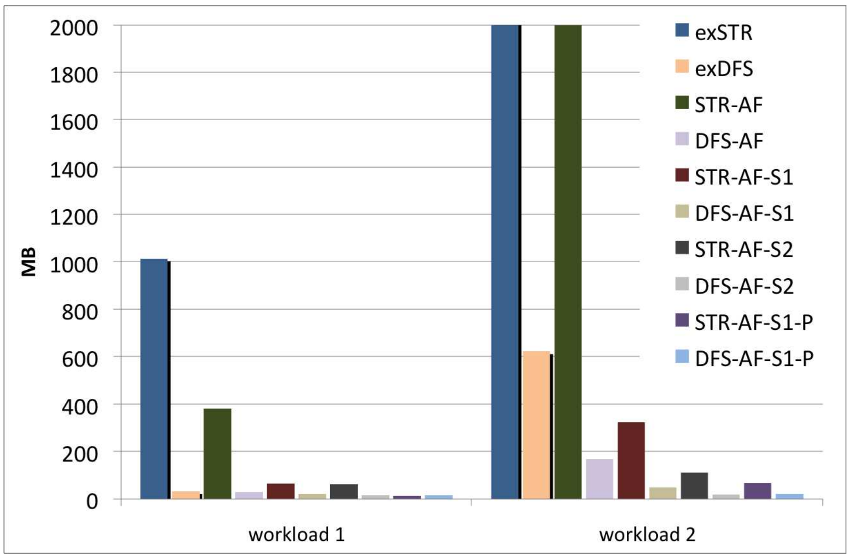

Figure 11: Memory consumption of each strategy.

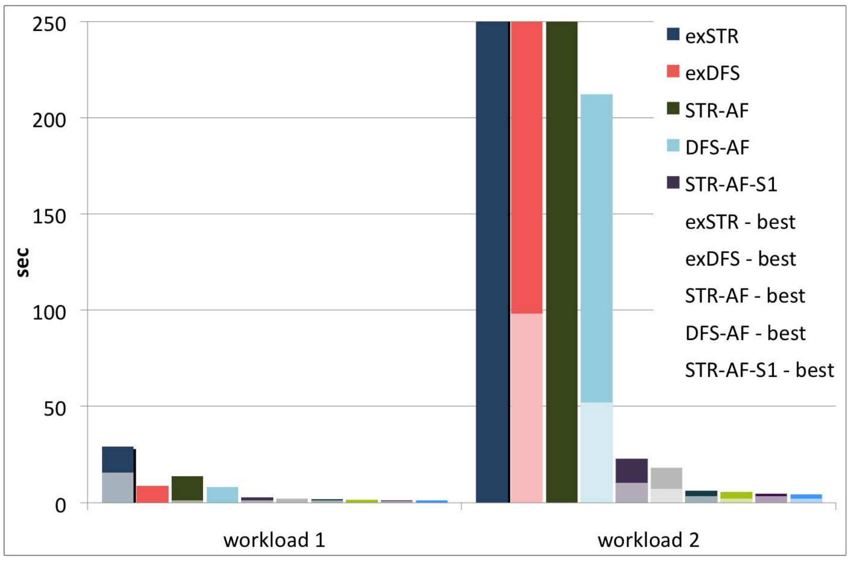

Figure 12: Total times and times until best state for each strategy.

The execution times for each strategy are depicted in Figure 12. For each strategy and variant, the lower part of the corresponding bar (in lighter color) corresponds to the time needed to find the best state, whereas the overall bar represents the total running time. One could expect that STR and DFS would have very close execution times, since they explore the same set of states. However, this is not always the case, due to the fact that STR spends more time reaching again and again the same state (recall the comparison between Figure 6 and Figure 8). Especially for those cases which demand a lot of memory, the time advantage of DFS is noticeable. Figure 12 also shows that our heuristics improve the execution time by a factor of more than 30. Finally, DFS is also faster at finding the best state.

Finally, we compare our ten strategies also from the viewpoint of the quality of the solution they are able to find. Figure 13 demonstrates that, as expected, aggressive fusion preserved optimality (gave the same scores as the exhaustive strategies). In most cases, enabling the rest of the heuristics did not prevent us from reaching the best state. Even when the overall best

	exSTR	exDFS	STR-AF	DFS-AF	DFS-AF-S1	DFS-AF-S2	DFS-AF-S1-P
workload 1	0.494	0.494	0.494	0.494	0.494	0.494	0.469
workload 2	0.540	0.540	0.540	0.540	0.540	0.506	0.540

Figure 13: The score of the best state reached by each strategy.

state was not explored, the returned best state had a score very close to the best one.

5.2 Quality estimation

The quality function we used in order to evaluate the quality of each state relies on a number of quality estimators. The most prevalent estimators we have considered are the following:

- join count (JCo): the total number of joins appearing in the rewritings R_i of a state S_i (favors states with small number of joins, to decrease execution times)
- average constant ratio (CR): the average number of constants that appear in each view (favors views with many constants, as they will be more selective and will probably occupy less space)
- selectivity estimator (SE): relies on the number of appearances of each constant in the data to determine how selective a constant is (like CR, it favors selective views)
- view popularity (VP): counts the number of times each view participates in the rewritings R_i (in an attempt to reduce the needed space, it favors states in which views are re-used in many rewritings).

To determine the best combination of them to use, we run a set of experiments using the DFS strategy with the aggressive view fusion and the $stop_{var}$ on five Barton queries (including 15 constants and 14 variables in total). In Figure 14, we report on the space (as a percentage of the triple table) occupied by the set of views selected by our search strategy, as well as the time needed to materialize these views. We have also given the materialization details after enabling reasoning support and using the rewriting algorithm described in Section 4, denoted by $CR+$ in the Figure. We used the RDF Schema provided by the Barton dataset.

The table in Figure 14 shows that for this workload, CR showed the best performance both in terms of space and time and, thus, we used CR as a quality estimator in the other experiments described here. We plan to experiment with more complex workloads in the future, and hopefully add to our framework an RDF query size estimation technique such as the ones in [11, 13], since they would provide us with more accurate information in assessing the quality of a proposed view set.

	MO	ALL	SE	CR/VP/JCo	VP	CR	CR+
Space Usage (%)	5.93	>100	>100	>100	43.70	1.33	3.08
Materialization time (min.)	2.5	>30	>30	>30	23	5	1.3

Figure 14: Quality estimators comparison.

5.3 Query execution times

We now compare the time it takes to evaluate each of the five Barton queries based on several organizations of the data in tables (views). We denote by TT0 the RDF database consisting of the single triple table (no index). Clearly, TT0 is a very poor data layout, therefore, for fairness, we also investigated using the triple table, in conjunction with several known indexing strategies. TT1 denotes the triple table with three indexes, on s, p and o respectively (also tried

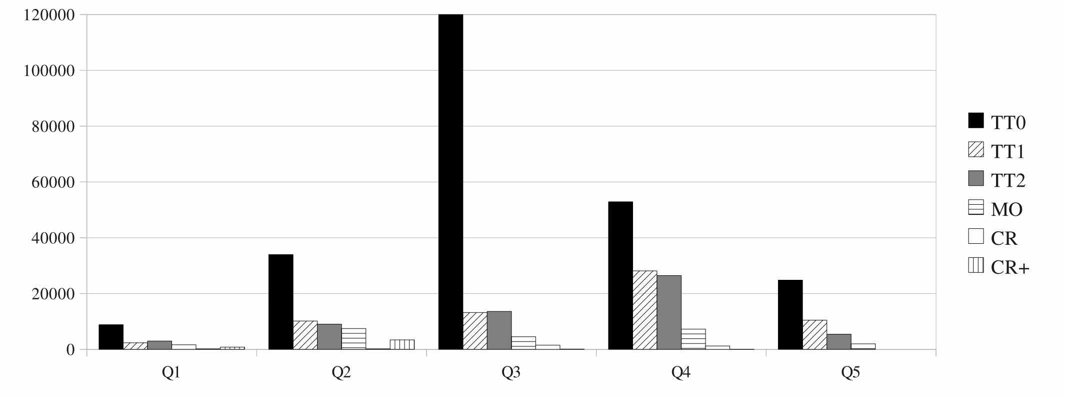

Figure 15: Workload execution times.

in [1]). TT2 adopts the approach described in [18], which consists of adding six indices, one on each permutations of the triple table columns. Prior to running the view selection algorithm, we also attempted a manual optimization of the storage (MO) selecting "by hand" some views which seemed promising for the considered workload. While the result of MO obviously depends on the user's expertise, we include it as a rough benchmark of a "reasonably-chosen" data layout. Finally, CR and CR+ denote the set of materialized views recommended by the DFS-AF-S1-P strategy, using the CR quality estimator, without reasoning (CR) respectively, with reasoning enabled (CR+).

Figure 15 depicts the execution time of the Barton 5-query workload on the above set of alternative data layouts. Each query was timed out after two minutes (which were insufficient for the third query on the TT0 storage). Figure 15 shows that the worst cases can be avoided with the six-indices approach of [18], however, there is still a lot of room for improvement. MO does significantly better than TT2, and the configurations automatically selected by DFS-AF-S1-P gives the best results. This validates the practical interest of a well-chosen set of materialized views.

6 Related works

View-based query rewriting The view-based rewritings we consider are related to the general problem of rewriting conjunctive queries using conjunctive views [9]. In particular, the main results on rewriting non-recursive Datalog queries using similar views (under set semantics) are owed to [4], whereas bag semantics is considered in [5]. In contrast with these works, our approach proposes views and *generates* rewritings in a constructive way, starting from the simple case when the views are identical to the queries. In contrast, in view-based query rewritings, the view set is given and rewritings must be *found* without any prior knowledge on how the query and the views are related.

Selection of relational materialized views has been intensely studied in the context of relational databases and in particular in relational data warehouses [10, 17]. Our approach is inspired from the one of [17]. Compared to that work, we innovated by developing techniques specific to RDF data and queries. In particular, specific choices had to be made given that RDF queries refer to one huge triple table, making a view that is identical to the triple table a not-so-desirable view (or even one to forbid). In the same style, while in a relational data warehouse scenario, it may make sense to materialize the cartesian product of two small tables (say, two dimension tables), cartesian products of the triple table with itself cannot be envisioned. Finally, we have devised search strategies specific to the RDF setting and have shown how RDF Schemas can be exploited to make view selection aware of data semantics.

RDF data management has been the topic of active work. Efficiently querying graph data is notoriously difficult, due to the potential very poor data access locality (a query may need to traverse large portions of the data set) [14]. This is the reason why RDF data management platforms (e.g. Sesame, 3store, Jena) are based on relational database management systems (RDBMSs in short).

Recent works in database research also considered extending or revisiting relational data management techniques to the processing of RDF queries. In [1], the authors propose splitting the data set using vertical partitioning. Thus, for each particular property, there is a two-column (s, o) table. This performs well for queries joining triples on the subject or object, and it allows selective access to triples of a given property. However, its performance degrades significantly for queries containing triples of unspecified property names, since this requires a union over all the property tables. The authors experimentally compare a standard RDBMS (Postgres) with a column store, and find that the latter is much more efficient in some settings. These results are revisited in [15], which shows that when properly tuned and complemented with indices, the RDBMS may perform almost as well as the vertically partitioned store. The authors also point out the fragility of the property tables when there are too many properties in the data set. Hexastore [18] proposes a more generic approach for efficiently storing RDF data, based on one single triple table and six indices, on all permutations of the attributes. Thus, there is an index on (s, p), another on (s, o), another one on (p, s) etc. Hexastore's indices can be used to speed up all types of joins over the triple table, however, as we have shown, custom views can make processing even more efficient. Finally, RDF-3X [12, 13] is a recent, efficient platform natively built for RDF. It implements the six indices of Hexastore, as well as a fixed set of indices inspired from a query workload.

The work we presented in this paper can integrate with any of these previous platforms. Instead of specific basic storage methods and indices, we propose a framework for exploring alternative data access support structures which could be materialized to help query processing even further. Materialized views improve query processing performance in all scenarios, simply by pre-computing results. On the other hand, complex views require more maintenance effort than simple indices over two attributes. Existing algorithms for incremental maintenance of conjunctive relational views [8] directly apply, and an estimation of view maintenance costs (based on the expected frequency of updates etc.) can be incorporated in our quality functions. Clearly, materialized views benefits are more important on relatively slow-changing data sets.

Techniques to estimate the selectivity of RDF query patterns were proposed in [11, 16]. The authors of [11] propose a summarization-based approach. They first identify a set of sub-patterns to be counted exactly, based on which the cardinality of more complex queries can be estimated. Finding the optimal set of such sub-patterns to materialize is NP-hard, thus some greedy algorithms are proposed. The work described in [16] focuses on optimizing in-memory RDF database queries. The authors study combinations of simple cardinality statistics, heuristics based on the query syntax, and a probabilistic estimation framework to decide join evaluation order. Finally, the more recent RDF-3X work [13] describes a set of simple cardinality statistics based on which join selectivities are estimated, and joins are ordered. As part of our future work, we plan to integrate a selectivity estimation framework in our quality functions.

The translation of SPARQL into SQL has been studied in [6], with a focus on SPARQL semantics preservation, and [7] focusing on the efficiency of the resulting SQL queries. Our algorithm for translating RDF queries (and views) into SQL follows the approach of [7].

Finally, ontologies or other semantic data source descriptions have been widely used to guide the integration (interoperability) of heterogeneous data sources, see, e.g., [3]. Our work bears some similarity to this, but we only exploit simple RDFS relationships.

7 Conclusion

Efficiently querying RDF data raises many challenging problems, in particular in the areas of access path selection and query processing, but also in the interpretation of results, given that RDF data often comes with associated RDF Schemas that allow to interpret and enrich it. This work is the first to investigate the adaptation of materialized view selection methods proposed for relational data management, to the management of RDF data. We have proposed a framework for materialized view set selection, studied the search space, formalized a set of interesting strategies, and evaluated them through a set of experiments, which confirm the feasibility and the interest of our approach. In future works, we plan to refine our quality functions, integrating query cardinality estimations, and experiment with more heuristics.

References

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

[2] P. Adjiman, F. Goasdoué, and M.-C. Rousset. SomeRDFS in the Semantic Web. *JODS*, 8, 2007.

[3] B. Amann, C. Beeri, I. Fundulaki, and M. Scholl. Ontology-based integration of xml web resources. In *ISWC*, 2002.

[4] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[5] S. Chaudhuri and M. Y. Vardi. Optimization of *real* conjunctive queries. In *PODS*, 1993.

[6] A. Chebotko, S. Lu, and F. Fotouhi. Semantics preserving SPARQL-to-SQL translation. *Data Knowl. Eng.*, 68(10), 2009.

[7] B. Elliott, E. Cheng, C. Thomas-Ogbuji, and Z. M. Ozsoyoglu. A complete translation from SPARQL into efficient SQL. In *IDEAS*, New York, NY, USA, 2009. ACM.

[8] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

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

[10] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In H. V. Jagadish and I. S. Mumick, editors, *SIGMOD*, 1996.

[11] A. Maduko, K. Anyanwu, A. P. Sheth, and P. Schliekelman. Graph summaries for subgraph frequency estimation. In *ESWC*, 2008.

[12] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. of VLDB*, 1(1), 2008.

[13] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, New York, NY, USA, 2009. ACM.

[14] D. Quass, A. Rajaraman, J. D. Ullman, J. Widom, and Y. Sagiv. Querying semistructured heterogeneous information. *Journal of Systems Integration*, 7(3/4), 1997.

[15] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *Proc. of VLDB*, 1(2), 2008.

[16] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.

[17] D. Theodoratos, S. Ligoudistianos, and T. K. Sellis. View selection for designing the global data warehouse. *Data Knowl. Eng.*, 39(3), 2001.

[18] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. of VLDB*, 1(1), 2008.

[19] RDF - semantic web standards. Available at www.w3.org/RDF/, 2004.

[20] RDF Vocabulary Description Language 1.0: RDF Schema. Available at www.w3.org/TR/rdf-schema/, 2004.

[21] SPARQL query language for RDF. Available at www.w3.org/TR/rdf-sparql-query/, 2008.