View Selection in Semantic Web Databases *

François Goasdoué[1] Konstantinos Karanasos[1] Julien Leblay[1] Ioana Manolescu[1]
[1]Leo team, INRIA Saclay and LRI, Université Paris-Sud 11
firstname.lastname@inria.fr

ABSTRACT

We consider the setting of a Semantic Web database, containing both explicit data encoded in RDF triples, and implicit data, implied by the RDF semantics. Based on a query workload, we address the problem of selecting a set of views to be materialized in the database, minimizing a combination of query processing, view storage, and view maintenance costs. Starting from an existing relational view selection method, we devise new algorithms for recommending view sets, and show that they scale significantly beyond the existing relational ones when adapted to the RDF context. To account for implicit triples in query answers, we propose a novel RDF query reformulation algorithm and an innovative way of incorporating it into view selection in order to avoid a combinatorial explosion in the complexity of the selection process. The interest of our techniques is demonstrated through a set of experiments.

1. INTRODUCTION

A key ingredient for the Semantic Web vision [4] is a data format for describing items from the real and digital world in a machine-exploitable way. The W3C's resource description framework (RDF, in short [27]) is a leading candidate for this role.

At a first look, querying RDF resembles querying relational data. Indeed, at the core of the W3C's SPARQL query language for RDF [28] lies conjunctive relational-style querying. There are, however, several important differences in the data model. First, an RDF data set is a single large set of triples, in contrast with the typical relational database featuring many relations with varying numbers of attributes. Second, RDF triples may feature *blank nodes*, standing for unknown constants or URIs; an RDF database may, for instance, state that the *author* of X is *Jane* while the *date* of X is *4/1/2011*, for a given, unknown resource X. This contrasts with standard relational databases where all attribute values are either constants or $null$. Finally, in typical relational databases, all data is *explicit*, whereas the semantics of RDF entails a set of *implicit* triples which must be reflected in query answers. One important source of implicit triples follows from the use of an (optional) RDF Schema (or RDFS, in short [27]), to enhance the descriptive power of an RDF data set. For instance, assume the RDF database contains the fact that the *driverLicenseNo* of *John* is *12345*, whereas an RDF Schema states that only a *person* can have a *driverLicenseNo*. Then, the fact that *John* is a *person* is implicitly present in the database, and a query asking for all *person* instances in the database must return *John*.

The complex, graph-structured RDF model is suitable for describing heterogeneous, irregular data. However, it is clearly not a good model for storing the data. Existing RDF platforms therefore assume a simple (application-independent) storage model, complemented by indexes and efficient query evaluation techniques [1, 15, 16, 17, 20, 24], or by RDF materialized views [6, 9]. While indexes or views speed up the evaluation of the fragments of queries matching them, the query processor may still need to access the main RDF database to evaluate the remaining fragments of the queries.

We consider the problem of *choosing a (relational) storage model for an RDF application*. Based on the application workload, we seek a set of views to materialize over the RDF database, such that *all workload queries can be answered based solely on the recommended views, with no need to access the database*. Our goal is to enable three-tier deployment of RDF applications, where clients do not connect directly to the database, but to an application server, which could store only the relevant views; alternatively, if the views are stored at the client, no connection is needed and the application can run off-line, independently from the database server.

RDF datasets can be very different: data may be more or less structured, schemas may be complex, simple, or absent, updates may be rare or frequent. Moreover, RDF applications may differ in the shape, size and similarity of queries, costs of propagating updates to the views etc. To capture this variety, we characterize candidate view sets by a cost function, which combines (i) query evaluation costs, (ii) view maintenance costs and (iii) view storage space. Our contributions are the following:

1. This is the first study of RDF materialized view selection supporting the rewriting of all workload queries. We show how to model this as a search problem in a space of states, inspired from a previous work in relational data warehousing [22].

2. Implicit triples entailed by the RDF semantics [27] must be reflected in the recommended materialized views, since they may participate to query results. Two methods are currently used to include implicit tuples in query results. *Database saturation* adds them to the database, while *query reformulation* leaves the database intact and modifies queries in order to also capture implicit triples. Our approach requires no special adaptation if applied on a saturated database. For the reformulation scenario, we propose a novel RDF query reformulation algorithm. This algorithm extends the state of the art in query processing in the presence of RDF Schemas [3,

*This work has been partially funded by *Agence Nationale de la Recherche*, decision ANR-08-DEFIS-004.

Proceedings of the VLDB Endowment, Vol. 5, No. 1

5], and is a contribution applying beyond the context of this work. Moreover, we propose an innovative method of using reformulation (called *post-reformulation*) which enables us to efficiently take into account implicit triples in our view selection approach.

3. We consider heuristic search strategies, since the complexity of complete search is extremely high. Existing strategies for relational view selection [22] grow out of memory and fail to produce a solution when the number of atoms in the query workload grows. Since RDF atoms are short (just three attributes), RDF queries are syntactically more complex (they have more atoms) than relational queries retrieving the same information, making this scale problem particularly acute for RDF. We propose a set of new strategies and heuristics which greatly improve the scalability of the search.

4. We study the efficiency and effectiveness of the above algorithms, and their improvement over existing similar approaches, through a set of experiments.

This paper is organized as follows. Section 2 formalizes the problem we consider. Section 3 presents the view selection problem as a search problem in a space of candidate states, whereas Section 4 discusses the inclusion of implicit RDF triples in our approach. Section 5 describes the search strategies and heuristics used to navigate in the search space. Section 6 presents our experimental evaluation. Section 7 discusses related works, then we conclude.

2. PROBLEM STATEMENT

In accordance with the RDF specification [27], we view an RDF database as a set of (s, p, o) triples, where s is the *subject*, p the *property*, and o the *object*. RDF triples are *well-formed*, that is: subjects can be URIs or *blank nodes*, properties are URIs, while objects can be URIs, blank nodes, or literals (i.e., values). Blank nodes are placeholders for unknown *constants* (URIs or literals); from a database perspective, they can be seen as existential variables in the data. While relational tuples including the $null$ token, commonly used to represent missing information, do not join ($null$ does not satisfy any predicate), RDF triples referring to *the same* blank node may be joined to construct complex results, as exemplified in the Introduction. Due to blank nodes, an RDF database can be seen as an incomplete relational database consisting of a single *triple table* $t(s, p, o)$, under the open-world assumption [2].

To express RDF queries (and views), we consider the basic graph pattern queries of SPARQL [28], represented *wlog* as a special case of conjunctive queries: conjunctions of atoms, the terms of which are either free variables (a.k.a. head variables), existential variables, or constants. We do not use a specific representation for blank nodes in queries, although SPARQL does, because they behave exactly like existential variables.

DEFINITION 2.1 (RDF QUERIES AND VIEWS). *An RDF query (or view) is a conjunctive query over the triple table* $t(s, p, o)$.

We consider *wlog* queries *without Cartesian products*, i.e., each triple shares at least one variable (joins at least) with another triple. We represent a query with a Cartesian product by the set of its independent sub-queries. Finally, we assume queries and views are *minimal*, i.e., the only containment mapping from a query (or view) to itself is the identity [7].

As a running example, we use the following query q_1, which asks for painters that have painted "Starry Night" and having a child that is also a painter, as well as the paintings of their children:

$$q_1(X, Z) :- t(X, hasPainted, starryNight), t(X, isParentOf, Y), t(Y, hasPainted, Z)$$

Based on views, one can rewrite the workload queries:

DEFINITION 2.2 (REWRITING). *Let q be an RDF query and $V = \{v_1, v_2, \ldots, v_k\}$ be a set of RDF views. A rewriting of q based on V is a conjunctive query (i) equivalent to q (i.e., on any data set, it yields the same answers as q), (ii) involving only relations from V and (iii) minimal, in the sense mentioned above.*

We are now ready to define our view selection problem, which relies on candidate view sets:

DEFINITION 2.3 (CANDIDATE VIEW SET). *Let Q be a set of RDF queries. A candidate view set for Q is a pair $\langle V, R\rangle$ such that:*

- *V is a set of RDF views,*
- *R is a set of rewritings such that: (i) for every query $q \in Q$ there exists exactly one rewriting $r \in R$ of q using the views in V; (ii) all V views are useful, i.e., every view $v \in V$ participates to at least one rewriting $r \in R$.*

We consider a *cost estimation function* c^ϵ which returns a quantitative measure of the costs associated to a view set. The lower the cost, the better the candidate view set is. Our cost components include the effort to evaluate the view-based query rewritings, the total space occupancy of the views and the view maintenance costs as data changes. More details about c^ϵ are provided in Section 3.4.

DEFINITION 2.4 (VIEW SELECTION PROBLEM). *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of RDF queries and c^ϵ be a cost estimation function. The view selection problem consists in finding a candidate view set $\langle V, R\rangle$ for Q such that, for any other candidate view set $\langle V', R'\rangle$ for Q: $c^\epsilon(\langle V, R\rangle) \leq c^\epsilon(\langle V', R'\rangle)$.*

3. THE SPACE OF CANDIDATE VIEW SETS

This Section describes our approach for modeling the space of possible candidate view sets. Section 3.1 introduces the notion of a state to model one such set, while Section 3.2 presents a set of transitions that can be used to transform one state to another. Then, Section 3.3 discusses the details of detecting view and state equivalence and finally, Section 3.4 shows how to assign a cost estimation to each state.

3.1 States

We use the notion of *state* to model a candidate view set together with the rewritings of the workload queries based on these views. The set of all possible candidate view sets, then, is modeled as a set of states, which we adapt from the previous work on materialized view selection in a relational data warehouse [22]. From here forward, given a workload Q, we may use $S(Q)$ (possibly with subscripts or superscripts) to denote a candidate view set for Q. To ease the exposition, we also employ from [22] a visual representation of each state by means of a *state graph*.

DEFINITION 3.1 (STATE GRAPH). *Given a query set Q and a state $S_i(Q) = \langle V_i, R_i\rangle$, the state graph $G(S_i) = (N_i, E_i)$ is a directed multigraph such that:*

- *each triple t_i appearing in a view $v \in V_i$ is represented by a node $n_i \in N_i$;*
- *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*;*
- *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 c_i, that appears in the attribute $a_i \in \{s, p, o\}$ of t_i, an edge labeled $v{:}n_i.a_i = c_i$ connects n_i to itself. Such an edge is called* selection edge.

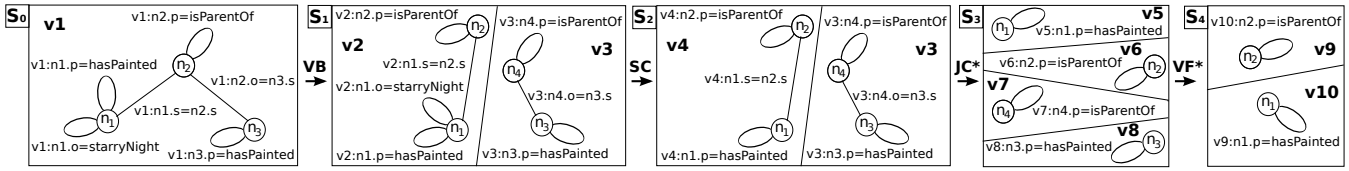


Figure 1: Sample initial state graph S_0, and states attained through successive transitions.

The *graph of* v is defined as the subgraph of $G(S_i)$ corresponding to v. Observe that in a view, two nodes may be connected by several join edges if their corresponding atoms are connected by more than one join predicates.

We define two states to be *equivalent* if they have the same view sets. Furthermore, to avoid a blow-up in the storage space required by the views, *we do not consider views including Cartesian products*. In a relational setting, some Cartesian products, e.g., between small dimension tables in an OLAP context, may not raise performance issues. In contrast, in the RDF context where all data lies in a single large triple table, Cartesian products are likely not interesting queries, and their storage overhead is prohibitive. The absence of Cartesian products from our views entails that *the graph of every view is a connected component of the state graph*.

As a (simple) example, consider the state $S_0(Q) = \langle\{v_1\}, R_0\rangle$, where $Q = \{q_1\}$ is a workload containing only the previously introduced query q_1, and $v_1 = q_1$. The rewriting set R_0 consists of the trivial rewriting $\{q_1 = v_1\}$. The graph $G(S_0)$ is depicted at left in Figure 1, and since it corresponds to a single view, it comprises only one connected component.

3.2 State Transitions

To enumerate candidate view sets (or, equivalently, states), we use four transitions, inspired from [22]. As we show in Section 5.1, our transition set is complete, i.e., all possible states for a given workload can be reached through our four transitions. The first three transitions remove predicates from views, thus can be seen as "relaxing", and may split a view in two, increasing the number of views. The last one factorizes two views into one, thus reducing the number of workload views. The graphs corresponding to the states before and after each transition are illustrated in Figure 1.

We use $v{:}e$ to denote an edge e belonging to the view v in a state graph. While we define rewritings as conjunctive queries, for ease of explanation, we now denote rewritings by (equivalent) relational algebra expressions. We use σ_e to denote a selection on the condition attached to the edge e in a view set graph. Since the query set Q is unchanged across all transitions, we omit it for readability.

DEFINITION 3.2 (VIEW BREAK (VB)). *Let $S = \langle V, R\rangle$ be a state, v a view in V and N_v the set of nodes of the graph of v with $|N_v| > 2$. Let N_{v_1}, N_{v_2} be two subsets of N_v such that:*

- *$N_{v_1} \nsubseteq N_{v_2}$ and $N_{v_2} \nsubseteq N_{v_1}$;*
- *$N_{v_1} \cup N_{v_2} = N_v$;*
- *the subgraph of the graph of v defined by N_{v_1} (respectively, by N_{v_2}) and the edges between these nodes is connected.*

We create two new views, v_1 and v_2. View v_1 (respectively v_2) derives from the graph of v by copying the nodes corresponding to N_{v_1} (N_{v_2}) and the edges between them. The head variables of v_1 (v_2) are those of v appearing also in the body of v_1 (v_2), together with all additional variables appearing in the nodes $N_{v_1} \cap N_{v_2}$.

The new state $S' = \langle V', R'\rangle$ consists of:

- *$V' = (V \setminus \{v\}) \cup \{v_1, v_2\}$,*
- *$G(S')$ is obtained from $G(S)$ by removing the graph of v and adding those of v_1 and v_2, and*
- *R' is obtained from R by replacing all the occurrences of v, with $\pi_{head(v)}(v_1 \bowtie v_2)$, where $\bowtie$ is the natural join.*

For example, we apply a view break on the view v_1 of state S_0 introduced in the previous Section, and obtain the new state S_1:

$$S_1 = \langle\{v_2, v_3\}, \{q_1 = \pi_{head(v_1)}(v_2 \bowtie v_3)\}\rangle$$

The two newly introduced views v_2 and v_3 are the following:

$v_2(X, Y)$:$-t(X, hasPainted, starryNight), t(X, isParentOf, Y)$

$v_3(X, Y, Z)$:$-t(X, isParentOf, Y), t(Y, hasPainted, Z)$

DEFINITION 3.3 (SELECTION CUT (SC)). *Let $S = \langle V, R\rangle$ be a state and $v{:}e$ be a selection edge in $G(S)$. A* selection cut *on e yields a state $S' = \langle V', R'\rangle$ such that:*

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

For instance, we apply a selection cut on the edge labeled $v_2{:}n_1.o = starryNight$ of $G(S_1)$ and obtain the state S_2, in which v_2 is replaced by a new view v_4:

$$S_2 = \langle\{v_3, v_4\}, \{q_1 = \pi_{head(v_1)}(\pi_{head(v_2)}(\sigma_{n_1.o=starryNight}(v_4)) \bowtie v_3)\}\rangle$$

View v_4 is the following:

$v_4(X, Y, W)$:$-t(X, hasPainted, W), t(X, isParentOf, Y)$

DEFINITION 3.4 (JOIN CUT (JC)). *Let $S = \langle V, R\rangle$ be a state and $v{:}e$ be a join edge in $G(S)$ of the form $n_i.c_i = n_j.c_j$, such that $c_i, c_j \in \{s, p, o\}$. A* join cut *on e yields a state $S' = \langle V', R'\rangle$, 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 view v' in which the variable corresponding to the join edge e becomes a head variable, and the occurrence of that variable corresponding to $n_i.c_i$ is replaced by a new fresh head variable. The new rewriting set R' is obtained from R by replacing v by $\pi_{head(v)}(\sigma_e(v'))$. The new graph $G(S')$ is obtained from $G(S)$ by removing the graph of v and adding the one of v'.*
2. *If the graph of v is split in two 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 $G(S')$ is obtained from $G(S)$ by removing the graph of v and adding the ones of v'_1 and v'_2.*

For example, cutting the join edge $v_4{:}n_1.s = n_2.s$ of $G(S_2)$ disconnects the graph of v_4, resulting in two new views, v_5 and v_6 (see Figure 1). View symbol v_4 is replaced in the rewritings by the expression $\pi_{head(v_4)}(v_5 \bowtie_{n_1.s=n_2.s} v_6)$. If we continue by cutting the edge $v_3{:}n_4.o = n_3.s$, v_3 is split into v_7 and v_8. The resulting state S_3 is:

$$S_3 = \langle\{v_5, v_6, v_7, v_8\}, \{q_1 = \pi_{head(v_1)}(\pi_{head(v_2)}(\sigma_{n_1.o=starryNight}(\pi_{head(v_4)}(v_5 \bowtie_{n_1.s=n_2.s} v_6))) \bowtie \pi_{head(v_3)}(v_7 \bowtie_{n_4.o=n_3.s} v_8)\}\rangle$$

The new views introduced in S_3 are the following:

$v_5(X,W)$:$-t(X, hasPainted, W)$

$v_6(X,Y)$:$-t(X, isParentOf, Y)$

$v_7(X,Y)$:$-t(X, isParentOf, Y)$

$v_8(Y,Z)$:$-t(Y, hasPainted, Z)$

DEFINITION 3.5 (VIEW FUSION (VF)). *Let $S = \langle V, R\rangle$ be a state and v_1, v_2 be two views in V such that their respective graphs are isomorphic (their bodies are equivalent up to variable renaming). We denote by $\langle i \to j\rangle$ the renaming of the variables of v_i into those of v_j. Let v_3 be a copy of v_1, such that $head(v_3) = head(v_1) \cup head(v_{2\langle 2\to 1\rangle})$. Fusing v_1 and v_2 leads to a new state $S' = \langle V', R'\rangle$ obtained as follows:*

- *$V' = (V \setminus \{v_1, v_2\}) \cup \{v_3\}$,*
- *$G(S')$ is obtained from $G(S)$ 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 with $\pi_{head(v_1)}(v_3)$, and of v_2 with $\pi_{head(v_2)}(v_{3\langle 3\to 2\rangle})$*

For example, in state S_3, the graphs of v_5 and v_8 are isomorphic, and can thus be fused creating the new view v_9. Similarly, v_6 and v_7 can be fused into a new view v_{10} leading to state S_4.

Our transitions adapt those introduced in [22] to our RDF view selection context.

3.3 View and State Equivalence

As mentioned in Definition 3.5 above, in order to apply a VF between two views, we first need to check whether these views are equivalent (up to variable renaming). It is shown in [7] that in the general case, equivalence between conjunctive queries is NP-complete, which would make VF very expensive. To this end, we have devised a signature-based filter to more efficiently determine view equivalence. In particular, we assign a signature to each view taking into account the number of atoms of the view, the constants and the position (s, p or o) in which each constant appears, as well as the joins to which each variable participates (for each variable, we count the number of s-s, s-p, s-o, p-s, etc. joins to which it participates in the views). These signatures are small and can be built very fast.

If two views are equivalent, their signatures are the same. Thus, in order to test if $v_1 \equiv v_2$, we first compare their signatures. If they are different, we are sure the views are not identical. If they coincide, we apply the full equivalence test of [7]. The filter eliminates quickly many non-equivalent pairs and speeds up the whole process significantly.

As for the state equivalence, we build state signatures by sorting and then concatenating the signatures of all the views composing the state. If two states have different signatures, they are certainly not equivalent. If they have the same signature, their views should be checked for equivalence, which is more costly. This efficient state equivalence test plays an important role when searching for candidate states, as it will be explained in Section 5.1.

3.4 Estimated State Cost

To each state, we associate a *cost estimation* c^ϵ, taking into account: the space occupancy of all the materialized views, the cost of evaluating the workload query rewritings, and the cost associated to the maintenance of the materialized views.

For any conjunctive query or view v, we use $len(v)$ to denote the number of atoms in v, $|v|$ for the number of tuples in v and $|v|^\epsilon$ for our *estimation* of this number. Let $S(Q) = \langle V, R\rangle$ be a state.

View space occupancy (VSO$^\epsilon$) To estimate the space occupancy of a given view $v \in V$, we need to estimate its cardinality. Several methods exist for estimating RDF query cardinality [13, 21]. In this work, we adopt the solution of [16], which consists in counting and storing the exact number of tuples (i) for each given s, p and o value; (ii) for each pair of (s,p), (s,o) and (s,p) values. This leads to exact cardinality estimations for any 1-atom view with 1 or 2 constants. The size of an 1-atom view with no constants is the size of the dataset; three-constants atoms are disallowed in our framework since they introduce Cartesian products in views.

We now turn to the case of multi-atom views. From each view $v \in V$, and each atom $t_i \in v$, $1 \le i \le len(v)$, let v^i be the conjunctive query whose body consists of exactly the atom t_i and whose head projects the variables in t_i. From our gathered statistics, we know $|v^i|$. We assume that values in each triple table column are *uniformly distributed*, and that values of different columns are *independently distributed*[1]. For the s, p and o columns, moreover, we store the number of distinct values, as well as the minimum and maximum values. Then, we compute $|v|^\epsilon$ based on the exact counts $|v^i|$ and the above assumptions and statistics, applying known relational formulas [18]. Finally, we use the average size of a subject, property, respectively object, the attributes in the head of v, and $|v|^\epsilon$, to estimate the space occupancy of view v.

Since the workload is known, we gather only the statistics needed for this workload: (i) we count the triples matching each of the query atoms (ii) we also count the triples matching all *relaxations* of these atoms, obtained by removing constants (as SC does during the search). Consider, for instance, the following query:

$$q(X_1, X_2){:-}t(X_1, rdf{:}type, picture), t(X_1, isLocatIn, X_2)$$

We count the triples matching the two query atoms:

$q^1(X_1){:-}t(X_1, rdf{:}type, picture), q^2(X_1, X_2){:-}t(X_1, isLocatIn, X_2)$

as well as the triples matching three *relaxed atoms*, obtained by removing the constants from q^1 and q^2:

$q^3(X_1, X_2){:-}t(X_1, rdf{:}type, X_2)$, $q^4(X_1, X_2){:-}t(X_1, X_2, picture)$, $q^5(X_1, X_2, X_3){:-}t(X_1, X_2, X_3)$.

Based on the cardinalities of the above atoms, we can estimate the cardinality of any possible view created throughout the search.

Rewriting evaluation cost (REC$^\epsilon$) This cost estimation reflects the processing effort needed to answer the workload queries using the proposed rewritings in R. It is computed as:

$$REC^\epsilon(S) = \sum_{r\in R}(c_1 \cdot io^\epsilon(r) + c_2 \cdot cpu^\epsilon(r))$$

where $io^\epsilon(r)$ and $cpu^\epsilon(r)$ estimate the I/O cost and the CPU processing cost of executing the rewriting r respectively, and c_1, c_2 are some weights. The I/O cost estimation is:

$$io^\epsilon(r) = \sum_{v\in r} |v|^\epsilon$$

where $v \in r$ denotes a view appearing in the rewriting r.

The CPU cost estimation $cpu^\epsilon(r)$ sums up the estimated costs of the selections, projections, and joins required by the rewriting r, computed based on the view cardinality estimations and known formulas from the relational query processing literature [18].

View maintenance cost (VMC$^\epsilon$) The cost of maintaining the views in V when the data is updated depends on the algorithm implemented to propagate the updates. In a conservative way, we chose to account only for the costs of writing/removing tuples to/from the views due to an update, ignoring the other maintenance operation costs. Consider the addition of a triple t_+ to the triple table, and a view v of $len(v)$ atoms. With some simplification, we consider that t_+ joins with f_1 existing triples for some constant f_1, the tuples resulting from this, in turn, join with f_2 existing triples etc.

[1] A very recent work [14] provides an RDF query size estimation method which does not make the independence assumption. This estimation method could easily be integrated in our framework.

Semantic relationship	RDF notation	FOL notation
Class inclusion	$(c_1, \text{rdfs:subClassOf}, c_2)$	$\forall X(c_1(X) \Rightarrow c_2(X))$
Property inclusion	$(p_1, \text{rdfs:subPropertyOf}, p_2)$	$\forall X \forall Y(p_1(X,Y) \Rightarrow p_2(X,Y))$
Domain typing of a property	$(p, \text{rdfs:domain}, c)$	$\forall X \forall Y(p(X,Y) \Rightarrow c(X))$
Range typing of a property	$(p, \text{rdfs:range}, c)$	$\forall X \forall Y(p(X,Y) \Rightarrow c(Y))$

Table 1: Semantic relationships expressible in an RDFS.

Adding the triple t_+ thus causes the addition of $f_1 \cdot f_2 \cdot \ldots \cdot f_{len(v)}$ tuples to v. A similar reasoning holds for deletions. To avoid estimating $f_1, f_2, \ldots, f_{len(v)}$, which may be costly or impossible for triples which will be added in the future, we consider a single user-provided factor f, and compute:

$$VMC^{\epsilon}(S) = \sum_{v \in V} f^{len(v)}$$

The **estimated cost** c^{ϵ} of a state S is defined as:

$$c^{\epsilon}(S) = c_s \cdot VSO^{\epsilon}(S) + c_r \cdot REC^{\epsilon}(S) + c_m \cdot VMC^{\epsilon}(S)$$

where the numerical weights c_s, c_r and c_m determine the importance of each component: if storage space is cheap c_s can be set very low, if the triple table is rarely updated c_m can be reduced etc.

Workload query weights An immediate extension to the cost function is to associate numerical weights $W = \{w_1, w_2, \ldots, w_n\}$ to each workload query, to reflect, e.g., the frequency and/or importance of a query. To account for weights, the rewriting evaluation cost needs to become a weighted sum:

$$REC^{\epsilon}_W(S) = \sum_{r_i \in R} w_i \cdot (c_1 \cdot io^{\epsilon}(r_i) + c_2 \cdot cpu^{\epsilon}(r_i))$$

denoting by w_i the weight of the query q_i for which r_i is a rewriting. Weights have no other impact on our approach and will be omitted in the sequel for simplicity.

Impact of transitions on the cost Transition SC increases the view size and adds to some rewritings the CPU cost of the selection. Thus, SC always increases the state cost. Transitions JC and VB may increase or decrease the space occupancy, and add the costs of a join to some rewritings. JC decreases maintenance costs, whereas VB may increase or decrease it. Overall, JC and VB may increase or decrease the state cost. Finally, VF decreases the view space occupancy and view maintenance costs. Query processing costs may remain the same or be reduced, but they cannot increase. Thus, VF always reduces the overall cost of a state.

4. VIEW SELECTION & RDF REASONING

The approach described so far does not take into consideration the implicit triples that are intrinsic to RDF and that complete query answers. Section 4.1 introduces the notion of RDF entailment to which such triples are due. Section 4.2 presents the two main methods for processing RDF queries when RDF entailment is considered, namely *database saturation* and *query reformulation*. In particular, we devise a novel reformulation algorithm extending the state of the art. Finally, Section 4.3 details how we take RDF entailment into account in our view selection approach.

4.1 RDF entailment

The W3C RDF recommendation [27] provides a set of *entailment rules*, which lead to deriving new implicit (or *entailed*) triples from an RDF database. We provide here an overview of these rules.

Some implicit triples are obtained by generalizing existing triples using blank nodes. For instance, a triple (s, p, o) entails the triple $(\_{:}b, p, o)$, where s is a URI and $\_{:}b$ denotes a blank node.

Some other rules derive implicit triples from the semantics of a few special URIs, which are part of the RDF standard, and are assigned special meaning. For instance, RDF provides the rdfs:Class URI whose semantics is the set of all RDF-specific (predefined) *and* user-defined URIs denoting classes to which resources may belong. For instance, when a triple states that a resource u belongs to a given user-defined class $painting$, i.e., $(u, \text{rdf:type}, painting)$ using the predefined URI rdf:type, an implicit triple states that $painting$ is a class: $(painting, \text{rdf:type}, \text{rdfs:Class})$.

Finally, some rules derive implicit triples from the semantics encapsulated in an *RDF Schema* (RDFS for short). An RDFS specifies semantic relationships between classes and properties used in descriptions. Table 1 shows the four semantic relationships allowed in RDF, together with their first-order logic semantics. Some rules derive implicit triples through the transitivity of class and property inclusions, and of inheritance of domain and range typing. For instance, if $painting$ is a subclass of $masterpiece$, i.e., $(painting, \text{rdfs:subClassOf}, masterpiece)$, which is a subclass of $work$, i.e., $(masterpiece, \text{rdfs:subClassOf}, work)$, then an entailed triple is $(painting, \text{rdfs:subClassOf}, work)$. If $hasPainted$ is a subproperty of $hasCreated$, i.e., $(hasPainted, \text{rdfs:subPropertyOf}, hasCreated)$, the ranges of which are the classes $painting$ and $masterpiece$ respectively, i.e., $(hasPainted, \text{rdfs:range}, painting)$ and $(hasCreated, \text{rdfs:range}, masterpiece)$, then those triples are implicit: $(hasPainted, \text{rdfs:range}, masterpiece)$, $(hasPainted, \text{rdfs:range}, work)$, and $(hasCreated, \text{rdfs:range}, work)$. Some other rules use the RDFS to derive implicit triples by propagating values (URIs, blank nodes, and literals) from subclasses and subproperties to their superclasses and superproperties, and from properties to classes typing their domains and ranges. If a resource u has painted something, i.e., $(u, hasPainted, \_{:}b)$, implicit triples are: $(u, hasCreated, \_{:}b)$, $(\_{:}b, \text{rdf:type}, painting)$, $(\_{:}b, \text{rdf:type}, masterpiece)$, and $(\_{:}b, \text{rdf:type}, work)$.

Returning complete answers requires considering all the implicit triples. In practice, RDF data management frameworks (e.g., Jena[2]) allow specifying the subset of RDF entailment rules w.r.t. which completeness is required. This is because the implicit triples brought by some rules, e.g., generalization of constants into blank nodes, may not be very informative in most settings. Of particular interest among all entailment rules are usually those derived from an RDFS, since they encode application domain semantics.

4.2 RDF entailment and query answering

We consider here the two main approaches previously proposed to answer queries w.r.t. a given set of RDF entailment rules: database saturation and query reformulation.

Database saturation The first approach *saturates* the database by adding to it all the implicit triples specified in the RDF recommendation [27]. The benefit of saturation is that standard query evaluation techniques for plain RDF can be applied on the resulting database to compute complete answers [28]. Saturation also has drawbacks. First, it needs more space to store the implicit triples, competing with the data and the materialized views. Observe that saturation adds all implicit triples to the store, whether user queries need them or not. Second, the maintenance of a saturated database, which can be seen as an inflationary fixpoint, when adding or removing data and/or RDFS statements may be complex and costly. Finally, saturation is not always possible, e.g., when querying is performed at a client with no write access to the database.

Query reformulation The second approach *reformulates* a (conjunctive) query into an equivalent union of (conjunctive) queries. The complete answers of the initial query (w.r.t. the considered RDF entailment rules) can be obtained by standard query evaluation techniques for plain RDF [28] using this union of queries

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

Algorithm 1: $\mathbf{Reformulate}(q, \mathcal{S})$

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 database D: $\text{evaluate}(q, \text{saturate}(D, \mathcal{S})) = \text{evaluate}(ucq, D)$

1 $ucq \leftarrow \{q\}$, $ucq' \leftarrow \emptyset$
2 **while** $ucq \neq ucq'$ **do**
3 $ucq' \leftarrow ucq$
4 **foreach** *conjunctive query* $q' \in ucq'$ **do**
5 **foreach** *atom* g *in* q' **do**
6 **if** $g = t(\mathrm{s}, rdf{:}type, c_2)$ *and* c_1 rdfs:subClassOf $c_2 \in \mathcal{S}$ **then**
7 $ucq \leftarrow ucq \cup \{q'_{[g/t(\mathrm{s},rdf:type,c_1)]}\}$ //rule 1
8 **if** $g = t(\mathrm{s}, p_2, \circ)$ *and* p_1 rdfs:subPropertyOf $p_2 \in \mathcal{S}$ **then** $ucq \leftarrow ucq \cup \{q'_{[g/t(\mathrm{s},p_1,\circ)]}\}$ //rule 2
9 **if** $g = t(\mathrm{s}, rdf{:}type, c)$ *and* p rdfs:domain $c \in \mathcal{S}$ **then**
10 $ucq \leftarrow ucq \cup \{q'_{[g/\exists X\ t(\mathrm{s},p,X)]}\}$ //rule 3
11 **if** $g = t(\circ, rdf{:}type, c)$ *and* p rdfs:range $c \in \mathcal{S}$ **then**
12 $ucq \leftarrow ucq \cup \{q'_{[g/\exists X\ t(X,p,\circ)]}\}$ //rule 4
13 **if** $g = t(\mathrm{s}, rdf{:}type, X)$ *and* $c_1, c_2 \ldots, c_n$ *are all the classes in* $\mathcal{S}$ **then** //rule 5
14 $ucq \leftarrow ucq \cup \bigcup_{i=1}^{n} \{(q'_{[g/t(\mathrm{s},rdf:type,c_i)]})_{\sigma=[X/c_i]}\}$
15 **if** $g = t(\mathrm{s}, X, \circ)$ *and* $p_1, p_2 \ldots, p_m$ *are all the properties in* $\mathcal{S}$ **then** //rule 6
16 $ucq \leftarrow ucq \cup \bigcup_{i=1}^{m} \{(q'_{[g/t(\mathrm{s},p_i,\circ)]})_{\sigma=[X/p_i]}\} \cup \{(q'_{[g/t(\mathrm{s},rdf:type,\circ)]})_{\sigma=[X/rdf:type]}\}$
17 **return** ucq

against the non-saturated database.

The benefit of reformulation is leaving the database unchanged. However, reformulation has an overhead at query evaluation time.

Query reformulation w.r.t. an RDFS Query reformulation algorithms have been investigated in the literature for the well-known *Description Logic fragment* of RDF [3, 5]: datasets with RDFSs, without blank nodes, and where RDF entailment only considers the rules associated to an RDFS (those of the third kind in Section 4.1). However, these algorithms allow reformulating queries from a strictly less expressive language than the one of our RDF queries (see Section 7 for more details) and, thus, cannot be applied to our setting. We therefore propose the Algorithm 1 that fully captures our query language, so that we can obtain the complete answers of any RDF query by evaluating its reformulation.

The algorithm uses the set of rules of Figure 2 to *unfold* the queries; in this Figure and onwards, we denote by s, p, respectively, $\circ$, a placeholder for either a constant or a variable occurring in the subject, property, respectively, object position of a triple atom. Notice that rules (1)-(4) follow from the four rules of Table 1. The evaluate and saturate functions, used in Algorithm 1 provide, respectively, the standard query evaluation for plain RDF, and the saturation of a data set w.r.t. an RDFS (Table 1). Moreover, $q_{[g/g']}$ is the result of replacing the atom g of the query q by the atom g' and $q_{\sigma=[X/c]}$ is the result of replacing any occurrence of the variable X in q with the constant c.

More precisely, Algorithm 1 uses the rules in Figure 2 to generate new queries from the original query, by a backward application of the rules on the query atoms. It then applies the same procedure on the newly obtained queries and repeats until no new queries can be constructed. Then, it outputs the union of the generated queries. The inner loop of the algorithm (lines 5-16) comprises six *if* statements, one for each of the six rules above. The conditions of these statements represent the heads (right parts) of the rules, whereas the consequents correspond to their bodies (left parts). In each iteration, when a query atom matches the condition of an *if* statement,

$$t(\mathrm{s}, rdf{:}type, c_1) \Rightarrow t(\mathrm{s}, rdf{:}type, c_2), \text{ with } c_1\ rdfs{:}subClassOf\ c_2 \in \mathcal{S} \quad (1)$$

$$t(\mathrm{s}, p_1, \circ) \Rightarrow t(\mathrm{s}, p_2, \circ), \text{with } p_1\ rdfs{:}subPropertyOf\ p_2 \in \mathcal{S} \quad (2)$$

$$t(\mathrm{s}, p, X) \Rightarrow t(\mathrm{s}, rdf{:}type, c), \text{with } p\ rdfs{:}domain\ c \in \mathcal{S} \quad (3)$$

$$t(X, p, \circ) \Rightarrow t(\circ, rdf{:}type, c), \text{with } p\ rdfs{:}range\ c \in \mathcal{S} \quad (4)$$

$$t(\mathrm{s}, rdf{:}type, c_i) \Rightarrow t(\mathrm{s}, rdf{:}type, X), \text{for any class } c_i \text{ of } \mathcal{S} \quad (5)$$

$$t(\mathrm{s}, p_i, \circ) \Rightarrow t(\mathrm{s}, X, \circ), \text{for any property } p_i \text{ of } \mathcal{S} \text{ and } rdf{:}type \quad (6)$$

Figure 2: Reformulation rules for an RDFS $\mathcal{S}$.

the respective rule is triggered, replacing the atom with the one that appears in the body of the rule. Note that rules 5 and 6 need to bind a variable X of an atom to a constant c_i, p_i, or $rdf{:}type$, thus use σ to bind all the occurrences of X in the query in order to retain the join on X within the whole new query.

We now prove the termination and correctness of $\mathbf{Reformulate}(q, \mathcal{S})$.

THEOREM 4.1 (TERMINATION OF $\mathbf{Reformulate}(q, \mathcal{S})$). *Given a query q over an RDFS $\mathcal{S}$, $\mathbf{Reformulate}(q, \mathcal{S})$ terminates and outputs a union of no more than $(2|\mathcal{S}|^2)^m$ queries, where $|\mathcal{S}|$ is the number of statements in $\mathcal{S}$ and m the number of atoms in q.*

PROOF. For the algorithm to terminate, it suffices to show that rules (1)-(6) can be sequentially applied a finite number of times to each atom of q. Let $|\mathcal{S}|$ be the number of statements in $\mathcal{S}$, $|R|$ be the number of relations (i.e., classes and properties), $|C|$ the number of classes and $|P|$ the number of properties participating in $\mathcal{S}$. Obviously, $|C| + |P| = |R|$. Observe that the atoms resulting from the application of rule 2 can only further trigger the same rule. Likewise, rules 3 and 4 can only trigger rule 2, whereas rule 1 only enables the application of rules 1, 3, and 4. Rule 5 only triggers rules 1, 3, and 4. Lastly, rule 6 may trigger any other rule.

Clearly, the worst case (longest sequence of rule applications) occurs when rule 6 is applied on an atom of the form $t(\mathrm{s}, X, Y)$: it generates $|P|$ atoms (as many as the distinct properties of $\mathcal{S}$) of the form $t(\mathrm{s}, p_j, Y)$ and one atom of the form $t(\mathrm{s}, rdf{:}type, Y)$. Each of the $|P|$ atoms can cause the recursive application of rule 2 as explained before. Rule 2 can be applied at most $|\mathcal{S}|$ times for each of these atoms (in case $\mathcal{S}$ includes only subPropertyOf statements), leading to a total number of $|P||\mathcal{S}|$ generated atoms. As for the atom $t(\mathrm{s}, rdf{:}type, Y)$ also output by rule 6, it can trigger rule 5, which then generates $|C|$ new atoms (as many as the distinct classes in $\mathcal{S}$) of the form $t(\mathrm{s}, rdf{:}type, c_i)$. As explained above, each of these $|C|$ atoms can enable the recursive application of rules 1, 3 and 4 at most $|\mathcal{S}|$ times, leading to $|C||\mathcal{S}|$ atoms. Summing up, we can have at most $|P||\mathcal{S}| + |C||\mathcal{S}| = (|P| + |C|)|\mathcal{S}| = |R||\mathcal{S}|$ rule applications. Now observe that the biggest number of distinct relations that can appear in $\mathcal{S}$ is $2|\mathcal{S}|$, which happens when no relation is used more than once in the statements in $\mathcal{S}$. Thus, the above sum is updated to $2|\mathcal{S}||\mathcal{S}| = 2|\mathcal{S}|^2$ which constitutes the upper bound for the number of reformulations per atom. That is, if q comprises m atoms, $\mathbf{Reformulate}(q, \mathcal{S})$ terminates after at most $(2|\mathcal{S}|^2)^m$ rule applications, leading to an equal number of queries. □

THEOREM 4.2 (CORRECTNESS OF ALGORITHM 1). *Let ucq be the output of $\mathbf{Reformulate}(q, \mathcal{S})$, for a query q over an RDFS $\mathcal{S}$. For any database D associated to $\mathcal{S}$:*
$\text{evaluate}(q, \text{saturate}(D, \mathcal{S})) = \text{evaluate}(ucq, D)$.

PROOF (SKETCH). **Soundness** We have to show that if $t \in \text{evaluate}(ucq, D)$ then $t \in \text{evaluate}(q, \text{saturate}(D, \mathcal{S}))$. From $t \in \text{evaluate}(ucq, D)$, t is an answer to a query q' in ucq. By construction, any query built by Algorithm 1 is subsumed by q w.r.t. $\mathcal{S}$,

so $\text{evaluate}(q', \text{saturate}(D, \mathcal{S})) \subseteq \text{evaluate}(q, \text{saturate}(D, \mathcal{S}))$, thus $t \in \text{evaluate}(q, \text{saturate}(D, \mathcal{S}))$.

Completeness We have to show that if $t \in \text{evaluate}(q, \text{saturate}(D, \mathcal{S}))$ then $t \in \text{evaluate}(ucq, D)$. Since $t \in \text{evaluate}(q, \text{saturate}(D, \mathcal{S}))$, t results from the projection upon m triples $t_1, \ldots, t_m$, given that q has m atoms. We therefore have to show that any t_i is also exhibited by a reformulation of the i-th atom of q.

First, observe that our set of rules is capable of capturing all possible cases of query atoms. Clearly, every atom $t(s, p, o)$ consists of three terms, each being either a variable or a constant. When an atom contains a variable in p, rule (6) is triggered, whereas when p is specified, rules (2) or (5) can be used (depending on whether p is $rdf{:}type$ or not, respectively). Finally, when p is $rdf{:}type$ and o is some constant, rules (1), (3) and (4) can be applied. Hence, all cases of query atoms are treated.

Now, we prove the above claim by induction on the number α of applications of the saturation rules, needed for t_i to be added in $\text{saturate}(D, \mathcal{S})$, those rules being exactly these of Table 1 applied in a forward-chaining fashion [27]. We actually show that the free variables of the i-th atom of q that are bounded by t_i, are equally bounded by the evaluation of a reformulation of the i-th atom of q. For $\alpha = 0$, $t_i \in D$ (i.e., t_i is an explicit triple), thus t_i is also a triple for the evaluation of the non-reformulated i-th atom of q. Suppose that the claim holds for $\alpha < k$, and let us consider the case for $\alpha = k$. Assume t_i is finally added after the application of the first closure rule on a triple $t_{\alpha-1}$. Then, $t_{\alpha-1} = (s_1, rdftype, c_1)$ and $t_i = (s_1, rdftype, c_2)$, where s_1, c_1 and c_2 are constants. Since $t \in \text{evaluate}(q, \text{saturate}(D, \mathcal{S}))$, t_i matches the i-th atom of q, which is, thus, of the form $t(\mathrm{s}, rdftype, c_2)$, $t(\mathrm{s}, X, c_2)$, $t(\mathrm{s}, rdftype, X)$, or $t(\mathrm{s}, X, Y)$.

In the first case, we perform a reformulation of the i-th atom using rule 1 and we obtain the atom $t(\mathrm{s}, rdftype, c_1)$, which indeed returns s_1 in the result, as if the triple t_i was stored in D. In the second case, we reformulate the query atom with rule 6 and we obtain (among others) the atom $t(\mathrm{s}, rdftype, c_2)$, which, after one more reformulation using rule 1, results in the atom $t(\mathrm{s}, rdftype, c_1)$ that was treated by the first case. In the third case, we apply rule 5, we immediately obtain the atom $t(\mathrm{s}, rdftype, c_1)$ and so we also return s_1 in the result. In the last case, we apply rule 6 and on the new atom $t(\mathrm{s}, rdftype, X)$ we apply rule 5 and then fall into the previous case.

Thus, for all four cases that can appear after applying the first saturation rule, we have proved by induction our claim. We proceed the same way for the three other saturation rules. □

4.3 View selection aware of RDF entailment

We now discuss possible ways to take RDF implicit triples into account in our view selection approach. As will be explained, the exact way (cardinality) statistics are collected for each view atom, described first in Section 3.4, play an important role here.

Database saturation If the database is saturated prior to view selection, the collected statistics do reflect the implicit triples.

Pre-reformulation Alternatively, one could reformulate the query workload and then apply our search on the new workload. To do so, we extend the definition of our initial state, as well as our rewriting language to that of *unions* of conjunctive queries. More precisely, given a set of queries $Q = \{q_1, \ldots, q_n\}$, and assuming that $\text{Reformulate}(q_i, \mathcal{S}) = \{q_i^1, \ldots, q_i^{n_i}\}$, it is sufficient to define $S_0(Q) = \langle V_0, R_0 \rangle$ as the set of conjunctive views $V_0 = \bigcup_{i=1}^{n} \{q_i^1, \ldots, q_i^{n_i}\}$ and the set of rewritings $R_0 = \bigcup_{i=1}^{n} \{q_i = q_i^1 \cup \cdots \cup q_i^{n_i}\}$. In this case, statistics are collected on the original (non-saturated) database for the reformulated queries.

$q^{1,\mathcal{S}}$	$q^1(X_1)$	$:\!-t(X_1, rdf{:}type, picture)$	(1)
	$\cup\, q^1(X_1)$	$:\!-t(X_1, rdf{:}type, painting)$	(2)
$q^{4,\mathcal{S}}$	$q^4(X_1, X_2)$	$:\!-(X_1, X_2, picture)$	(1)
	$\cup\, q^4(X_1, isLocatIn)$	$:\!-t(X_1, isLocatIn, picture)$	(2)
	$\cup\, q^4(X_1, isExpIn)$	$:\!-t(X_1, isExpIn, picture)$	(3)
	$\cup\, q^4(X_1, rdf{:}type)$	$:\!-t(X_1, rdf{:}type, picture)$	(4)
	$\cup\, q^4(X_1, isLocatIn)$	$:\!-t(X_1, isExpIn, picture)$	(5)
	$\cup\, q^4(X_1, rdf{:}type)$	$:\!-t(X_1, rdf{:}type, painting)$	(6)

Table 2: Term reformulation for post-reasoning.

As stated in Theorem 4.1, query reformulation can yield a significant number of new queries, increasing the number of views of our initial state and leading to a serious increase of the search space. As an example, the following simple query on the Barton [25] dataset

$$q(X_1, X_2, X_3) :\!- t(X_1, rdf{:}type, text), t(X_1, relatedTo, X_2), t(X_2, rdf{:}type, subjectPart), t(X_1, language, fr), t(X_2, description, X_3)$$

is reformulated with the Barton schema into a union of 104 queries. Given the very high complexity of the exhaustive search problem (Section 5.1), such an increase may significantly impact view selection performance.

Post-reformulation To avoid this explosion, we propose to apply reformulation not on the initial queries but directly on the views in the final (best) state recommended by the search.

Directly doing so introduces a source of errors: since statistics are collected directly on the original database, and the queries are not reformulated, the implicit triples will not be taken into account in the cost estimation function c^ϵ. To overcome this problem, we reflect implicit triples *to the statistics, by reformulating each view atom v^i into a union of atoms* $\text{Reformulate}(v^i, \mathcal{S})$ *prior to the view search, and then replacing $|v^i|$ (i.e., the cardinality of v^i) in our cost formulas with* $|\text{Reformulate}(v^i, \mathcal{S})|$. This results in having the same statistics as if the database was saturated. Then, we perform the search using the (non-reformulated) queries and get the same best state as in the database saturation approach (as we use the same initial state and statistics). Since materializing the best state's views directly would not include the implicit triples, we need to reformulate these views first. Theorem 4.2 guarantees the correctness of post-reformulation (materializing the reformulated views on the non-saturated database is the same as materializing the non-reformulated ones on the saturated database).

Consider the query q of Section 3.4, with the following schema:

$$\mathcal{S} = \{painting \text{ rdfs:subClassOf } picture, \; isExpIn \text{ rdfs:subPropertyOf } isLocatIn\}$$

We first count (see Section 3.4) the exact number of triples matching the query atoms and their relaxed versions, namely q^1 to q^5.

We now reformulate each q^i based on $\mathcal{S}$ into a union of queries, denoted $q^{i,\mathcal{S}}$. Table 2 illustrates this for q^1 and q^4 . Rule 1 (Figure 2) has been applied on q^1, adding to it a second union term. Applying rule 6 on q^4 leads to replacing X_2 with $isLocatIn$, $isExpIn$, and $rdf{:}type$ respectively in the second, third and fourth union terms of $q^{4,\mathcal{S}}$. In turn, the second term triggers rule 2 producing a fifth term, while the fourth term triggers rule 1 to produce the sixth union term.

The cardinality of each reformulated atom $q^{i,\mathcal{S}}$ is estimated prior to the search. Then, we perform the search for the non-reformulated version of q using these statistics, and get the following best state:

$$v_1(X_1, X_2) :\!- t(X_1, rdf{:}type, X_2), \; v_2(X_1, X_2) :\!- t(X_1, isLocatIn, X_2)$$
$$r_3 = \pi_{v_1.X_1, v_2.X_2}(\sigma_{X_2 = picture}(v_1) \bowtie_{v_1.X_1 = v_2.X_1} v_2)$$

After the search has finished, instead of the recommended views v_1 and v_2, we materialize their reformulated variants v_1' and v_2':

$$v_1'(X_1, X_2) :- t(X_1, rdf{:}type, X_2)$$
$$\cup\, v_1'(X_1, painting) :- t(X_1, rdf{:}type, painting)$$
$$\cup\, v_1'(X_1, picture) :- t(X_1, rdf{:}type, picture)$$
$$\cup\, v_1'(X_1, picture) :- t(X_1, rdf{:}type, painting)$$

$$v_2'(X_1, X_2) :- t(X_1, isLocatIn, X_2)$$
$$\cup\, v_2'(X_1, X_2) :- t(X_1, isExpIn, X_2)$$

Executing r_3 on v_1' and v_2' provides the complete answers for q.

In post-reformulation, finding the best state does not require saturating the database, nor multiplying the queries (as pre-reformulation does) and making the search space size explode. Thus, this is the best approach for situations where database saturation is not an option, which is also shown through our experiments in Section 6.5.

Some measurements about the cost of the statistics collection for each of the three reasoning approaches are also given in Section 6.5.

5. SEARCHING FOR VIEW SETS

This Section discusses strategies for navigating in the search space of candidate view sets (or states), looking for a low- or minimal-cost state. We discuss the exhaustive search strategies and identify an interesting subset of *stratified* strategies in Section 5.1, based on which we analyze the size of the search space. In Section 5.2, we present several efficient optimizations and search heuristics.

5.1 Exhaustive Search Strategies

We define the *initial state* of the search as $S_0(Q) = \langle V_0, R_0 \rangle$, such that $V_0 = Q$, i.e., the set of views is exactly the set of queries, and each rewriting in R_0 is a view scan. The state graph $G(S_0)$ corresponds to the queries in Q. Clearly, the rewriting cost of S_0 is low, since each query rewriting is simply a view scan. However, its space consumption and/or view maintenance costs may be high.

We denote by $S \xrightarrow{\tau} S'$ the application of the transition $\tau \in \{\text{SC},\text{JC},\text{VB},\text{VF}\}$ on a state S, leading to the state S'.

DEFINITION 5.1 (PATH). *A path is a sequence of transitions of the form:* $S_0 \xrightarrow{\tau_0} S_1$, $S_1 \xrightarrow{\tau_1} S_2$, …, $S_{k-1} \xrightarrow{\tau_{k-1}} S_k$.

For instance, in Figure 3, $(S_0 \xrightarrow{\text{SC}(c_2)} S_3), (S_3 \xrightarrow{\text{JC}} S_6)$ is a path. We may denote a path simply by its transitions, e.g., (SC(c_2), JC).

It can be shown that any path is cycle-free. The intuition is that SC and JC remove query-specified predicates from the views, and no transition ever brings them back. Similarly, VB and JC create ever smaller views, while no transition replaces a view (or two views) by a larger one. It follows that any path is of finite length.

THEOREM 5.1 (COMPLETENESS OF THE TRANSITION SET). *Given a workload Q and an initial state S_0, for every possible state $S(Q)$, there exists a path from the initial state S_0 to S.*

PROOF. Given a workload Q, an initial state S_0 and a possible state $S(Q) = \langle V, R \rangle$ that corresponds to a candidate view set for Q, we have to show that S can be attained through a sequence of transitions, all of which belong to our transition set $\{\text{SC},\text{JC},\text{VB},\text{VF}\}$.

By the definition of the candidate view set, we know that from every view v, there exists an embedding ϕ into at least one query $q_v \in Q$, otherwise, v would not be usable in any query rewriting. If ϕ mapped two atoms of v to the same atom of q_v, then v would be non-minimal, which contradicts our definition of the candidate view set. Thus, ϕ maps each v atom to a distinct q_v atom, which entails that v has at most as many atoms as q_v.

To start with, assume that our workload Q consists of a single query q. Each state consists of a possible view set that can be used to rewrite q. We initially assume that no rewriting uses a specific view more than once. We now distinguish two cases depending on the number of views $|V|$ in S:

$|\mathbf{V}| = \mathbf{1}$. In this case q is rewritten based only on one view v. Then v must have the same number of atoms as q: we have already shown that v cannot have more atoms than q; if v had less atoms than q, it could not suffice to rewrite q. Moreover, the graph of v has to be a subgraph of q (i.e., less restrictive than q) in order to be able to answer q. By repeatedly applying a number of SC and JC starting from q, we can obtain all possible subgraphs of q, among which we find the graph of the given view v. This means that in this case we can reach S starting from S_0.

$|\mathbf{V}| \geq \mathbf{1}$. Now assume that V contains $k > 1$ views, all of which participate to the (single) rewriting r of q. Hence, all views can be embedded into q. Notice though, that for every two views $v_1, v_2 \in V$, the nodes of q to which v_1 is mapped cannot be a subset of the nodes of q to which v_2 is mapped, because this entails the existence of a rewriting r' which does not use v_1, making r non-minimal, which contradicts our assumptions[3]. Starting from the initial view $q \in S_0$, we can use a sequence of JC and VB to split q into a set of views V_q, containing exactly k views. The k views of V_q are created so as to establish a one-to-one correspondence between V and V_q, as follows. For each view $v \in V$, we create exactly one view $v_1 \in V_q$ consisting of the q atoms to which v is mapped. This means that by construction v can be embedded into v_q. Thus, the graph of v is a subgraph of v_q and since they have the same number of nodes (as v and v_q have the same number of atoms), we can reach v from v_q by a sequence of SC and JC, as explained in the previous case when $|V| = 1$. Thus, starting from q we have shown how we can create the set V and reach from S_0 the state S.

We now turn to the case when we still have one query q in our workload, but the rewriting of q can use the same view more than once. We construct a new state S' in which we have created for each view of the view set V of S as many copies as the number of times it is used in the rewriting of q. Thus, in S' the rewriting of q includes views that are used only once. Reaching S from S' can be done easily by applying a sequence of VFs which revert the view duplication steps that brought us to S'. Moreover, showing that S' is attainable from S_0 falls into the case when $|V| \geq 1$ that was described above. Hence, S can be indeed attained from S_0.

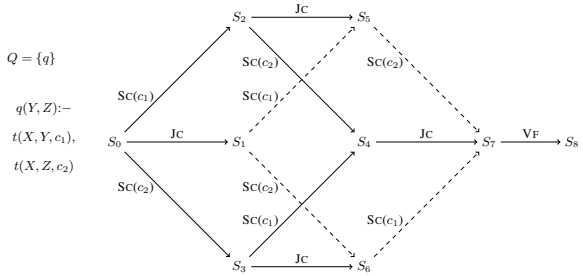
Assume now that we have more than one, say m, queries in Q. The set of rewritings R of the given state S contains m rewritings. We distinguish two cases:

No view in V is used in more than one rewriting. In this case, we can treat each query $q \in Q$ as a separate initial state and, as explained above, construct all the views that participate in the rewriting of q. This way we will obtain m new states, one for every query. Once this is done, we combine the m states and create a single state that has as views the union of views of the individual "one-query" states, and as rewritings the union of rewritings of the m states. This is equivalent to the state S.

Some views in V are used in multiple rewritings. Given a state S, we construct a new state S' in which we copy each V view that participated in n_v rewritings in S, into n_v distinct views, each of which participates to exactly one rewriting in S'[4]. It follows from the case above (when no view could be used in more than one rewriting) that S' can be reached from S_0 using our set of transitions. Moreover, one can easily reach S from S' by simply applying a sequence of VFs which revert the view duplication steps we took to obtain S'. Thus, we have finished showing that S can be

[3]This is also the reason we impose the restriction in the definition of VB that when a view is broken into two new views with node sets N_1 and N_2, we should have $N_1 \nsubseteq N_2$ and $N_2 \nsubseteq N_1$.

[4]Observe that our definitions of views and candidate view sets does not preclude the existence of two identical views in the same candidate view set, as long as each view is minimal.



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 3: Sample exhaustive strategy (solid arrows), ExNaïve strategy (solid and dashed arrows), and view sets corresponding to each state.

obtained from S_0. □

Definition 5.2 (Strategy). *A search strategy Σ is a sequence of transitions 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 $j \in [1..k]$ $\tau_{i_j} \in \{\text{Sc}, \text{Jc}, \text{Vb}, \text{Vf}\}$, and for every $j \in [2..k]$ there exists $l < j$ such that $S'_{i_l} = S_{i_j}$ (each state but S_0 must be attained before it is transformed).

For example, for the one-query workload depicted at the top left of Figure 3, one possible strategy is:

$$\Sigma_1 = (S_0 \xrightarrow{\text{Sc}(c_1)} S_2), (S_2 \xrightarrow{\text{Sc}(c_2)} S_4), (S_0 \xrightarrow{\text{Sc}(c_2)} S_3), (S_3 \xrightarrow{\text{Sc}(c_1)} S_4), (S_0 \xrightarrow{\text{Jc}} S_1)$$

A strategy Σ is *exhaustive* if any state S that can be reached through a path, is also reached in Σ (not necessarily through the same path). For instance, in Figure 3, the solid arrows depict an exhaustive strategy, reaching all possible states.

We first consider a simple family of strategies called ExNaïve and described through Algorithm 2. ExNaïve strategy (as all strategies presented in this work) maintains a *candidate state set* CS and a set of *explored states* ES. CS keeps the states on which more transitions can be possibly applied and is initially $\{S_0\}$. ES is disjoint from CS and is empty in the beginning. A state S is explored, when any state $S' = \tau(S)$ obtained by applying some transition $\tau \in \{\text{Sc}, \text{Jc}, \text{Vb}, \text{Vf}\}$ to S, already belongs either to CS or to ES. ExNaïve at each point picks a state S_c from CS and tries to apply a transition to it ($applyTrans$, line 4). If no new state is obtained, S_c was already explored and is moved to ES (line 5); otherwise, the newly obtained state (S_{new}) is copied to CS (line 7). During the search, we also keep the *best state* found so far (denoted S_b), i.e., having the lowest cost $c^\epsilon(S)$ (line 8). The strategy stops when no new states can be found. Clearly, ExNaïve strategies are exhaustive. In Figure 3, the solid and dashed arrows, together, illustrate an ExNaïve strategy.

Note that checking whether a state S belongs to CS or ES is done using the signature-based filter presented in Section 3.3. To this end, we organized the CS and ES sets as hash maps where on a state signature we keep one or several states. This speeds up the process of look-up, since we search in CS and ES directly by a look-up on S's signature.

Algorithm 2: ExNaïve(S_0)

Input : an initial state S_0
Output: the best state S_b found

```
1 S_b ← S_0, S_new ← null, CS ← {S_0}, ES ← ∅, NS ← ∅
2 while CS ≠ ∅ do
3     foreach state S_c ∈ CS do
4         S_new ← applyTrans({Sc,Jc,Vb,Vf}, S_c, (ES ∪ CS))
5         if S_new = null then move S_c from CS to ES
6         else
7             CS ← CS ∪ {S_new}
8             if c^ε(S_new) < c^ε(S_b) then S_b ← S_new
```

For a given strategy Σ, the *paths to a state* $S \in \Sigma$, denoted $^{\hookrightarrow}S$, is the set of all Σ paths whose final state is S. In an ExNaïve strategy there may be multiple paths to some states, e.g., S_6 is reached twice in our example, which slows down the search. We define the notion of *stratification* to reduce the number of such duplicate states.

Definition 5.3 (Stratified path). *A path $p \in {}^{\hookrightarrow}S$ for some state $S \in \Sigma$ is stratified iff it belongs to the regular language:* Vb* Sc* Jc* Vf*.

A stratified path constrains the order among the types of transitions on the path: all possible view breaks appear only in the beginning of the path and are followed by the selection cuts. Join cuts appear only after all selection cuts are applied and are in turn followed by zero or more view fusions. In Figure 3, all solid-arrow paths starting from S_0 are stratified.

The following theorem formalizes the interest of stratified paths.

Theorem 5.2 (Completeness of stratified paths). *Let Q be a query workload and $S(Q)$ be a state for Q. There exists a stratified path leading from the initial state S_0 to S.*

Proof. If S is a state for Q, due to Theorem 5.1, there exists a path p belonging to some strategy Σ (for instance, an ExNaïve strategy) reaching S. We show that if p is not stratified, it can be transformed into a stratified path p', that also has S as its final state. Notice that paths including a single transition are always stratified, so hereafter, we focus only on paths of bigger length.

For a given transition τ appearing in a path p, we define the *forward inversion count of τ in p* (denoted $FIC(\tau, p)$) as the number of transitions that appear *after* τ in p, and that should appear *before* τ if p was stratified. For instance, consider the path $p_1 =$ (Sc_1, Vf_1, Jc_1, Vf_2, Sc_2, Vb_1, Jc_2) where the states are not shown, and transitions of the same kind are distinguished by their subscripts. We have $FIC(\text{Sc}_2, p_1) = 1$, since Vb_1 appears after Sc_2; the other transition appearing after Sc_2, namely Jc_2, does not violate stratification. Similarly, $FIC(\text{Jc}_1, p_1) = 2$ since Sc_2 and Vb_1 appear after Jc_1 in p_1; moreover, $FIC(\text{Vb}_1, p_1) = 0$ and $FIC(\text{Vf}_2, p_1) = 3$. Clearly, for any $\tau \in p$, $0 \leq FIC(\tau, p) < |p|$, where $|p|$ is the number of transitions in p.

Further, we define the forward inversion count of a path p as the sum of all the FICs of the transitions in p, that is: $FIC(p) = \sum_{\tau \in p} FIC(\tau, p)$. Path p is stratified if and only if $FIC(p) = 0$.

We now turn to consider our non-stratified path p. From the definition of stratified paths, one easily derives a set of six elementary stratification violations: these are path fragments of the form (τ_1, τ_2), each of which contradicts stratification, and at least one of which must be present in any non-stratified path. For each such violating fragment p_v going from a state S_1 to another state S_3, we provide another path fragment p_s going from S_1 to S_3, and which is stratified.

Case 1 ($p_v = S_1 \xrightarrow{\text{Jc}} S_2 \xrightarrow{\text{Sc}} S_3$). By the semantics of Sc and Jc, it follows readily that, since Sc and Jc target different edges of the

state graph, there exists a state S_1', such that we can now reach S_3 through the stratified path $p_s = S_1 \xrightarrow{\text{SC}} S_1' \xrightarrow{\text{JC}} S_3$.

Case 2 ($p_v = S_1 \xrightarrow{\text{VF}} S_2 \xrightarrow{\text{SC}} S_3$). Here we distinguish two cases:

- If VF and SC are performed at different places of S_1 (VF fuses two views, while SC cuts an edge from a third, distinct view), then we can apply them in the inverse order and reach S_3 with the stratified path $p_s = S_1 \xrightarrow{\text{SC}} S_1' \xrightarrow{\text{VF}} S_3$. This case is similar to Case 1 above.
- In p_v, when SC erases a selection on a constant appearing in the fused view resulting from VF , we need to apply two SC steps prior to VF in order to attain S_3 from S_1 through the stratified path $p_s = S_1 \xrightarrow{\text{SC}} S_1' \xrightarrow{\text{SC}} S_2' \xrightarrow{\text{VF}} S_3$.

Case 3 ($p_v = S_1 \xrightarrow{\text{VF}} S_2 \xrightarrow{\text{JC}} S_3$). Three sub-cases can occur:

- When VF and JC are applied on different views, we can reach reach S_3 through the stratified $p_s = S_1 \xrightarrow{\text{JC}} S_1' \xrightarrow{\text{VF}} S_3$.
- When JC is applied on the view resulting from VF and it does not disconnect this view, we need the stratified path $p_s = S_1 \xrightarrow{\text{JC}} S_1' \xrightarrow{\text{JC}} S_2' \xrightarrow{\text{VF}} S_3$, whereas
- if it disconnects the view, we need two JC steps and then two VF, that is the path $p_s = S_1 \xrightarrow{\text{JC}} S_1' \xrightarrow{\text{JC}} S_2' \xrightarrow{\text{VF}} S_3' \xrightarrow{\text{VF}} S_3$.

Case 4 ($p_v = S_1 \xrightarrow{\text{SC}} S_2 \xrightarrow{\text{VB}} S_3$). In a way similar to the above cases, if SC and VB affect different views, it suffices to use the stratified path $p_s = S_1 \xrightarrow{\text{VB}} S_1' \xrightarrow{\text{SC}} S_3$, while when VB is performed on the view resulting from SC, we need the new path $p_s = S_1 \xrightarrow{\text{VB}} S_1' \xrightarrow{\text{SC}} S_2' \xrightarrow{\text{SC}} S_3$.

Case 5 ($p_v = S_1 \xrightarrow{\text{JC}} S_2 \xrightarrow{\text{VB}} S_3$). In the simple case that JC and VB are applied on different views, we can reach S_3 through the stratified path $p_s = S_1 \xrightarrow{\text{VB}} S_1' \xrightarrow{\text{JC}} S_3$. When JC and VB affect the same view, we distinguish the following sub-cases:

- Assume that JC does not disconnect the view on which it is applied. Notice that, by definition, VB may remove some of the edges of the view to which it is applied. If the edge that JC removes, would also be removed by VB (if it is applied prior to JC), then we can omit JC and reach S_3 through the path $p_s = S_1 \xrightarrow{\text{VB}} S_3$. In the opposite case, JC removed an edge that has to be also removed from both the views output by VB. Thus, we need the stratified path $p_s = S_1 \xrightarrow{\text{VB}} S_1' \xrightarrow{\text{JC}} S_2' \xrightarrow{\text{JC}} S_3$.
- If JC disconnected the view, then the sequence of JC followed by VB created lead to three views, and JC was applied in a different part of the initial view. Thus, we can simply use the inverse order of transitions and still reach S_3, through the stratified path $p_s = S_1 \xrightarrow{\text{VB}} S_1' \xrightarrow{\text{JC}} S_3$.

Case 6 ($p_v = S_1 \xrightarrow{\text{VF}} S_2 \xrightarrow{\text{VB}} S_3$). In this case, if VF and VB affect different views, we need the stratified path $p_s = S_1 \xrightarrow{\text{VB}} S_1' \xrightarrow{\text{VF}} S_3$, while when VB is performed on the view resulting from VF, we need the new path $p_s = S_1 \xrightarrow{\text{VB}} S_1' \xrightarrow{\text{VB}} S_2' \xrightarrow{\text{VF}} S_3' \xrightarrow{\text{VF}} S_3$.

Applying one of the path substitutions above on a sub-path $p_v = (\tau_1, \tau_2)$ of p turns it into a new path p', in which p_v is replaced with a subpath p_s of one of the following forms: (τ_2', τ_1'), $(\tau_2', \tau_1', \tau_1'')$, $(\tau_2', \tau_2'', \tau_1')$ or $(\tau_2', \tau_2'', \tau_1', \tau_1'')$[5]. In all cases, we have:

$$FIC(\tau_1', p') = FIC(\tau_1'', p') = FIC(\tau_1, p) - 1$$
$$FIC(\tau_2', p') = FIC(\tau_2'', p') = FIC(\tau_2, p)$$

In other words, the FIC of the first among the two transitions is diminished by 1. This is because (i) the path substitution ensures that τ_2' (and τ_2'') no longer contributes to the FIC of τ_1' (and τ_1''), (ii) no other transition introduced by the path substitution contributes to $FIC(\tau_1)$ and (iii) the FICs of p transitions after the end of p_v are unaffected by the substitution.

To turn p into a stratified path p', our first step is to bring all VFs at the end of the path. To do so, we repeatedly identify the last VF transition in the current path p_c, call it VF_k, for which $FIC(\text{VF}_k, p_c) \neq 0$ and apply on VF_k and its successor transition, the path substitution which is appropriate (one of Cases 2, 3 and 6 is sure to apply). Each step reduces $FIC(\text{VF}_k, p_c)$ by 1. At the end of this step, the FIC of all VF transitions in the transformed path is 0, which also means that all VFs are placed at the end of the path.

We now continue the procedure for the JCs: we identify the last occurrence of JC, call it JC_k, for which $FIC(\text{JC}_k, p_c) \neq 0$ and we apply Case 1 or 5, depending on its successor transition. When this step is finished, we repeat the procedure for SCs to place them immediately before the JCs. We do not need to do the same for VBs, as previous operations have already pushed them at the beginning of the resulting path p' which, by now, is stratified, i.e. for all its transitions τ we will have $FIC(\tau, p') = 0$ and, thus, $FIC(p') = 0$.

Notice though that some path substitutions increase the length of the path, e.g. (SC,VB) may be replaced by (VB,SC,SC). However, for all violations there exists one case that does not affect the length of the path. At the same time, a bound holds on the maximal path size, as follows:

- The number of VBs that can be applied successively on a given state is limited, because each time the views that result from a VB have less atoms than the view VB is applied on.
- SCs and JCs are bound by the number of edges in the graphs of the views, which is finite.
- An infinite number of VFs would mean an infinite number of initial queries, which is a contradiction.

Thus, the above procedure always terminates and results in a stratified path p'. □

We can now identify an interesting family of strategies.

DEFINITION 5.4 (STRATIFIED STRATEGY). *A strategy Σ is stratified iff for any $S \in \Sigma$ and $p \in \hookrightarrow S$, p is stratified.*

In Figure 3, any topological sort of the solid edges is a stratified strategy, more efficient than the EXNAÏVE one illustrated in the Figure, since the latter performs four extra transitions. Observe that a stratified strategy does not constrain the order of transitions that *are not on the same path*. For instance, in Figure 3, a stratified strategy may apply the transition $S_0 \xrightarrow{\text{JC}} S_1$ before all the SCs.

We now define the important family of EXSTR strategies. Starting from the initial state S_0, an EXSTR strategy picks any state on which it applies any applicable transition, *preserving the stratification of all strategy paths*. Several EXSTR strategies may exist for a workload, differing in their ordering of the transitions. We will simply use EXSTR to refer to any of them. The EXNAÏVE strategy (Algorithm 2) can be turned to an EXSTR one through the following modification: when $applyTrans$ (line 4) is called on a state S_c, it should apply the transitions in a stratified way, i.e., first it attempts a VB and only if no new state is obtained, it applies an SC, and then a JC and, finally, a VF.

[5]Notice that τ_1', τ_1'' are of the same kind as τ_1, and τ_2', τ_2'' are of the same kind as τ_2.

THEOREM 5.3 (INTEREST OF EXSTR). *(i) Any* EXSTR *strategy is exhaustive. (ii) For a given workload Q, and arbitrary* EXSTR *strategy Σ_S and* EXNAÏVE *strategy Σ_N, Σ_S has at most the number of transitions of Σ_N.*

PROOF. Exhaustiveness of EXSTR strategies follows from the fact that any state can be reached by a stratified path (Theorem 5.2), and that EXSTR only stops when no more states can be discovered. Moreover, Σ_S disables some transitions by restricting the outgoing transitions of a state S depending on S's incoming paths $\hookrightarrow S$, whereas Σ_N allows these transitions. □

Due to Theorem 5.3, among the exhaustive strategies, we will only consider *wlog* the stratified ones.

Size of the search space We quantify the size of the search space by the number of states that can be reached through our transitions for a given query workload Q. Due to Theorem 5.1, this number is equal to the number of all possible states for Q.

We start the analysis by considering that Q consists of a single query q of n atoms. Consider a possible state $S(Q) = \langle V, R\rangle$. Every view $v \in V$ participates in the rewriting r of q and can, thus, be embedded into it. This means that the graph of v is a subgraph of the graph of q. Since r is a complete rewriting of q, the union of the nodes of the graphs of the views in V is equal to the node set of the graph of q. In other words, the node sets of the view graphs constitute a cover for the node set of q.

However, the set of nodes of a view graph does not uniquely determine a view: we can have more than one different states that have the same node sets for their views. This occurs because they may have different selection and join edges. For instance, consider the query $q(Y) = t(c_1, X, c_2), t(X, c_3, Y)$ and the view sets $V_1 = v_1, v_2$ and $V_2 = v_3, v_2$, where $v_1(X_1) = t(c_1, X_1, c_2)$, $v_2(X_2, Y) = t(X_2, c_3, Y)$ and $v_3(X_1) = t(c_1, X_1, Z)$. In this case, the graphs of the views of V_1 and of V_2 are the same cover of the nodes of the query graph, but the views are not the same (v_1 has one more selection edge than v_3).

To this end, we define a restricted class of states C_r, the view sets of which can be exclusively determined by their node sets. Assume a cover of the node set of the query graph. For each subset of nodes N_s in this cover, we construct a graph G_s which also has N_s as its node set. As for the edge set of G_s, it includes all the selection edges of the query graph that are incident to the nodes of N_s, as well as all the join edges between these nodes. Notice that there are some cases in which the graph that is created is not connected. These graphs correspond to views with Cartesian products and, thus, do not lead to valid rewritings in our setting. However, in the worst case that a query is a clique, every possible subset of its node, represents a connected graph. Moreover, the cover of the query node set has to be minimal, otherwise our rewritings will not be minimal.

Observe that the states in C_r are those obtained by applying only VBs and from the JCs only those that disconnect the graph of a view. They are the most restricted states (having views with the biggest number of selection and join edges) given a specific cover of the query nodes, as no additional SCs or JCs are applied on them.

Given the above, the problem of enumerating the states in C_r is reduced to the problem of finding the minimal covers of the query node set. Hence, the number of states in C_r is bound by the number of minimal covers.

Let $\mu(n, k)$ be the number of minimal covers with k members of a set of n elements, given by the following formula:

$$\mu(n,k) = \frac{1}{k!}\sum_{m=k}^{a_k}\binom{2^k-k-1}{m-k} m!\, S(n,m)$$

where $S(n, m)$ is the Stirling number of the second kind (i.e., the number of ways to partition a set of n objects into m groups) and $a_k = \min(n, 2^k - 1)$. The overall upper bound of the number of states in C_r, given one query of n atoms is:

$$NS_r(q,n) = \sum_{k=1}^{n}\mu(n,k)$$

For each state $S_r \in |C_r|$, we can obtain more states through SCs and JCs (those JCs that do not disconnect the graphs). We now need to compute how many such states can we have for a given S_r. To do so, we will first compute the number of nodes of the views in S_r and subsequently the number of selection and join edges. Assume that the query q has n nodes and the view set of S_r comprises k views. In the best case we will have n nodes in total in the views and this happens when all the views were created through JCs (no view node was duplicated). In the worst case, each of the views should cover as many of the query nodes as possible, but without having a view covering a subset of the query nodes of another (due to the need for minimal rewriting). This means that each of the k views should cover $n-k$ query nodes: if a view covered $n-k+1$ then there would exist another view that would cover a subset of the query nodes of this view. For each node we can have at most 3 selection edges. Hence, each of the k views has $3(n-k)$ selection edges. As for the join edges, in the worst case the view will be a clique, having $\frac{(n-k)(n-k-1)}{2}$. To this end, the k views will have $|e_{sj}| = k\left(3(n-k) + \frac{(n-k)(n-k-1)}{2} = \frac{k(n-k)(n-k+5)}{2}\right)$ edges leading to at most $2^{|e_{sj}|}$ for each S_r state.

We define the class of states C_{re} that includes all the states being obtained from the initial one by applying any possible sequence of VB, SC and JC transitions. For this class we now have:

$$NS_{re}(q,n) = \sum_{k=1}^{n} 2^{|e_{sj}|}\mu(n,k)$$

Finally, on each state $S_{re} \in C_{re}$, we can apply a sequence of VFs to obtain new states. In the worst case, any subset of the views in the view set V_{re} of S_{re} consists of isomorphic views. Thus, the number of states resulting from S_{re} by applying VF is bound by the number of partitions of the view set of V_{re}. Assuming the size of V_{re} is k and denoting by B_k the Bell number (the number of partitions of a set of size k), an upper bound for the total number of distinct states belonging to the complete class of states, denoted C_{ref}, is:

$$NS_{ref}(q,n) = \sum_{k=1}^{n} 2^{|e_{sj}|}\mu(n,k)B_k$$

We now escalate to the general case when our workload Q consists of n_q queries. Assume each query q_i has n_i atoms, where $1 \le i \le n_q$ and $\sum_{k=1}^{n_q} n_i = n$. Then the total number of states is:

$$NS_{ref}(Q,n) = \sum_{k=1}^{n_q} NS_{ref}(q_i, n_i)$$

Clearly, the above sum is asymptotically bound by the query with the biggest number of atoms. Thus, the worst case is to have a single query in the workload, because this will lead to the biggest number of atoms per query for a given number of atoms. Thus, in the worst case we have $NS_{ref}(Q,n) = NS_{ref}(q,n)$, where q the single query of the workload.

Time complexity The time complexity of exhaustive search can be derived from the number of states created by each transition and the time complexity of the transition. The cost of a SC, JC and VB is

linear in the size of the largest view, which is bound by $3n$, whereas VF requires checking query equivalence, which is in $O(2^n)$ [7].

The complexity of exhaustive search is very high and, even if views are selected off-line and thus time is not a concern, it brings real issues due to memory limitations. This highlights the need for robust strategies with low memory needs, and efficient heuristics.

5.2 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.

Depth-first search strategies (DFS) 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 VB*. Immediately after S is reached, Σ enumerates all states recursively attainable from S by SC only. This process is then repeated with JC and then with VF. The pseudocode of DFS can be obtained by replacing lines 3-4 of Algorithm 2 with the following ones, where $recApplyTrans$ returns all states that can be reached by a specific transition starting from a given state:

```
foreach state S_VB ∈ {recApplyTrans(VB, S_0)} do
    foreach state S_SC ∈ {recApplyTrans(SC, S_VB)} do
        foreach state S_JC ∈ {recApplyTrans(JC, S_SC)} do
            foreach state S_VF ∈ {recApplyTrans(VF, S_JC)} do
                ...
```

For instance, in Figure 3, the following strategy Σ_3 is DFS:

$$\Sigma_3 = (S_0 \xrightarrow{\text{SC}(c_1)} S_2), (S_2 \xrightarrow{\text{SC}(c_2)} S_4), (S_4 \xrightarrow{\text{JC}} S_7),$$
$$(S_7 \xrightarrow{\text{VF}} S_8), (S_0 \xrightarrow{\text{SC}(c_2)} S_3), (S_3 \xrightarrow{\text{JC}} S_6)$$

An advantage of DFS strategies is that they fully explore each obtained state more quickly, reducing the number of states stored in CS. This results in a significant reduction of the maximum memory needs during the search compared, e.g., with EXNAÏVE, which develops a huge number of candidates before fully exploring them.

Aggressive view fusion (AVF) This technique can be included in any strategy and is based on the fact that VF can only decrease the overall cost of a state (Section 3.4). Once a new state S is obtained through some SC, JC or VB, we recursively apply on S all possible VFs (until no more views can be fused). It can be shown that such repeated VFs converge to a single state S^{VF}. We then discard all intermediate states leading from S to S^{VF} and add only S^{VF} to CS. Thus, AVF preserves the optimality of the search, all the while eliminating many intermediary states whose estimated cost is guaranteed to be higher than that of S^{VF}. For example, assume we reach a state S containing three identical views. We apply a VF on S fusing two of the three views and obtain the state S'. We then apply a VF on S' fusing the two remaining identical views and obtain S^{VF}. AVF discards S' and keeps only S^{VF} to continue the search.

Greedy stratified (GSTR) This strategy starts by applying all possible VB transition sequences on S_0. It then discards *all the obtained states but* S_b, and repeatedly applies on it all possible SC. Keeping only S_b, it proceeds in the same way by applying JC and then VF. The interest of GSTR lies in the possibility to combine it with the AVF technique, leading to the GSTR-AVF strategy. GSTR-AVF has low memory needs due to the many states dropped by GSTR and AVF and moves fast towards lower-cost states due to AVF. Although neither GSTR nor GSTR-AVF can guarantee optimality, they perform well in practice, as our experiments show.

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

- $stop_{tt}(S)$: 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. In this case, $stop_{var}$ would prevent *any* search. However, such queries are of very limited interest. Therefore, $stop_{var}$ can be used in many settings to restrict the search while leaving many meaningful options.
- $stop_{time}(S)$: true if the search has lasted more than a given amount of time. Observe that our approach is guaranteed to have *some* recommended S_b state at any time.

Pull&push constants technique (PPC) This technique makes *educated guesses* on which selection edges to cut and which to preserve. It orders all constants from the workload, according to their number of occurrences. The more frequent the constant, the more likely it is to appear in the selected view state, because it represents a selective condition shared by many views. Thus, prior to any search, we cut *all* selection edges corresponding to constants appearing one or a few times ("pull constants" part). If this preprocessing 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 (obtained from S_0 by the l successive SCs). *After* the search has finished, however, we may be able to "push" back some of the constants cut in the "pull" stage. This is the case if, for a recommended view v, *all* rewritings using v apply the same selection on v, corresponding to a constant eagerly removed by the "pull".

PPC may compromise optimality, given that the comparisons performed during the search ignore the fact that some selections may be brought back by the post-processing.

For example, consider the following two simple queries:

$q_3(X_1, X_2)$:-$t(X_1, hasTitle, X_2), t(X_1, type, paint)$
$q_4(X_3)$:-$t(X_3, hasTitle, starryNight)$

If we apply PPC by *pull*ing all constants that appear only once in the workload, namely $paint$ and $night$, our initial state becomes:

$v_3(X_1, X_2, X_4)$:-$t(X_1, hasTitle, X_2), t(X_1, type, X_4)$
$v_4(X_3, X_5)$:- $t(X_1, hasTitle, X_5)$

Assume the best state of the search is obtained after applying a JC on v_3 (leading to two new views v_5 and v_6) and a VF between v_5 and v_4. The views of the state are:

$v_6(X_1, X_4)$:-$t(X_1, type, X_4)$, $v_7(X_6, X_7)$:-$t(X_6, hasTitle, X_7)$

The rewritings are (projections are omitted for readability):

$q_3 = \sigma_{X_4=paint}(v_7 \bowtie_{X_6=X_1} v_6)$, $q_4 = \sigma_{X_7=night}(v_7)$

In this case, removing the constant $night$ was a good choice, as it enabled a VF. However, removing $paint$ did not help and this constant can be *push*ed back, creating the view $v_8(X_1)$:-$t(X_1, type, paint)$ which will be used instead of v_6 in the rewriting of q_3. The resulting state with $paint$ pushed back has a lower cost, since v_8 occupies less space than v_6, and the rewriting of q_3 is also more efficient (a simple scan on a smaller table).

6. EXPERIMENTAL EVALUATION

This Section presents an experimental evaluation of our approach, which we have fully implemented as a Java 6 application. The application takes as input a set of conjunctive RDF queries and possibly an RDF schema, and produces as output the set of recommended views and query rewritings. It uses a database back-end to store both the original RDF data and schema, and the views.

Platform and data layout We needed a back-end of reasonable scalability to facilitate the evaluation of the rewritings proposed by our approach. We have chosen PostgreSQL (version 8.4.3), both

for its reputation as a (free) efficient platform, and because it has been used in several related works [1, 15, 16, 20, 24]. Integrating our view selection approach with another platform is easy as soon as that platform supports the evaluation of our select-project-join rewritings, and provided that the cost function is appropriately customized to account for the respective evaluation engine.

As in many previous works, for efficiency, we stored the data in a dictionary-encoded triple table, using a distinct integer for each distinct URI or literal appearing in an s, p or o value. The encoding dictionary was stored as a separate table indexed both by the integer dictionary code and by the encoded constant. The triple table was clustered by the columns p and then s, to enhance the efficiency of (frequent) queries where the p values are specified in most or all atoms. Moreover, we indexed the encoded triple table on s, p, o, and all two- and three-column combinations.

Data and queries As in previous works [1, 15, 24], we used the Barton RDF dataset and RDFS [25]. The initial dataset consists of about 50 million triples. After some cleaning (removing formatting errors, eliminating duplicates etc.) we kept about 35 million distinct triples. The space occupied by the encoded triple table, the dictionary and the indexes within PostgreSQL was 39 GB.

The Barton query workload [25] contains few queries with no commonality among them. To better test our approach, we built two query generators, producing queries of controllable size, shape, and commonality. The first one simply outputs the desired queries, and has maximum flexibility. The second takes as input not only the workload characteristics, but also a dataset (RDF + RDFS) and generates queries having non-empty answers on the given dataset. We used it to obtain interesting workloads on the Barton dataset.

Weights of cost components For VSO and REC (Section 3.4), we used c_s=1 and c_r=1. For each workload, we set the value of c_m taking into account the database size and the average number of atoms in each query, so that for the initial state S_0, $c_m \cdot VMC$ is within at most two orders of magnitude from the other two cost components, $c_s \cdot VSO$ and $c_r \cdot REC$. In most cases, this lead to c_m=0.5. Finally, we set f=2 in VMC, since this value gave the most appropriate range to VMC through the search.

Hardware and memory The PostgreSQL server ran on a separate 2.13 GHz Intel Xeon machine with 8GB RAM. We ran search algorithms on two classes of hardware: a *desktop* 8-core Intel Xeon 2.13 GHz machine with 16 GB RAM (the JVM was given 4 GB), and several *cluster machines*, each of which is a 4-core Intel Xeon 2.33 GHz with 4 GB RAM (the JVM was given 3 GB). Each experiment ran on one machine. While there are opportunities for parallelization (see Section 8), we did not exploit them in this work. All machines were running Mandriva Linux 2.6.31.

6.1 Competitor search strategies

We have implemented the three strategies, *Pruning*, *Greedy* and *Heuristic*, introduced in the relational view selection work which inspired our states and transitions [22]. All these strategies follow a divide-and-conquer approach. They start by breaking down the initial state into a set of 1-query states, and apply all possible edge removals, then all possible view breaks on each such state. Then, they seek to put back together states corresponding to the complete workload by adding up and, when appropriate, fusing, one state for each workload query. Since any combination of partial states leads to a valid state in [22], the number of states thus created explodes. To avoid it, *Pruning* discards partial states outgrowing the given space or cost budget, whereas *Greedy* develops very few states: it only keeps the best combined state, say, for the workload queries $\{q_1, q_2\}$, even though this may prevent finding the best combined state for $\{q_1, q_2, q_3\}$. Finally, *Heuristic* resembles *Pruning*, except that after having built all one-query states, it only keeps: the minimal-cost state for each query, and any states which offer some view fusing opportunity. Since our algorithms do not use a cost or space budget, we did not give one to the [22] strategies either. This does not prevent their pruning which is mostly based on comparing two states and discarding the less interesting one.

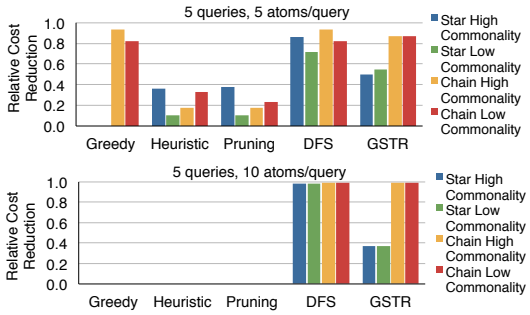


Figure 4: Strategy comparison on small workloads.

Search strategy acronyms In the sequel, for convenience, we will refer to the [22] strategies simply as *Pruning*, *Greedy* and *Heuristic*. Among the strategies we propose (see Section 5.2), DFS is the (stratified) depth-first search, while GSTR is the greedy strategy. The suffixes -AVF and -PPC after a strategy name denote aggressive view fusion, respectively, pull&push constants, applied in conjunction with that strategy. The suffix -STV denotes that the $stop_{var}$ stop condition is used, while -PPC-k, where k is an integer, denotes the pull&push constants optimizations removing all selection predicates on constants that appear less than k times in the workload.

Relative cost reduction To assess search effectiveness, we define the *relative cost reduction* (rcr) of a given strategy Σ and workload Q, at a given moment, as the ratio $(c^\epsilon(S_0) - c^\epsilon(S_b))/c^\epsilon(S_0)$, that is, the fraction of the cost of the initial state S_0, avoided by the current best state found by Σ by that moment during the search.

6.2 Comparison with existing strategies

We compare our strategies with those of [22] for two small workloads of 5 queries each. While the queries they tested involve on average 4 relations, one needs more RDF atoms than relations to express the same logical query, since data that would fit in a wide relational tuple is split over many RDF triples. Thus, queries in the first and second workload have 5 and 10 atoms each, respectively.

Figure 4 shows the rcr of the three strategies of [22] and our strategies DFS-AVF-STV and GSTR-AVF-STV. The reasons for using the specific heuristics on our strategies are explained in Section 6.3. The Figure considers workloads of star and chain queries, which are typical in RDF. In particular, star queries translate to query graphs (Definition 3.1) that are cliques (each atom is connected to all others), allowing for many VBs and JCs and, therefore, have a search space of increased size, whereas chain queries can be considered an average case regarding the difficulty of the search. The workloads were generated both with high and low commonality across queries and we used the $stop_{time}$ stop condition set to 30 minutes. While this may seem long, recall that the complexity of search is high (Section 5.1). We consider this duration acceptable as view selection is an *off-line* process. The overhead is worth it especially for large workloads, and/or queries asked repeatedly.

As can be seen in Figure 4, for the smaller workload, all strategies ran well, with DFS-AVF-STV and GSTR-AVF-STV being the best. The runs did not finish, i.e., the strategies might have found better solutions by searching longer. *Greedy* managed to reduce the cost significantly for chains but failed to find any state better than the initial one for stars queries. For the larger workload, the [22] strategies failed to produce any solution, as they outgrow the available memory building *partial* states (for $1, 2, 3$ queries etc.) be-

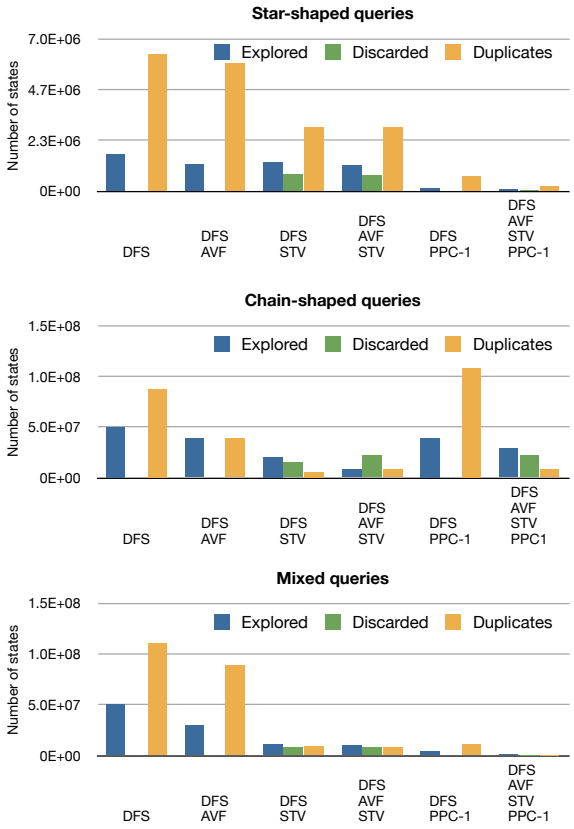


Figure 5: Impact of heuristics on the DFS strategy for star, chain and mixed query workloads.

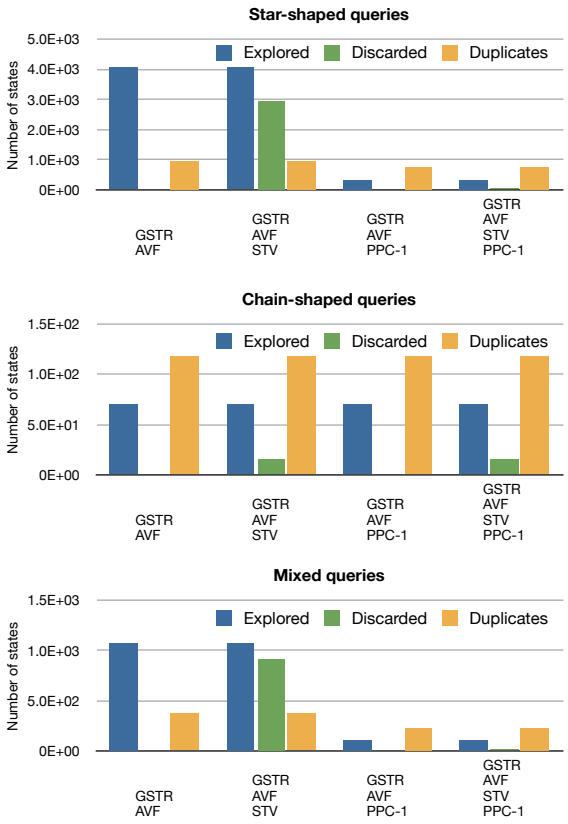


Figure 6: Impact of heuristics on the GSTR strategy for star, chain and mixed query workloads.

fore building *any* state covering all 5 queries. In contrast, DFS-AVF-STV and GSTR-AVF-STV keep running and achieve interesting cost reductions. The same trend was observed on workloads with cycle- and random graph-shaped queries (we generated both sparse and dense graphs), at high and low commonality.

Thus, from now on, and in particular for large workloads, we focus only on our strategies, since those of [22] systematically outgrow the memory before reaching a full candidate view set.

6.3 Impact of heuristics and optimizations

We now study the impact of the AVF, STV and PPC techniques on the search space explored by our algorithms. Tiny workloads of 2 queries of 4 atoms each suffices to illustrate this. We used the DFS and GSTR strategies with several combinations of heuristics and compared them with the 3 algorithms of [22]. Figure 5 shows numbers collected with the DFS strategies on star-shaped, chain-shaped and mixed query workloads. Figures 6 and 7 show results obtained with the same workloads running the GSTR strategy and the algorithms of [22] respectively. The states *created* are those reached by the search. States previously attained through a different path are recognized by searching for their presence in the ES or CS sets (Section 5), considered *duplicates* and ignored. The STV heuristic leads to *discarding* some states. Finally, *explored* states are those from which all outgoing transitions respecting the given strategy have been explored.

A first remark based on Figure 5 is that the number of duplicate states may be quite important. Duplicates occur because even when using a stratified strategy, a state may be reached by more than one path. For instance, assume for some given views v_1, v_2 that an SC modifies v_1 into v_1' (denoted $v_1 \xrightarrow{\text{SC}(c_1)} v_1'$) and simi-

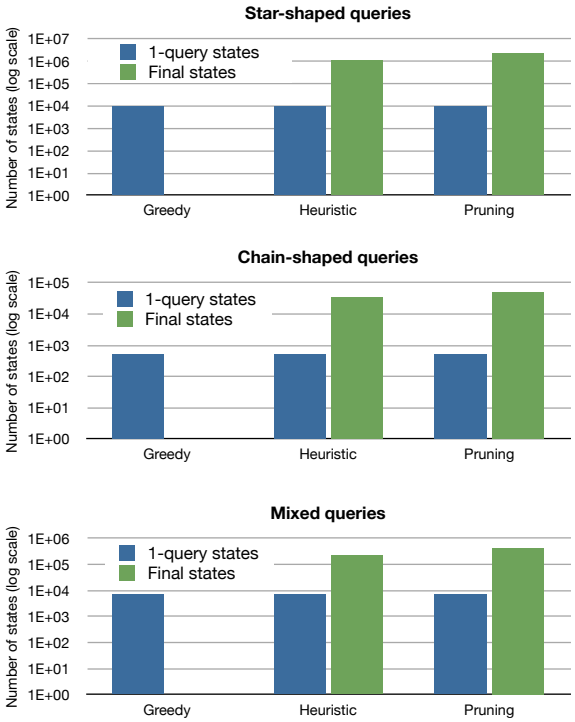


Figure 7: Impact of heuristics on the *Greedy*, *Heuristic* and *Pruning* strategies for star, chain and mixed query workloads.

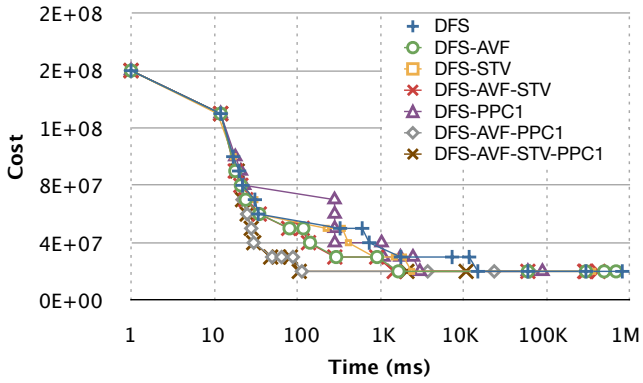


Figure 8: Cost reduction over time with an exhaustive DFS strategy and various combinations of heuristics.

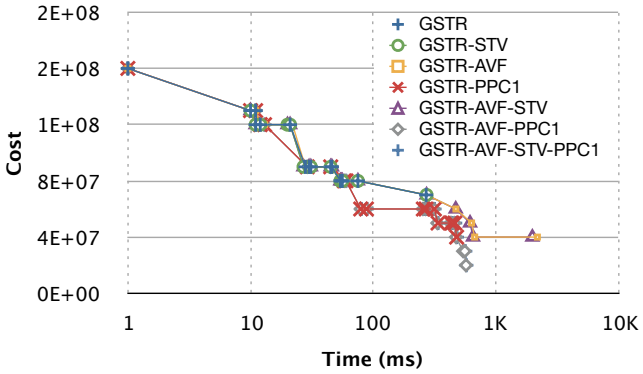


Figure 9: Cost reduction over time with GSTR strategy and various combinations of heuristics.

larly $v_2 \xrightarrow{\text{SC}(c_2)} v_2'$. From the state (v_1, v_2), our algorithms reach the state (v_1', v_2') twice: once through (v_1, v_2') and a second time through (v_1', v_2). Our algorithm identifies such states as soon as they are created, in order not to repeat their exploration.

Second, the space obtained from chain-shaped queries is two orders of magnitude larger than for star-shaped queries, although the initial number of atoms per query was identical. Chains give way to the highest possible number of VBs; each state reached through a VB on a chain query will itself contain chain queries. This explains why the search space is significantly larger for a pure chain-query workload.

AVF alone has marginal effect on the number of states explored, but STV prunes the space very efficiently regardless of the workload type. As for PPC-1, its impact is not always beneficial w.r.t. the search space size. Although it dramatically cuts down the total size of a star-shaped query workload search space, using PPC on chain-shaped queries does not significantly lower the number of duplicates reached. Chains generally contain less constants than stars, thus attempting to reduce the space size simply by removing constants has a mild effect. In this case, PPC-1 reduced the total space size only by 21%.

In Figure 6, the GSTR strategy is used in conjunction with AVF, since we observed through our experiments that AVF when used with GSTR significantly improved the cost of the best returned state (not waiting until the last stage of GSTR to perform VFs, increases the chances to find a better state). As can be seen from the Figure, GSTR overall generates much less states than DFS. The STV heuristic did not have an impact in these cases, because it seems that the states discarded from this heuristic were not among those states that GSTR was going to continue the search upon. As for PPC-1, it reduced the number of explored states for stars, but had no significant impact on the chain-query workload.

Let us now examine the search space size for the three strategies of [22] (Figure 7). Due to the differences in the nature of the searches, we cannot directly compare the number of states created by those strategies with the ones created by ours. In particular, for the strategies of [22], we measure the 1-query states and the final states. As explained in Section 6.1, all three strategies first create 1-query states (states for each of the queries of the workload) and then combine them to reach final states (that contain rewritings for all the queries). As expected, all three strategies generate the same number of 1-query states, as in this stage no pruning has been performed yet. The *Greedy* strategy has only one final state, as it only picks each time the best state (resulting from a combination of 1-query states) to continue the search. Between the *Pruning* and the *Heuristic*, we observe that the additional heuristic criterion that the latter uses, indeed contributes in generating less final states.

Optimality of search strategies We also studied the evolution of the best cost with the search time both for DFS and GSTR with various combinations of heuristics. Figures 8 and 9 show the results of this experiment. We recall that DFS and DFS-AVF are optimal and, thus, find the globally optimal state.

The experiment illustrates, first, that GSTR explores much less states and therefore is much faster (3 orders of magnitude in this example) than DFS. On this example which was purposely small, all DFS variants converged to the same globally optimal state, although this cannot be guaranteed in general. Observe, however, that the final state reached by GSTR versions is not the optimal one, reached by plain DFS. In particular, on this example the ratio between the cost of the optimal state (found by DFS) and that of the best state obtained by GSTR ranges from 0.28 to 0.99, depending on the heuristics used. Again, this ratio holds for this specific example and depends on the given workload.

Among the DFS variants, we notice that applying AVF has a consistently good impact, i.e., the cost of the best state found decreases more quickly when AVF is enabled. Hence, by using AVF, we can stop the search earlier and get a better result than the plain DFS would give at the same point. The combination of all heuristics yielded the best results in this example, although in general, PPC and STV may compromise the quality of the best state in exchange for shortening the search.

Among the GSTR variants, an interesting remark is that the PPC heuristic leads to attaining a better final state than if PPC is not used. This can be explained as follows. Applying PPC at the very beginning of the search leads to a state from which the heuristic exploration of GSTR turned out to find a better final state. However, this cannot be guaranteed in general.

In the sequel, given the above results, we will systematically use the combination AVF-STV both for DFS and GSTR, since these heuristics in most cases make the search significantly faster without increasing a lot the cost of the returned state.

6.4 Cost reduction on large workloads

We study the scalability of our DFS and GSTR algorithms for large query workloads. To this purpose, we generated workloads of 5, 10, 20, 50, 100 and 200 queries; each query has 10 atoms, i.e., the views of the initial states contains 10 atoms on average. We consider workloads consisting of: star queries only; chain queries only; random-graph shaped queries (with two variants, dense graph and sparse graph); mixed, combining queries of all the previous shapes. For each kind of workload, we generate three low- and

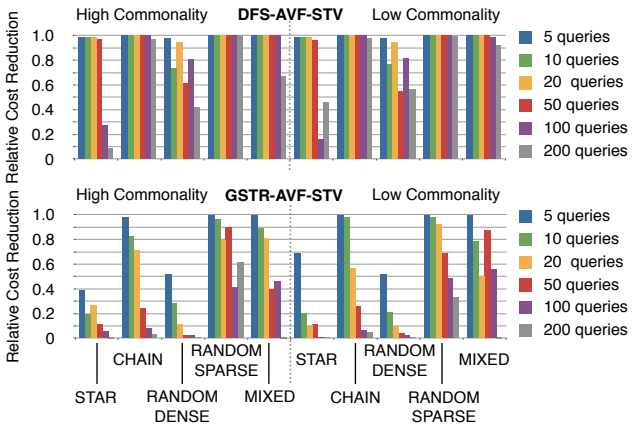


Figure 10: Relative cost reduction for large workloads.

Workload Q	$\|Q\|$	$\#a(Q)$	$\#c(Q)$	$\|Q^r\|$	$\#a(Q^r)$	$\#c(Q^r)$
Q_1	5	33	35	20	143	157
Q_2	10	76	77	231	1436	1651

Table 3: Workloads used for reformulation experiments.

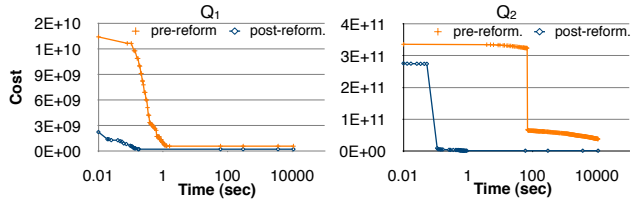


Figure 11: Search for view sets using reformulation.

three high-commonality variants. On each of these 30 workloads, we ran DFS-AVF-STV and GSTR-AVF-STV. We used the $stop_{time}$ stop condition set to 3 hours. These experiments ran in the cluster.

Figure 10 plots for each of the 10 workload types, the rcr averaged over the 3 workloads of that type, at the end of the search. A first remark is that DFS's relative cost reduction is very impressive overall, and in many cases around 0.99. Second, note that the rcr of GSTR-AVF-STV is generally smaller than that of DFS-AVF-STV, because GSTR explores significantly fewer states than DFS and might miss interesting opportunities. Third, we can distinguish "easier" workloads, such as chains and random-sparse graphs, resulting in query graphs with fewer edges and, thus, fewer transitions. For such workloads, the rcr is higher since the search space is smaller (and bigger part of it was explored). Stars and random-dense graphs are difficult cases, as they lead to many edges, thus smaller rcrs. Finally, the rcrs obtained for high-commonality workloads are generally higher than for low-commonality, e.g., for random-dense and mixed workloads. This confirms the intuition that more factorization opportunities lead to higher gains. DFS-AVF-STV resulted in views with 3.2 atoms in average, whereas GSTR-AVF-STV produced views with 6.5 atoms in average.

We conclude that DFS-AVF-STV scales well up to 200 queries, depending on the workload structural complexity, and can achieve very significant reductions in the state cost.

6.5 View selection and implicit triples

We study the impact of implicit triples on view selection performance. Starting from a non-saturated database D and workload Q, three scenarios are possible: (i) saturated database D^s, search on Q and the statistics of D^s; (ii) original database D, search on the pre-reformulated workload Q^r and the statistics of D; (iii) original database D, search on Q with the statistics of the saturated database D^s (recall from Section 4.3 that we gather them *without* actually saturating the database). Of course, we consider the same RDF entailment rules for the three scenarios, i.e., those brought by an RDFS. Saturation and post-reformulation coincide for any search algorithm, since they lead to the same input statistics and workload. Hence, we only study the search for pre- and post-reformulation.

This experiment uses the Barton dataset as well. The schema consists of 61 properties, 39 classes, 24 rdfs:subClassOf statements, 2 rdfs:subPropertyOf statements, 39 rdfs:domain statements and 41 rdfs:range statements. We generated two satisfiable workloads Q_1 and Q_2, whose properties and those of their reformulated versions Q_1^r and Q_2^r are characterized in Table 3; $|Q|$ denotes the number of queries in Q, $\#a(Q)$ the number of atoms and $\#c(Q)$ the number of constants. Q_1 is a subset of Q_2.

Figure 11 shows the evolution of the best cost found by DFS-AVF-STV for both workloads (post-reformulation) and their reformulated variants (pre-reformulation). The search was cut after 3 hours. We see that the initial state for reformulated workloads has higher cost than the original workloads. Further, the best state cost decreases rapidly with post-reformulation, because the workload is much smaller and the search space is traversed faster. In contrast, the important workload sizes slow down the cost decrease for pre-reformulation. At the end of the search, pre-reformulation's best cost found is higher than that of post-reformulation, by a factor of 2.7 for Q_1, and 22 for Q_2. This confirms our expectation that the advantages of post-reformulation are most visible for larger workloads (with larger Q^r). Moreover, the best cost is reached faster in post-reformulation.

The number of implicit triples increases with the size of the database D and of the schema $\mathcal{S}$. Indeed, given the statements allowed in an RDFS (Table 1), it is easy to see that in the worst case the number of assertions $\#t$ for a relation can yield $2.\#t$ implicit assertions in another relation. This occurs when the $\#t$ assertions are for a property whose domain and range are a same class: the assertion (u_1, p, u_2), such that $(p, \text{rdfs:domain}, c)$ and $(p, \text{rdfs:range}, c)$ are in the RDFS, yields the two implicit triples $(u_1, \text{rdf:type}, c)$ and $(u_2, \text{rdf:type}, c)$. In turn, all those implicit triples can be further propagated through subclass statements. Thus, provided that $|\mathcal{S}|$ is the number of statements in the RDFS, the number of classes in $\mathcal{S}$ is bounded by $2.|\mathcal{S}|$: every class occurs only once in the RDFS. It follows that the size of the saturation is in $O(|D|.|S|)$, where $|D|$ is the number of assertions in the database.

Similarly, $|Q^r|$ may be the same as $|Q|$, or exponentially larger (Theorem 4.1). *In a reformulation-based setting*, view selection based on post-reformulation is clearly better than based on pre-reformulation, since the initial state is better and search is faster, especially for large workloads. *Among saturation and post-reformulation*, the best choice strongly depends on the context (distribution, rights to update the database, frequency and types of updates etc.) as explained in Section 4.2. The views recommended in a saturation and a post-reformulation context are the same.

Cost of statistics collection In order to assess the additional cost brought by the statistics collection in each of the three reasoning approaches, we measured the corresponding time needed to gather the statistics for a workload of 10 queries having a total of 24 atoms on the Barton database. Overall, we found these costs to be acceptable.

- gathering the (workload-relevant) statistics on the saturated database took 59 seconds, while saturating the Barton dataset took 37 minutes.
- gathering the statistics for pre-reasoning took 147 seconds.
- post-reasoning requires estimating the size of a set of atoms: those appearing in the workload, as well as any relaxation

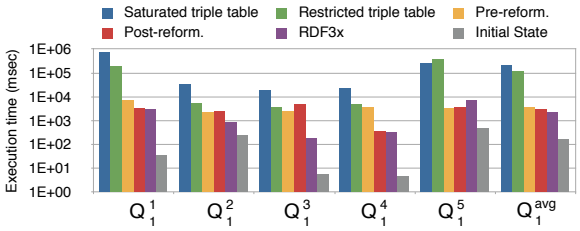


Figure 12: Execution times for queries with RDFS.

thereof (the size of these atoms is counted in all cases, as explained in Section 3.4). Recall though that in the post-reformulation approach, the size of this set of atoms must be counted *as if* the database was saturated, without actually saturating it, as explained in Section 4.3. Observe that the most relaxed atom t_0, defined by $(?X, ?Y, ?Z)$ (the size of the saturated database) needs to be counted for *any* workload, because it is a relaxation of any possible query atom. In our workload, computing the size of this "most general" term took 849 seconds, which is quite high, but observe that this only needs to be gathered *once for the whole database*, and can be subsequently cached and re-used for various workloads. Computing the size of *all other relaxed atoms* in our example took 391 seconds, which is comparable with the 147 seconds of pre-reformulation. Thus, the overhead of the statistics collection in the case of reformulation is not significant, given the benefit that this approach brings compared to pre-reformulation with respect to the time needed for the actual search.

6.6 View-based query evaluation

We now study the benefits that our recommended views actually bring to query evaluation (recall though that our view selection does not optimize for query evaluation *only*, but for a combination including storage and maintenance costs). For the workload Q_1 described in Section 6.5, we materialized the views recommended by pre- and post-reformulation, and ran the 5 queries Q_1^1 to Q_1^5 of Q_1 using (i) the views, (ii) the (dictionary-encoded, heavily indexed) *saturated* triple table in PostgreSQL, (iii) a restricted version of (ii) only with the triples needed for answering Q_1, (iv) RDF-3X [17] (loading the saturated database in it), and (v) the materialization of the query workload (initial state). RDF-3X times were put as a reference; by using PostgreSQL (even with views) we did not expect to get better times than those of the state-of-the-art RDF platform.

The views were materialized in 81 seconds for post-reformulation (the total view size was 433 MB or 15% of the database size), and 103 seconds for pre-reformulation (601 MB or 21% of the database size). Figure 12 shows that using our views, queries are evaluated more than an order of magnitude faster than on the triple table, even when using the restricted triple table (iii). Both pre- and post-reformulation performed in the range of RDF-3X. This is a promising result, since our approach can be used on top of RDF-3X and achieve an even bigger gain. Finally, as expected, materializing the queries gives the best results (simply scanning the views is sufficient).

Table 4 and 5 show detailed query evaluation times for 2 additional workloads, Q_3 and Q_4, of high and low commonality respectively, recommended by the DFS and GSTR strategies. These were run with the AVF and STV heuristics (omitted here for readability). The TT column refers to the evaluation of the workloads directly onto the triple table without any optimization other than that of the host DBMS. The first three columns were obtained from searches performed without any RDF Schema. The three last ones correspond to similar settings, but with the Barton RDF Schema

	DFS	GSTR	TT	DFS^r	GSTR^r	TT^r
Q_3^1	596	600	732	154	623	19282
Q_3^2	559	586	271	76	587	3317
Q_3^3	1798	3277	271	76	3354	95755
Q_3^4	1014	1037	23479	1537	1068	62688
Q_3^5	75	77	1772	146	71	11490

Table 4: Execution times (msec) for high commonality workload of 5 queries.

	DFS	GSTR	TT	DFS^r	GSTR^r	TT^r
Q_4^1	90	39	1635	75	121	460
Q_4^2	48	43	4525	31	64	17930
Q_4^3	3014	9210	66	3101	17034	54989
Q_4^4	11	13	367	39	59	238
Q_4^5	43	257	1451	70	158	1192

Table 5: Execution times (msec) for low commonality workload of 5 queries.

used for post-reformulation (DFS^r and GSTR^r) and for saturating the dataset (TT^r).

Obviously, evaluating the queries against the saturated version of the database TT^r, leads to a serious overhead in terms of query evaluation time. On the other hand, the view sets recommended by algorithms display no significant increase in overall evaluation times, with the exception of a single query (Q_4^3 under GSTR^r). These examples also show that view sets recommended by DFS are generally more efficient than those found by GSTR whether implicit triples are taken into account or not.

Pre-computed views are likely to speed up query evaluation in any platform, simply by avoiding computations at runtime. Moreover, our framework (i) avoids the overhead of query rewriting at run-time, as query rewritings are also pre-computed and (ii) could easily translate our rewritings directly to any RDF platform's logical plans, exploiting its physical optimization capabilities.

6.7 Influence of the cost function components

To examine how each component of the cost function (see Section 3.4) affects the search, we ran several searches using DFS-AVF-STV with multiple combinations of the cost components. Since most of the cost reduction is achieved within the first minutes of search, we set the timeout at 5 minutes. Each workload was given an average of 5 atoms per query, while workload sizes varied from 5 to 200 queries. Figure 13 shows the cost of the best state over time for typical workloads of 20 chain and mixed queries. Our online experiment page [26] hosts a complete set of results with varying weights for the cost components. Overall, we observed that a reduction of at least 1 order of magnitude is achieved within 5 minutes regardless of the input workload size or type.

We now study the impact of c_m (the weight of the view maintenance component) on the characteristics of the proposed view and, in particular, on the average number of atoms in the views. We set $c_s = 1$ and $c_r = 1$ and varied c_m from $5 \cdot 10^4$ to 0.5. The results are depicted in Figure 14, showing that relaxing c_m has a direct impact on the number of atoms in views. As expected, a strong VMC weight tends to promote smaller views of approximately 1.8 atoms on average (since the smaller the number of atoms, the smaller the maintenance cost).

6.8 Experiment conclusion

Our experiments have shown that the GSTR and DFS strategies scale well on up to 200 queries and achieve impressive cost reduction factors in many cases close to 99%. The strategies of [22] are also effective for small workloads, but they outgrow the memory on larger ones before producing a solution. The AVF and STV heuris-

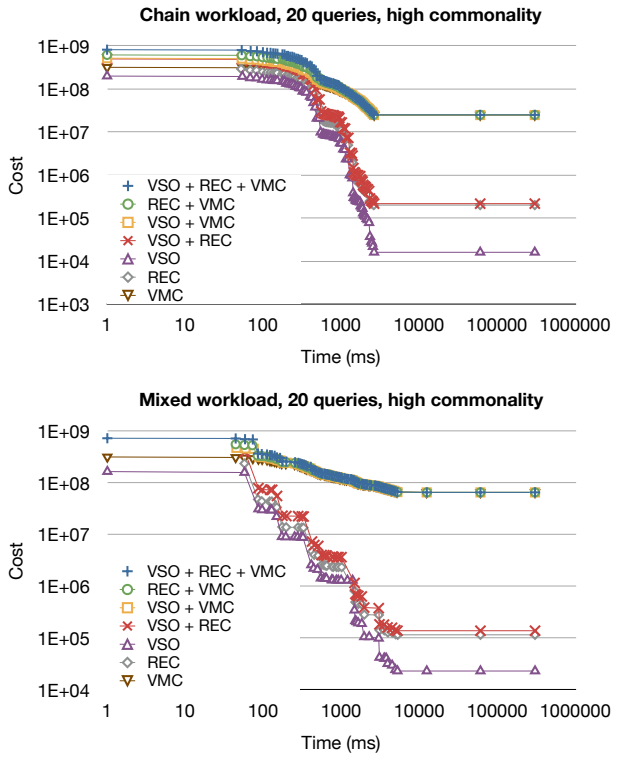


Figure 13: Cost reduction with varying cost components.

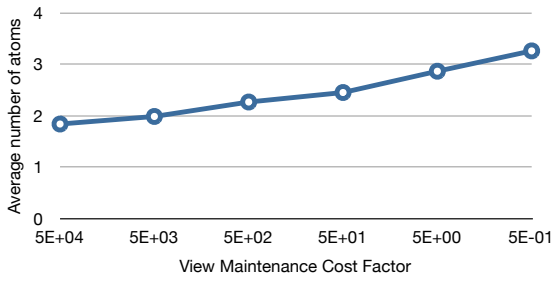


Figure 14: Average number of atoms per view in selected states.

tics are efficient and effective, i.e., they reduce the search space while preserving view set quality. Post-reformulation largely outperforms pre-reformulation in terms of speed and effectiveness of the candidate view set selection. Finally, our recommended views do reduce query evaluation times by several orders of magnitude. A digest of our experimental results can found at [26].

A tighter integration of the view selection tool with the internals of the data management platform, and/or using a dedicated RDF system, is likely to increase performance gains even more.

7. RELATED WORKS

Our work is among the first to explore materialized view selection in RDF databases. The closest works related to ours are [6] and [9]. RDFMatView [6] recommends RDF indices to materialize for a given workload, while in [9] a set of path expressions appearing in the given workload is selected to be materialized, both aiming at improving the performance of query evaluation. Unlike our approach, none of these works aims at rewriting the queries completely using the materialized indices or paths and, thus, cannot be used in scenarios where the client needs to process her queries even without access to the database. Moreover, they do not consider the implicit triples that are inherent to RDF.

Commonly used RDF management platforms (e.g., Sesame, 3store or Jena) are based on a relatively simple mapping of triples within a relational database. Many research works have addressed the efficient processing of RDF queries and updates [1, 15, 16, 17, 20, 23, 24], proposing various storage and indexing models. In vertical partitioning [1] one (s, o) relation is created for each property value (possibly leading to large unions for queries with variables in the p position). The authors of [16, 17] have built RDF-3X, a native RDF query engine. In many of the approaches, the (s, p, o) table is indexed in multiple ways (by each attribute, each pair of attributes etc.), a technique originally introduced in [24]. These techniques have been shown to result in good RDF query and update performance. We view our approach as complementary to these works, since we seek to identify materialized views to store *on top (independently) of* the base store and indexes. To adapt our approach to a specific RDF data management platform, one only needs (i) an execution framework capable of evaluating our simple select-project-join rewritings and (ii) possibly, tailoring the cost function to the particularities of the platform. Our approach improves performance by *exploiting pre-computed results and thus avoiding computations at query evaluation time*, gains likely to extend to any context.

Techniques to estimate the selectivity of RDF query patterns were proposed in [13, 21]. We compute the simple cardinalities advocated in RDF-3X [16], which are also shown to lead to satisfactory join size estimation.

The main results on query rewriting for answering queries using views are surveyed in [11]. In contrast with query rewriting algorithms, views are not part of the input of view selection, but are part of the output together with the rewritings. In particular, and following [22], our view selection algorithm generates rewritings while searching for candidate views. As for the rewritings themselves, view selection produces equivalent rewritings, as query rewriting does in the setting of query optimization, while query rewriting for data integration typically produces maximally-contained rewritings due to the incompleteness of the data sources.

Materialized view selection has been intensely studied in relational databases [8] and data warehouses [12]. We used [22] as a starting point for our work, as it is one of the prevalent works in the area and the closest to our problem definition and query language. However, in [22] the restriction that *no relation may appear twice in a workload query* is imposed, under which view equivalence can be tested in PTIME. This simplification is incompatible with RDF queries, which repeatedly use the triple table. In our context, determining view equivalence (needed for VF and for the search strategies) is NP-complete [7]. This, along with the typically bigger size of RDF queries compared to the relational ones (since only one table with three attributes is used), increase the complexity of the problem even more. Hence, the strategies presented in [22] are not effective in our context. We innovate over [22] by proposing new search strategies and heuristics, which, as demonstrated in Section 6, do not suffer from memory limitations and lead to the selection of efficient views, even if we limit the time of the search. The set of transitions used in [22] comprised: edge removal (ER′), attribute removal (AR′), view break (VB′), view merging (VM′), and attribute transfer (AT′). We modified them for our context as follows. First, AR′ and AT′ do not apply in our setting due to the differences between the SQL-like language they use and our Datalog formalism. In particular, AR′ considers that a given attribute (variable in our setting) can appear more than once in the query head, while AT′ assumes that constants may appear in the query head. Neither is supported in our Datalog formalism (we could extend it to include them, but this would not enlarge the set of can-

didate view sets). Second, in [22], join edges are removed by ER′ (which may introduce a Cartesian product) but only VB′ can split a view in two smaller ones. We allow JC, which removes join edges, to also split a view in two if its graph has become disconnected. Finally, the transition VM′ of [22] fuses views with inequality predicates, which are not needed in our RDF context.

Multi-query optimization [29] and partial view materialization [30] are also related works, although, unlike our approach, none of them aims to completely rewrite the queries using the views. In [29], common query subexpressions among the queries are recognized to be materialized. Views with disjunctions are supported, which we also plan to do as future work. In [30] views are only partially materialized and their content is adjusted as the queries change, which is another difference with our work (we consider static queries).

Query reformulation (a.k.a. unfolding) is directly related to query answering under constraints interpreted in an open-world assumption (e.g., [19]), i.e., when constraints are used as deductive rules. In particular, our query reformulation algorithm builds on those in the literature considering the so-called *Description Logic (DL) fragment* of RDF [3, 5], i.e., description logic constraints. This fragment corresponds to RDF databases without blank nodes that are made of an RDFS, called a Tbox, and a dataset made of assertions for classes and properties in the RDFS, called an Abox, i.e., well-formed triples of the form $(\mathtt{s}, \mathrm{rdf{:}type}, c)$ or $(\mathtt{s}, p, \mathtt{o})$, where c is a class and p a property of the RDFS. Lastly, the RDF entailment rules considered are only those dedicated to an RDFS (see Section 4.1). Reformulation algorithms for the DL fragment of RDF actually reformulate queries from a strictly less expressive language than our RDF queries. They only support atoms in which the class or the property is specified i.e., they do not support atoms like $t(\mathtt{s}, \mathrm{rdf{:}type}, X)$ or $t(\mathtt{s}, X, \mathtt{o})$ with X a variable, stating respectively that $\mathtt{s}$ is an instance of a class or that $\mathtt{s}$ is somehow related to $\mathtt{o}$. To overcome this, our reformulation algorithm extends the state of the art to our RDF queries, i.e., the BGP of SPARQL.

An early version of this work was demonstrated in [10].

8. CONCLUSION AND FUTURE WORK

We considered the setting of a Semantic Web database, including both explicit data encoded in RDF triples, and implicit data, derived from the RDF entailment rules [27]. Implicit data is important since correctly evaluating a query against an RDF database also requires taking it into account. In this context, we have addressed the problem of efficiently recommending a set of views to materialize, minimizing a combination of query evaluation, view storage and view maintenance costs. Starting from an existing relational approach, we have proposed new search algorithms and shown that they scale to large query workloads, for which previous search algorithms fail. Our view selection approach can be used as well with a saturated RDF database (where all implicit triples are added explicitly to the data), or with a non-saturated one (when queries need to be reformulated to reflect implicit triples). We have proposed a new algorithm for reformulating queries based on an RDF Schema, as well as a novel post-reformulation method for taking into account implicit triples in a query reformulation context. Post-reformulation can be much more efficient than naïve pre-reformulation, due to the high complexity of view search in the number of queries.

As future work, we consider parallelizing our view search algorithms by identifying workload queries that do not have many commonalities and running the search in parallel for each group. We also consider extending our query and view language, as well as adapting our approach to dynamic query workloads.

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

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

[4] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American Magazine*, 2001.

[5] D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *JAR*, 39(3), 2007.

[6] R. Castillo and U. Leser. Selecting materialized views for RDF data. In *ICWE Workshops*, 2010.

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

[8] R. Chirkova, A. Y. Halevy, and D. Suciu. A formal perspective on the view selection problem. *VLDB J.*, 11(3):216–237, 2002.

[9] V. Dritsou, P. Constantopoulos, A. Deligiannakis, and Y. Kotidis. Optimizing query shortcuts in RDF databases. In *ESWC*, 2011.

[10] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. RDFViewS: a storage tuning wizard for RDF applications. In *CIKM*, 2010.

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

[12] M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis. *Fundamentals of Data Warehouses*. Springer, 2001.

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

[14] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, 2011.

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

[16] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, 2009.

[17] T. Neumann and G. Weikum. x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *PVLDB*, 3(1), 2010.

[18] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 2003.

[19] R. Rosati. On the finite controllability of conjunctive query answering in databases under open-world assumption. *J. Comput. Syst. Sci.*, 77(3):572–594, 2011.

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

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

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

[23] O. Udrea, A. Pugliese, and V. S. Subrahmanian. GRIN: A graph based RDF index. In *AAAI*, 2007.

[24] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for Semantic Web data management. *PVLDB*, 1(1), 2008.

[25] The Barton data set. http://simile.mit.edu/wiki/Dataset:_Barton.

[26] Experimental results. Available at http://rdfvs.saclay.inria.fr/experiments/index.html, 2011.

[27] RDF. Available at www.w3.org/RDF/, 2004.

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

[29] J. Zhou, P.-Å. Larson, J. C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, 2007.

[30] J. Zhou, P.-Å. Larson, J. Goldstein, and L. Ding. Dynamic materialized views. In *ICDE*, 2007.