Optimizing Reformulation-based Query Answering in RDF

Damian Bursztyn, François Goasdoué, Ioana Manolescu, Alexandra Roatis

Optimizing Reformulation-based Query Answering in RDF

Optimisation de la réponse aux requêtes par reformulation dans les bases de données RDF

Damian Bursztyn[§,⋆] François Goasdoué[†,§]
Ioana Manolescu[§,⋆] Alexandra Roatiş[⋆,§]

[§]Inria, France [⋆]Université Paris-Sud, France [†]Université Rennes 1, France
firstname.lastname [§]@inria.fr [⋆]@lri.fr [†]@irisa.fr

Abstract

Reformulation-based query answering is a query processing technique aiming at answering queries against data, under constraints. It consists of reformulating the query based on the constraints, so that evaluating the reformulated query directly against the data (i.e., without considering any more the constraints) produces the correct answer set.

In this paper, we consider optimizing reformulation-based query answering in the setting of *ontology-based data access*, where SPARQL conjunctive queries are posed against RDF facts on which constraints expressed by an RDF Schema hold. The literature provides solutions for various fragments of RDF, aiming at computing the equivalent union of maximally-contained conjunctive queries w.r.t. the constraints. However, in general, such a union is large, thus it cannot be efficiently processed by a query engine.

Our contribution is (i) to generalize the query reformulation language so as to investigate a space of reformulated queries (instead of having a single possible choice), and then (ii) to select the reformulated query with lower estimated evaluation cost. Our experiments show that our technique enables reformulation-based query answering where the state-of-the-art approaches are simply unfeasible, while it may decrease their costs by orders of magnitude in other cases.

Keywords

RDF query answering, SPARQL, query reformulation, query optimization, heuristic algorithms

Résumé

La technique de réponse aux requêtes par reformulation vise à répondre à des requêtes sur des données sous contraintes. Elle consiste de reformuler la requête en fonction des contraintes, de sorte que l'évaluation de la requête reformulée, directement sur les données (c'est-à-dire en ne tenant plus compte des contraintes), produit l'ensemble de réponses correctes.

Dans cet article, nous considérons l'optimisation de la réponse aux requêtes par reformulation dans le cadre de l'accès aux données au travers d'ontologies, où des requêtes conjonctives SPARQL sont posées sur des faits RDF associés à des contraintes de schéma RDF. La littérature fournit des solutions pour divers fragments de RDF, visant à calculer l'union équivalente de requêtes conjonctives maximalement contenues par rapport aux contraintes. Mais, en général, une telle union est grande et ne peut être efficacement traitée par un moteur de requêtes.

Notre contribution est (i) de généraliser le langage de reformulation de requêtes afin de couvrir un espace de requêtes reformulées équivalentes (au lieu d'avoir une seule reformulation possible), puis (ii) de sélectionner la requête reformulée avec le coût d'évaluation estimée le plus bas. Nos expériences montrent que notre technique permet la réponse aux requêtes par reformulation où les approches sur l'état de l'art sont tout simplement irréalisable, tandis qu'elle peut diminuer leurs coûts de plusieurs ordres de grandeur dans les autres cas.

1. INTRODUCTION

The Resource Description Framework (RDF) [1] is a graph-based data model promoted by the W3C as the standard for Semantic Web applications. As such, it comes with an ontology language, *RDF Schema* (RDFS), that can be used to enhance the description of RDF *graphs*, i.e., RDF datasets. The W3C standard for querying RDF is SPARQL Protocol and RDF Query Language (SPARQL) [2]. The present work is placed in the setting of *efficiently querying RDF graphs in the presence of an RDFS ontology*, which also motivates our previous work [3]. We borrow from [3] the following introduction to scientific issues arising in this context, before describing the novel contribution of this work.

Answering SPARQL queries requires to handle the *implicit information* modeled in RDF graphs, through the essential RDF reasoning mechanism called *RDF entailment*. Observe that query answers are defined w.r.t. both explicit and implicit RDF information in an RDF graph, thus ignoring implicit information leads to incomplete answers [2]. There are two trends for answering SPARQL queries, both of which consists of a *reasoning* step, either on the graph or on the query, then followed by a *query evaluation* step.

Saturation and reformulation. A popular reasoning method is *graph saturation* (*closure*). This consists of precomputing (making explicit) and adding to the RDF graph all implicit information. Answering queries using saturation amounts to evaluating the queries against the saturated graph. While saturation leads to efficient query processing, it requires time to be computed, space to be stored, and must be recomputed upon updates. The alternative reasoning step is *query reformulation.* This consists in turning the query into a *reformulated query*, which, evaluated against the original graph, yields the exact answers to the original query. Since reformulation is made at query run-time, it is intrinsically robust to updates; the query reformulation process in itself is also typically very fast, since it only operates on the query, not on the data. However, reformulated queries are often syntactically more complex than the original ones, thus their evaluation may be costly.

Reformulation-based query answering has been studied for the Description Logics (DL) [4] fragment of RDF and the relational conjunctive SPARQL subset [5, 6, 7], and extensions thereof [8, 9, 10, 11], including the so far most expressive Database fragment of RDF we introduced in [3]. Most solutions aim at reformulating a query into the equivalent union of maximally-contained conjunctive queries w.r.t. the RDF Schema. However, in general, such a union is large thus cannot be efficiently processed by a query engine. In [3], we studied saturation-based and reformulation-based query answering for the database fragment, designed to work on top of any RDBMS. Our experiments have shown that the most efficient between saturation and reformulation depends on the database, the schema, and the frequency of updates (insertions or deletions) to the data and the schema.

Saturation has been quite well studied by now, and efficient algorithms exist, including incremental ones, which derive implicit data based on triples added to or removed from the data and/or schema, and using as little as possible the other triples [3, 12]. In contrast, the efficient evaluation of reformulated queries on very large data volumes is still challenging, due to their size and complexity. Most real RDF data management systems use saturation-based query answering; only a few marginally use reformulation-based query answering (see the related work in Section 6).

In this paper, we focus on *optimizing* reformulation-based query answering in RDF.

We consider the setting in which *SPARQL conjunctive queries, once reformulated, are handled for evaluation to a query evaluation engine*, which can be: a relational database management system (RDBMS), a dedicated RDF storage and query processing engine, or more generally any system capable of evaluating *selections*, *projections*, *joins* and *unions*, since these are the operations one encounters in reformulated queries. As our experiments have shown, the evaluation of reformulated queries may be very challenging even for well-established relational or native RDF processors; this is because reformulated queries may consist of unions of hundreds or thousands of terms.

Concretely, the present work uses the query reformulation algorithm introduced in [3] for the so-called database fragment of RDF, the largest such fragment for which query reformulation has been investigated. However, since (as explained above) query reformulation leads to large unions of joins in many other settings, the performance-improving techniques presented here apply more generally to any of these settings.

Contributions. The solution we bring to the problem of efficient evaluation of reformulated queries can be outlined as follows (see Figure 1):

1. We generalize the state-of-the-art query reformulation language, so as to go beyond a single possible reformulated query. This leads to a space of (equivalent) alternative reformulated queries having different query processing costs - in some cases much lower, in other cases much higher than the cost of the plain reformulated query. This space corresponds to the yellow-background box in Figure 1. We characterize the size of this space and show that it is oftentimes too large to be completely examined.
2. We define a cost model for the evaluation of our reformulated queries through a relational engine.
3. Based on this cost model, we devise an novel algorithm which selects one alternative reformulated query, namely q^{best} in Figure 1, which (i) computes the same result as the original (plain) reformulated query, and (ii) reduces significantly the query evaluation cost (or simply makes it possible when evaluating the plain reformulation fails!)
4. We implemented this algorithm and deployed it on top of the PostgreSQL RDBMS [13]. Extensive experiments with large RDF datasets confirm that our algorithm (i) makes possible the evaluation, through reformulation, of conjunctive queries which previous state-of-the-art techniques were incapable of handling, and (ii) brings performance improvements of several orders of magnitude.
5. Finally, we put our efficient reformulation-based query answering technique in perspective by comparing them against saturation-based query answering, both based on PostgreSQL and through the dedicated Semantic Web data management platform Virtuoso. Our experiments show that our technique is, overall, getting closer to the performance of saturation-based query answering.

The work is organized as follows. Section 2 introduces RDF, basic graph pattern queries, query reformulation and illustrates the performance issues raised by the evaluation of reformulated queries. Section 3 characterizes our solution search space and formalizes our problem statement. In Section 4, we present our cost model and introduce our algorithm, which we evaluate through experiments in Section 5. We discuss related work in Section 6, then we conclude.

2. PRELIMINARIES

In Section 2.1 we introduce *RDF graphs*, modeling RDF datasets. Section 2.2 presents the SPARQL conjunctive queries, a.k.a. *Basic Graph Pattern queries.* Finally, Section 2.3 recalls the reformulation-based query answering technique to optimize.

2.1 RDF Graphs

An *RDF graph* (or *graph*, in short) is a set of *triples* of the form `s p o`. A triple states that its *subject* `s` has the *property* `p`, and the value of that property is the *object* `o`. We consider only well-formed RDF triples, as per the RDF specification [1], using uniform resource identifiers (URIs), typed or un-typed literals (constants) and blank nodes (unknown URIs or literals).

Blank nodes are essential features of RDF allowing to support *unknown URI/literal tokens.* These are conceptually similar to the variables used in incomplete relational databases based on *V-tables* [14, 15].

Notations. We use `s`, `p`, `o` and $\_{:}b$ in triples as placeholders. Literals are shown as strings between quotes, e.g., "*string*". Finally, the set of values – URIs (U), blank nodes (B), literals (L) – of an RDF graph `G` is denoted `Val(G)`.

Figure 2 (top) shows how to use triples to describe resources, that is, to express class (unary relation) and property (binary relation) assertions. The RDF standard [1] provides a set of built-in classes and properties, as part of the

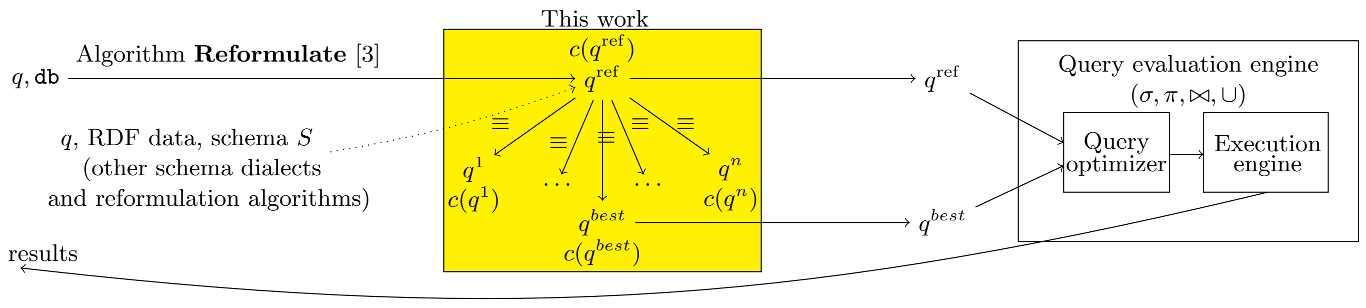


Figure 1: Outline of our approach for efficiently evaluating reformulated RDF queries.

Assertion	Triple	Relational notation
Class	s rdf:type o	$\mathtt{o}(\mathtt{s})$
Property	s p o	$\mathtt{p}(\mathtt{s}, \mathtt{o})$

Constraint	Triple	OWA interpretation
Subclass	s rdfs:subClassOf o	$\mathtt{s} \subseteq \mathtt{o}$
Subproperty	s rdfs:subPropertyOf o	$\mathtt{s} \subseteq \mathtt{o}$
Domain typing	s rdfs:domain o	$\Pi_{\text{domain}}(\mathtt{s}) \subseteq \mathtt{o}$
Range typing	s rdfs:range o	$\Pi_{\text{range}}(\mathtt{s}) \subseteq \mathtt{o}$

Figure 2: RDF (top) & RDFS (bottom) statements.

rdf: and rdfs: pre-defined namespaces. We use these namespaces exactly for these classes and properties, e.g., rdf:type specifies the class(es) to which a resource belongs.

Example 1 (**RDF graph**). *The RDF graph* G *below describes information about a book. The book is identified by the resource* doi$_1$*, and properties of its outgoing edges describe information about the author (a blank node* _:b_1 *related to a literal node for the author name), title and date of publication.*

$$\mathtt{G} = \begin{array}{l}\{\text{doi}_1 \text{ rdf:type Book},\\ \;\text{doi}_1 \text{ writtenBy } \_{:}b_1,\\ \;\text{doi}_1 \text{ hasTitle ``Game of Thrones''},\\ \;\_{:}b_1 \text{ hasName ``George R. R. Martin''},\\ \;\text{doi}_1 \text{ publishedIn ``1996''}\}\end{array}$$

RDF Schema. A valuable feature of RDF is RDF Schema (RDFS) that allows enhancing the descriptions in RDF graphs. RDFS triples declare *semantic constraints* between the classes and the properties used in those graphs.

Figure 2 (bottom) shows the allowed constraints and how to express them; *domain* and *range* denote respectively the first and second attribute of every property. The RDFS constraints (Figure 2) are interpreted under the open-world assumption (OWA) [14]. For instance, given two relations R_1, R_2, the OWA interpretation of the constraint $R_1 \subseteq R_2$ is: any tuple t in the relation R_1 *is considered as being also in the relation* R_2 (the inclusion constraint *propagates* t *to* R_2). More specifically, when working with the RDF data model, if the triples hasFriend rdfs:domain Person and Anne hasFriend Marie hold in the graph, then so does the triple Anne rdf:type Person. The latter is due to the rdfs:domain constraint in Figure 2.

RDF entailment. *Implicit triples* are an important RDF feature, considered part of the RDF graph even though they are not explicitly present in it, e.g., Anne rdf:type Person above. W3C names *RDF entailment* the mechanism through

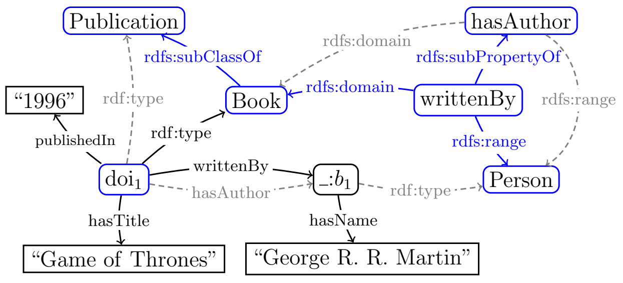


Figure 3: Sample RDF graph.

which, based on a set of explicit triples and some *entailment rules*, implicit RDF triples are derived. We denote by $\vdash^i_{\text{RDF}}$ *immediate entailment*, i.e., the process of deriving new triples through a single application of an entailment rule. More generally, a triple s p o is entailed by a graph G, denoted $\mathtt{G} \vdash_{\text{RDF}} \mathtt{s\ p\ o}$, if and only if there is a sequence of applications of immediate entailment rules that leads from G to s p o (where at each step of the entailment sequence, the triples previously entailed are also taken into account).

Example 2 (**RDFS constraints**). *Assume that the RDF graph* G *in Example 1 is extended with the following RDFS constraints.*

— *books are publications:*
Book rdfs:subClassOf Publication
— *writing something means being an author:*
writtenBy rdfs:subPropertyOf hasAuthor
— *books are written by people:*
writtenBy rdfs:domain Book
writtenBy rdfs:range Person

The resulting graph is depicted in Figure 3. Its implicit triples are those represented by dashed-line edges.

Saturation. The immediate entailment rules allow defining the finite *saturation* (a.k.a. closure) of an RDF graph G, which is the RDF graph $\mathtt{G}^\infty$ defined as the fixed-point obtained by repeatedly applying $\vdash^i_{\text{RDF}}$ rules on G.

The saturation of an RDF graph is unique (up to blank node renaming), and does not contain implicit triples (they have all been made explicit by saturation). An obvious connection holds between the triples entailed by a graph G and its saturation: $\mathtt{G} \vdash_{\text{RDF}} \mathtt{s\ p\ o}$ if and only if $\mathtt{s\ p\ o} \in \mathtt{G}^\infty$.

RDF entailment is part of the RDF standard itself; in particular, *the answers of a query posed on* G *must take into account all triples in* $\mathtt{G}^\infty$, since *the semantics of an RDF* *graph is its saturation.*

2.2 BGP Queries

We consider the well-known subset of SPARQL consisting of (unions of) *basic graph pattern* (BGP) queries, modeling the SPARQL conjunctive queries. Subject of several recent works [3, 16, 17, 18], BGP queries are the most widely used subset of SPARQL queries in real-world applications [18]. A BGP is a set of *triple patterns*, or triples/atom in short. Each triple has a subject, property and object, some of which can be variables.

Notation. In the following we use the conjunctive query notation $q(\bar{x})$:- $t_1, \ldots, t_\alpha$, where $\{t_1, \ldots, t_\alpha\}$ is a BGP; the query head variables $\bar{x}$ are called *distinguished variables*, and are a subset of the variables occurring in $t_1, \ldots, t_\alpha$; for boolean queries $\bar{x}$ is empty. The head of q is $q(\bar{x})$, and the body of q is $t_1, \ldots, t_\alpha$. We use x, y, z etc. to denote variables in queries. We denote by $\texttt{VarBl}(q)$ the set of variables *and* blank nodes occurring in the query q.

Query evaluation. Given a query q and an RDF graph $\texttt{G}$, the *evaluation of* q *against* $\texttt{G}$ is:

$$q(\texttt{G}) = \{\bar{x}_\mu \mid \mu : \texttt{VarBl}(q) \rightarrow \texttt{Val}(\texttt{G}) \text{ is a total assignment such that } t_1^\mu \in \texttt{G}, t_2^\mu \in \texttt{G}, \ldots, t_\alpha^\mu \in \texttt{G}\}$$

where we denote by t^μ the result of replacing every occurrence of a variable or blank node $e \in \texttt{VarBl}(q)$ in the triple t, by the value $\mu(e) \in \texttt{Val}(\texttt{G})$.

Note that evaluation *treats the blank nodes in a query exactly as it treats non-distinguished variables* [19]. Thus, in the sequel, without loss of generality, we consider queries where all blank nodes have been replaced by (new) distinct non-distinguished variables.

Query answering. The evaluation of q against $\texttt{G}$ uses only $\texttt{G}$'s explicit triples, thus may lead to an incomplete answer set. The (complete) *answer set* of q against $\texttt{G}$ is obtained by the evaluation of q against $\texttt{G}^\infty$, denoted by $q(\texttt{G}^\infty)$.

Example 3 (**Query answering**). *The following query asks for the names of authors of books somehow connected to the constant 1996:*

$q(x_3)$:- x_1 hasAuthor x_2, x_2 hasName x_3, x_1 x_4 "1996"

Its answer against the graph in Figure 3 is $q(\texttt{G}^\infty) = \{\langle\text{"George R. R. Martin"}\rangle\}$. *The answer results from* $\texttt{G} \vdash_{\text{RDF}}$ doi$_1$ hasAuthor _:b_1 *and the assignment* $\mu = \{x_1 \leftarrow \text{doi}_1, x_2 \leftarrow \_{:}b_1, x_3 \leftarrow \text{"George R. R. Martin"}, x_4 \leftarrow \text{publishedIn}\}$. *Observe that evaluating* q *directly against* $\texttt{G}$ *leads to the incomplete answer* $q(\texttt{G}) = \emptyset$.

2.3 Query answering against RDF databases

The *database (DB)* fragment of RDF [3] is, to the best of our knowledge, the most expressive RDF fragment for which both *saturation-* and *reformulation-based RDF query answering* has been defined and practically experimented. The fragment is thus named due to the fact that query answering against any graph from this fragment, called an *RDF database*, can be easily implemented on top of any RDBMS. This DB fragment is defined by:

— *Restricting RDF entailment* to the RDF Schema constraints only (Figure 2), a.k.a. RDFS entailment. Consequently, the DB fragment focuses on the application domain knowledge only, a.k.a. ontological knowledge, and not on the RDF meta-model knowledge which mainly begets high-level typing of subject, property, object values found in triples with abstract built-in classes, e.g., rdf:Resource, rdfs:Class, rdf:Property etc.

— *Not restricting graphs in any way.* In other words, any triple allowed by the RDF specification is also allowed in the DB fragment.

We will use $\texttt{db} = \langle \texttt{S}, \texttt{D} \rangle$ to denote a graph from the DB fragment of RDF, where the *schema* $\texttt{S}$ contains RDFS constraint triples and the *data* $\texttt{D}$ contains RDF assertion triples (Figure 2). Observe that $\texttt{S}$ and $\texttt{D}$ form a partition of any RDF graph, as a triple belongs to exactly one of them.

Saturation-based query answering amounts to precomputing the saturation of a database $\texttt{db}$ using its RDFS constraints in a forward-chaining fashion, so that the *evaluation* of every incoming query q against the saturation yields the correct answer set [3]: $q(\texttt{db}^\infty) = q(\textbf{Saturate}(\texttt{db}))$. This technique follows directly from the definitions in Section 2.1 and 2.2, and the W3C's RDF and SPARQL recommendations.

Reformulation-based query answering, in contrast, leaves the database $\texttt{db}$ untouched and reformulates every incoming query q using the RDFS constraints in a backward-chaining fashion, $\textbf{Reformulate}(q, \texttt{db}) = q^{\text{ref}}$, so that the *relational evaluation* of this reformulation against the (non-saturated) database yields the correct answer set [3]: $q(\texttt{db}^\infty) = q^{\text{ref}}(\texttt{db})$. The **Reformulate** algorithm, introduced in [17] and extended in [3], exhaustively applies a finite set of 13 reformulation rules. Starting from the incoming BGP query q to answer against $\texttt{db}$, these rules produce a *union of BGP queries* retrieving the correct answer set from the database, even if the latter is not saturated. As usual, the BGP queries in this union are those maximally-contained in q w.r.t. the constraints in $\texttt{db}$. This process is exemplified below.

Example 4 (**Query reformulation**). *The reformulation of the query* $q(x, y)$:- x rdf:type y *w.r.t. the database* $\texttt{db}$ *(obtained from the RDF graph* $\texttt{G}$ *from Figure 3), asking for all resources and the classes to which they belong, is:*

(0) $q(x, y)$:- x rdf:type y $\cup$
(1) $q(x, \text{Book})$:- x rdf:type Book $\cup$
(2) $q(x, \text{Book})$:- x writtenBy z $\cup$
(3) $q(x, \text{Book})$:- x hasAuthor z $\cup$
(4) $q(x, \text{Publication})$:- x rdf:type Publication $\cup$
(5) $q(x, \text{Publication})$:- x rdf:type Book $\cup$
(6) $q(x, \text{Publication})$:- x writtenBy z $\cup$
(7) $q(x, \text{Publication})$:- x hasAuthor z $\cup$
(8) $q(x, \text{Person})$:- x rdf:type Person $\cup$
(9) $q(x, \text{Person})$:- z writtenBy x $\cup$
(10) $q(x, \text{Person})$:- z hasAuthor x

The disjuncts (1), (4) *and* (8) *result from* (0) *by instantiating the variable* y *with classes from* $\texttt{db}$, *namely* $\{\text{Book}, \text{Publication}, \text{Person}\}$. *Item* (5) *results from* (4) *by using the subclass constraint between books and publications.* (2), (6) *and* (9) *result from their direct predecessors in the list, by making use of the domain and range constraints. Finally,* (3), (7) *and* (10) *result from their direct predecessors in the list by applying the information given by the sub-property constraint in the database.*

Evaluating this reformulation against $\texttt{db}$ *returns the same answer as* $q(\texttt{G}^\infty)$, *i.e., the answer set of* q.

3. PROBLEM STATEMENT

This work focuses on identifying techniques for *efficient reformulation-based query answering.* We first introduce the performance issues raised by the evaluation of state-of-the-art reformulated queries using the following motivating examples. We then formalize the query optimization problem we address.

Motivating Example 1. *Consider the six atoms query q_1 shown below, corresponding to Query 9 of the LUBM benchmark [20]. This query asks for tuples made of a student, a professor and a course, such that this student takes this course of this advising professor.*

$$\begin{array}{lll} q_1(x,y,z) \text{ :- } & x \text{ rdf:type } ub{:}Student, & (1) \\ & y \text{ rdf:type } ub{:}Faculty, & (2) \\ & z \text{ rdf:type } ub{:}Course & (3) \\ & x\ ub{:}advisor\ y, & (4) \\ & y\ ub{:}teacherOf\ z, & (5) \\ & x\ ub{:}takesCourse\ z & (6) \end{array}$$

The next table gives some intuitions about the difficulty of answering q_1 over a 100-million triples LUBM dataset:

Triple	*#answers*	*#reformulations*	*#answers after reformulation*
(1)	0	2	5,537,082
(2)	0	13	4,884,254
(3)	755,582	5	3,412,304
(4)	2,870,164	1	2,870,164
(5)	1,510,695	1	1,510,695
(6)	20,139,138	1	20,139,138

The saturation-based approach needs to evaluate the join of the six atoms corresponding to relations whose sizes range from a 1.5 *million to a* 20 *million triples. The evaluation of q_1, delegated to a standard RDBMS installation (described in Section 5, where q_1 is labeled Q_{05}) takes* ***56 seconds****.*

In contrast, reformulation-based query answering needs to evaluate a reformulated query q_1', which is a union of 130 *conjunctive queries, each of which consists of six atoms (one for the reformulation of each atom in the original q_1). Observe that in q_1', many sub-expressions are repeated; for instance, the three-way join over the single atoms resulting from the reformulation of atoms* 4*,* 5 *and* 6 *will appear in all of the* 130 *queries. Evaluating q_1' on the LUBM* 100 *million triples dataset takes more than* ***178 seconds****, in the same experimental setting.*

Alternatively, one could consider the equivalent query $q_1'' = (t_1)^{ref} \bowtie (t_2)^{ref} \bowtie (t_3)^{ref} \bowtie (t_4)^{ref} \bowtie t_5^{ref} \bowtie (t_6)^{ref}$, where $(.)^{ref}$ stands for the reformulation of the subquery obtained from the set of triples in parenthesis. In other terms, q_1'' reformulates each atom first (into, respectively, unions of 2, 15, 5 *respectively* 1, 1 *and* 1 *atom), and then joins them. This avoids the repeated work, yet it still requires about* ***45 seconds*** *to evaluate.*

Let us now consider the following equivalent query $q_1''' = (t_1,t_4)^{ref} \bowtie (t_2)^{ref} \bowtie (t_3,t_5)^{ref} \bowtie (t_6)^{ref}$ where $t_1,\dots,t_6$ are the triple atoms of the query q_1. Evaluating q_1'' in the same experimental setting takes just ***27 seconds****, more than* ***six times faster*** *than the initial reformulation*[1]*.*

The performance improvement of q_1''' over q_1'' is due to the intelligent grouping of the atoms t_1 and t_4 together, and similarly to the grouping of t_3 with t_5. Such groups of atoms help reduce the cardinality of the respective reformulated queries. Thus, $(t_1,t_4)^{ref}$ has 1,107,545 *answers, while $(t_3,t_5)^{ref}$ has* 1,510,695 *answers. A comparison with the cardinalities of $(t_1)^{ref}$, $(t_3)^{ref}$, $(t_4)^{ref}$ and $(t_5)^{ref}$, shown in the table at the beginning of our example, helps see the reduction in the number of tuples to be joined.*

Motivating Example 2. *Consider now the four atoms query q_2 (labeled Q_{04} in Section 5) asking for resources, their type, and organizations they are members of, such that the resource holds a doctoral degree from a university.*

$$\begin{array}{lll} q_2(x,y,z) \text{ :- } & x \text{ rdf:type } y, & (1) \\ & u \text{ rdf:type } ub{:}University, & (2) \\ & x\ ub{:}doctoralDegreeFrom\ u, & (3) \\ & x\ ub{:}memberOf\ z & (4) \end{array}$$

We detail again the query atom information, when evaluated over a LUBM dataset of 100 *million triples:*

Triple	*#answers*	*#reformulations*	*#answers after reformulation*
(1)	18,999,082	188	33,328,108
(2)	2,928,828	6	6,201,632
(3)	503,395	1	503,395
(4)	7,299,701	3	7,817,083

In this case the saturation-based query answering time is ***561 seconds****, while the reformulation leads to a query q_2' corresponding to a union of* 3,384 *four atoms queries. Due to the large number of queries in the reformulation and their considerable number of results, q_2'* ***could not be evaluated*** *in the same experimental setting*[2]*.*

Evaluating the equivalent reformulated query $q_2'' = (t_2)^{ref} \bowtie (t_1,t_3)^{ref} \bowtie (t_4)^{ref}$, where $t_1,\dots,t_4$ are the triple atoms of q_2, in the same setting returns the answer in only ***58 seconds****. As in the previous example, q_2'' gains over q' by first, reducing repeated work, and second, intelligently grouping atoms so that the query corresponding to each atom group can be efficiently evaluated and returns a result of manageable size. In particular, the biggest-size atom (1) has been grouped with the third one, and thus the number of results of $(t_1,t_3)^{ref}$ is* 2,432,964*. Since the query $(t_1,t_3)^{ref}$ is evaluated first, one then only needs to join its result, instead of manipulating the individual atoms $(t_1)^{ref}$ and t_3^{ref} (especially the former, with its* 33,328,108 *results).*

Finally, consider the query $q_2''' = (t_2)^{ref} \bowtie (t_1,t_3)^{ref} \bowtie (t_3,t_4)^{ref}$, also equivalent to q_2'. Evaluating q_2''' takes ***49 seconds****. This performance improvement w.r.t. q_2'' is achieved by making the subqueries more selective: $(t_2)^{ref}$ and $(t_1,t_3)^{ref}$ are the same as for q_2'', but $(t_3,t_4)^{ref}$ has only* 503,395 *answers. Similarly, this makes it easier to "handle" the atom t_4 (recall that t_4^{ref} has* 7,817,083 *results).*

1. Incidentally, in this example, this is also twice as fast as the saturation-based approach, without requiring the effort to saturate the database!

2. Concretely, an I/O exception was thrown by the DBMS, in connection with a failed attempt to materialize an intermediary result. While it may be possible to tune some parameters to make the evaluation of such queries possible, the same error was raised by many large-reformulation queries, a signal that their peculiar shape is problematic.

As the above examples show, generalizing the state-of-the-art query reformulation language of unions of BGP queries to that of *join* of unions of BGP queries offers a great potential for query processing optimizations. We introduce:

Definition 3.1 (JUCQ). *A* Join of Unions of Conjunctive Queries (JUCQ) *is defined as follows:*
— *any conjunctive query (*CQ*) is a* JUCQ*;*
— *any union of* CQ*s (*UCQ*) is a* JUCQ*;*
— *any join of* UCQ*s is* JUCQ.

In this work, we address the challenge of finding the best-performing JUCQ *reformulation* of a BGP query against an RDF database, among those that can be derived from a *query cover.* We introduce:

Definition 3.2 (JUCQ reformulation). *A* JUCQ reformulation $q^{\texttt{JUCQ}}$ *of a BGP query* q *w.r.t. a database* $\texttt{db}_1$ *is a* JUCQ *such that* $q^{\texttt{JUCQ}}(\texttt{db}_2) = q(\texttt{db}_2^{\infty})$*, for any RDF database* $\texttt{db}_2$ *having the same schema as* $\texttt{db}_1$.

Recall that two RDF databases have the same schema iff their saturations have the same RDFS statements.

BGP query fragmentation is a technique we introduce for exploring a space of JUCQ reformulations of a given query. The idea is to *cover* a query q with (possibly overlapping) subqueries, so as to produce a JUCQ reformulation of q by joining the (state-of-the-art) UCQ reformulations of these subqueries, obtained through any reformulation algorithm in the literature (e.g., [3]). Formally:

Definition 3.3 (BGP query cover). *A* cover *of a BGP query* $q(\bar{x})$:- $t_1, \ldots, t_n$ *is a set* $F = \{f_1, \ldots, f_m\}$ *of non-empty subsets of* q*'s triples, called* fragments*, such that* $\bigcup_{i=1}^{m} f_i = \{t_1, \ldots, t_n\}$*, no fragment is included into another, i.e.,* $f_i \not\subseteq f_j$ *for* $1 \leq i, j \leq m$ *and* $i \neq j$*, and if* F *consists of more than 1 fragment then any fragment joins at least with another, i.e., they share a variable.*

Definition 3.4 (Cover queries of a BGP query). *Let* $q(\bar{x})$:- $t_1, \ldots, t_n$ *be a BGP query and* $F = \{f_1, \ldots, f_m\}$ *one of its cover. A* cover query $q_{|i, 1\leq i\leq m}$ *of* q *w.r.t.* F *is the subquery whose body consists of the triples in* f_i *and whose head variables are the distinguished variables* $\bar{x}$ *of* q *appearing in the triples of* f_i*, plus the variables appearing in a triple of* f_i *that are shared with some triple of another fragment* $f_{j, 1\leq j\leq m, j\neq i}$*, i.e., on which the two fragments join.*

The theorem below states that evaluating a query q as the join of the cover queries resulting from one of its covers, yields the answer set of q:

Theorem 3.1 (Cover-based reformulation). *Let* $q(\bar{x})$:- $t_1, \ldots, t_n$ *be a BGP query and* $F = \{f_1, \ldots, f_m\}$ *be any of its covers,*

$$q^{\texttt{JUCQ}}(\bar{x}) \text{:- } q^{\texttt{UCQ}}_{|f_1} \bowtie \cdots \bowtie q^{\texttt{UCQ}}_{|f_m}$$

is a JUCQ *reformulation of* q *w.r.t. any database* db*, where every* $q^{\texttt{UCQ}}_{|f_i}$ *is a* UCQ *reformulation of the cover query* $q_{|f_i}$*, for* $1 \leq i \leq m$.

An *upper bound* on the size of the JUCQ cover-based reformulation space for a given query of n triples is given by the number of *minimal covers* of a set $\mathcal{S}$ of n elements [21], i.e., a set of non-empty subsets of $\mathcal{S}$ whose union is $\mathcal{S}$, and whose union of all these subsets but one is not $\mathcal{S}$. It is worth noting that this bound grows extremely rapidly as the number n of triples in a query's body increases, e.g., 1 for $n = 1$, 2 for $n = 2$, 8 for $n = 3$, 49 for $n = 4$, 462 for $n = 5$, 6424 for $n = 6$ (`http://oeis.org/A046165`). In practice, however, we require each fragment to share a variable with another (if any). Therefore, the number of JUCQ cover-based reformulations is smaller than the number of minimal covers of its set of atoms.

In order to select the best performing JUCQ reformulation within the above space, we assume given a *cost function* c which, for a JUCQ q, returns the cost $c(q(\texttt{db}))$ of evaluating it through an RDBMS storing the database db. Function c may reflect any (combination of) query evaluation costs, such as I/O, CPU etc. As customary, we rely on a *cost estimation* function c^{ϵ}, which statically provides an approximate value of c. For simplicity, in the sequel we will use c to denote the estimated cost.

The problem we study can now be stated as follows:

Definition 3.5 (Optimization problem). *Let* db *be a RDF database and* q *be a BGP query against it. The* optimization problem *we consider is to find a* JUCQ *reformulation* $q^{\texttt{JUCQ}}$ *of* q *w.r.t.* db*, based on a cover of* q*, which has the lowest cost among all the cover-based* JUCQ *reformulations of* q.

Optimized queries vs. optimized plans. As stated above and illustrated in Figure 1, the problem we consider is finding the best *query* that is the optimized reformulation of q against db, and take advantage of existing query evaluation engines for *optimizing and executing* the query. Alternatively, one could have placed this study *within an evaluation engine* and investigate *optimized plans.* We comment more on the reasons for prefering our approach in Section 6.

4. EFFICIENT QUERY ANSWERING

We present now the ingredients for setting up our cost-based query answering technique. We introduce, in Section 4.1, our cost model for JUCQ evaluation through an RDBMS. We then provide, in Section 4.2, an anytime algorithm that outputs the best cover of a BGP query found so far, i.e., whose corresponding cover-based reformulation has the lowest cost found so far. This cover is then used to evaluate the query as stated by Theorem 3.1.

4.1 Cost model

In this section we detail the cost of evaluating a JUCQ sent to the RDBMS. Recall that a JUCQ, denoted $q^{\texttt{JUCQ}}$, issued from a cover $F = \{f_1, f_2, \ldots, f_m\}$ of a BGP query $q(\bar{x})$:- $t_1, t_2, \ldots, t_n$, is the join of the set of UCQ sub-queries $F(q) = \{q_1^{\texttt{UCQ}}, q_2^{\texttt{UCQ}}, \ldots, q_m^{\texttt{UCQ}}\}$ built from the aforementioned fragments: $q_i^{\texttt{UCQ}} = q_{|f_i}$.

The evaluation cost of $q^{\texttt{JUCQ}}$ is:

$$c(q^{\texttt{JUCQ}}) = c_{\texttt{db}} + c_{unique}(q^{\texttt{JUCQ}}) + \sum_{q_i^{\texttt{UCQ}} \in F(q)} c_{eval}(q_i^{\texttt{UCQ}}) + c_{join}(q^{\texttt{UCQ}}_{i, 1\leq i\leq m}) + c_{mat}(q^{\texttt{UCQ}}_{i, 1\leq i\leq m, i\neq k}) \quad (1)$$

reflecting:

(*i*) the fixed overhead of connecting to the RDBMS $c_{\texttt{db}}$;

(*ii*) the cost of eliminating duplicate rows from the result;

(*iii*) the cost to *evaluate* each of its UCQ sub-queries $q_i^{\texttt{UCQ}} \in F(q)$;

(*iv*) the cost of eliminating duplicate rows from each of its UCQ sub-queries results;

(*v*) the cost to *join* these sub-query results; and

(*vi*) the *materialization* costs: the SQL query corresponding to a JUCQ may have many sub-queries. At execution time, some of these subqueries will have their results materialized (i.e., stored in memory or on disk) while at most one sub-query will be executed in pipeline mode. We assume without loss of generality, that the largest-result sub-query, denoted $q_k^{\texttt{UCQ}}$, is the one pipelined (this assumption has been validated by our experiments so far).

Notations. For a given query q over a database db, we denote by $|q|_t$ the estimated number of tuples in q's answer set. Recall that $q_{|\{t_i\}}$ stands for the restriction of q to its i-th atom. Using the notations above, the number of tuples in the answer set of $q_{|\{t_i\}}$ is denoted $|q_{|\{t_i\}}|_t$.

Elimination of duplicate rows. RDBMS implements several techniques to eliminate duplicate rows from the result, each suitable for a distinct scenario. The choice among them is based on the number of results (including duplicates) and configuration parameters (such as allocable memory). When hash aggregation is suitable we estimate the cost of eliminating duplicate rows from $q^{\texttt{JUCQ}}$ (and $q^{\texttt{UCQ}}$ as a particular case) result as:

$$c_{unique}(q^{\texttt{JUCQ}}) = c_l \times |q^{\texttt{JUCQ}}|_t$$

where c_l is the CPU and I/O effort involved in sorting the results.

When the results are large enough that disk merge sort is needed, we estimate the cost of eliminating duplicate rows from $q^{\texttt{JUCQ}}$ (and $q^{\texttt{UCQ}}$ as a particular case) result as:

$$c_{unique}(q^{\texttt{JUCQ}}) = c_k \times |q^{\texttt{JUCQ}}|_t \times \log |q^{\texttt{JUCQ}}|_t$$

where c_k is the CPU and I/O effort involved in (disk-based) sorting the results.

UCQ evaluation cost. We estimate the cost of evaluating a UCQ based the cost of evaluating all the CQs in this union:

$$c_{eval}(q_i^{\texttt{UCQ}}) = c_{unique}(q_i^{\texttt{UCQ}}) + \sum_{q^{\texttt{CQ}} \in q_i^{\texttt{UCQ}}} c_{eval}(q^{\texttt{CQ}})$$

The cost of evaluating *one* conjunctive query $c_{eval}(q^{\texttt{CQ}})$, where $q^{\texttt{CQ}}(\bar{x})$:- $t_1, t_2, \ldots, t_n$, through the RDBMS is made of the *scan* cost for retrieving the tuples for each of its atoms, and the cost of *joining* these tuples:

$$c_{eval}(q^{\texttt{CQ}}) = c_{scan}(q^{\texttt{CQ}}) + c_{join}(q^{\texttt{CQ}})$$

We estimate the *scan cost* of $q^{\texttt{CQ}}$ to:

$$c_{scan}(q^{\texttt{CQ}}) = c_t \times \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t$$

where c_t is the fixed cost of retrieving one tuple.

The *join* cost of $q^{\texttt{CQ}}$ represents the CPU and I/O effort involved in making the comparisons for the joins; assuming efficient join algorithms such as hash- or merge-based etc. are available, this cost is *linear in the total size of its inputs*:

$$c_{join}(q^{\texttt{CQ}}) = c_j \times \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t$$

Therefore, we have:

$$c_{eval}(q_i^{\texttt{UCQ}}) = (c_t + c_j) \times \sum_{q^{\texttt{CQ}} \in q_i^{\texttt{UCQ}}} \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t \qquad (2)$$

UCQ join cost. As before, we consider the join cost to be linear in the total size of its inputs:

$$c_{join}(q^{\texttt{UCQ}}_{i,1\leq i\leq m}) = c_j \times \sum_{q_i^{\texttt{UCQ}} \in F(q)} \sum_{q^{\texttt{CQ}} \in q_i^{\texttt{UCQ}}} \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t \qquad (3)$$

UCQ materialization cost. Finally, we consider the materialization cost associated to a query q is $c_m \times |q|_t$ for some constant c_m:

$$c_{mat}(q^{\texttt{UCQ}}_{i,1\leq i\leq m, i\neq k}) = c_m \times \sum_{q_i^{\texttt{UCQ}} \in F(q), i\neq k} \sum_{q^{\texttt{CQ}} \in q_i^{\texttt{UCQ}}} \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t \qquad (4)$$

where $q_k^{\texttt{UCQ}}$ is the largest-result sub-query, and the one which is picked for pipelining (and thus not materialized).

Injecting the equations 2, 3 and 4 into the global cost formula 1 leads to the estimated cost of a given JUCQ. This formula relies on estimated cardinalities of various subqueries of the JUCQ, as well as on the system-dependent constants $c_{\texttt{db}}$, c_{scan}, c_{join} and c_{mat} which can be determined by running a set of simple calibration queries on the RDBMS being used. The details are straightforward and we omit them here.

4.2 Anytime cover algorithm (GCov)

We now describe a search algorithm (GCov) which explores *covers* of the query, that is, decompositions into sets of atoms which may overlap. Intuitively, the algorithm attempts to identify fragments such that each fragment once reformulated can be efficiently evaluated by the underlying database engine. The key to doing so is to include highly selective, few-reformulations atoms in several fragments in order to speed up their evaluation.

Algorithm 1 (GCov) is based on this observation. GCov starts with a simple cover consisting of one atom fragments and explores possible *moves* starting from this state. A move consists of adding an atom to one fragment such that the estimated cost of the larger fragment thus obtained is smaller than the original fragment cost. Possible moves based on the initial cover are developed and added to a list, which is sorted in the decreasing order of the cost reduction they bring. Next the algorithm starts the exploration of possible moves. It picks the most promising one from the queue and applies it, leading to a new fragmentation F'. If the estimated cost of this fragmentation is smaller than the best (least) cost encountered so far, the best solution is updated to reflect this F'. Thus, the algorithm is anytime, meaning that at any point during the search the best cover found so far is known. The algorithm explores covers in breadth-first fashion adding to the queue possible moves starting from the current cover.

As described in Algorithm 1, GCov is greedy, i.e., it makes the move that produce the maximum cost reduction at each step, and avoids moves that do not decrease the current cost (and their derived moves). In practice, one could easily change the stop condition, for instance to return the best found cover as soon as its cost has diminished by a certain ratio, or after time-out period has elapsed, etc.

Algorithm 1: Anytime cover algorithm (**GCov**)

```
Input : BGP query $q(\bar{x}$:- $t_1, \ldots, t_n)$, database db $= \langle S, D \rangle$
Output: Fragmentation $F$ for the BGP query $q$
1  $F_0 \leftarrow F = \{\{t_1\}, \{t_2\}, \ldots, \{t_n\}\}$;
2  $T \leftarrow F = \{t_1, t_2, \ldots, t_n\}$;
3  $F_{best} \leftarrow F_0$;
4  $moves \leftarrow \emptyset$;
5  foreach $f \in F_0, t \in T$ s.t. $t \notin f$ do
6      $r_{F_0,f,t} \leftarrow$ estimated cost reduction obtained by adding $t$ to the fragment $f$ of $F_0$
7      if $r_{F_0,f,t} > 0$ then
8          $moves \leftarrow moves \cup (F_0, f, t, r_{F_0,f,t})$;
9  $visited \leftarrow \emptyset$;
10 while $moves \neq \emptyset$ do
11     $(F, f, t, r) \leftarrow moves.head()$;
12     $F' \leftarrow F.add(f, t)$;
13     $visited \leftarrow visited \cup F'$;
14     if $F'$ has a lower estimated cost than $F_{best}$ then
15         $F_{best} \leftarrow F'$;
16     foreach $f \in F', t \in T$ s.t. $t \notin f$ do
17         $r_{F',f,t} \leftarrow$ estimated cost reduction obtained by adding $t$ to the fragment $f$ of $F'$
18         if $r_{F',f,t} > 0$ AND $F'.add(f,t) \notin visited$ then
19             $moves \leftarrow moves \cup (F', f, t, r_{F',f,t})$;
20 return $F_{best}$;
```

5. EXPERIMENTAL EVALUATION

This section presents an experimental assessment of our approach. Section 5.1 describes the experimental settings. Section 5.2 studies the efficiency and effectiveness of our optimized query reformulation algorithm GCov. Section 5.3 compares them with the alternative saturation-based query answering method previously introduced in [3]. Finally, Section 5.4 widens our analysis to include RDF query answering through a native RDF data management engine.

5.1 Settings

Data. For our experimental evaluation we use data from the Lehigh University Benchmark (**LUBM**, in short) [20], which has been extensively used in RDF data management works. We report on experiments conducted using **1** and **100** millions triples, respectively.

Queries. We used a set of 30 BGP queries for our evaluation; the queries appear in Tables 3, 4 and 5 [3], while the main query characteristics (their number of union terms in their plain reformulation, denoted $|q^{ref}|$, as well as the number of query results on 1M and 100M triples, are shown in Table 1. The queries have between 2 and 9 atoms, with an average of 4.66 atoms. Some of the queries are part of the LUBM benchmark itself; we designed the others so that (*i*) they are plausible, i.e., they have an intuitive meaning, (*ii*) they exhibit a variety of result cardinalities and (*iii*) their reformulations are syntactically complex, in order for these queries to allow a study of the performance issues involved.

In all our tests, schema-level triples are kept in memory, while instance-level triples are stored in a `Triples(s, p, o)` table, indexed by all permutations of the (s, p, o) columns, leading a total of 6 indexes. Our indexing choice is inspired by [22, 23], to give the RDBMS efficient query evaluation opportunities.

Software. All our algorithms are fully implemented in Java$^{\text{TM}}$6 [24] and we deployed them on top of PostgreSQL [13], version 9.3 (shared_buffers = 2Gb; work_mem = 4Gb; effective_cache_size = 6Gb) as the database back-end. We chose Postgres since it is a (free) efficient platform, used in several works [9, 17, 23]. All measured times are averaged over 5 executions, since no major variations were detected between the different executions.

As in [3, 17, 22, 23], for efficiency we stored the data in a dictionary-encoded `Triples(s, p, o)` table, using a unique integer for each distinct value (URIs and literals) in the `s`, `p` and `o` positions of the dataset. The encoded `Triples` table is indexed by all the possible combinations of the three columns (i.e., a total of six indexes). Moreover, the encoding dictionary is stored as a separate table, indexed both by the code and by the encoded value (URI or literal).

Before each experiment, the VACUUM ANALYZE command is run. Moreover, before measuring each query we warm up the database cache by executing the query once. Queries whose evaluation requires more than 3 hours were interrupted, and pointed out when describing the experiments.

Hardware. The PostgreSQL [13] server ran on a 8-core Intel Xeon (E5506) 2.13 GHz machine with 16GB RAM, running Mandriva Linux release 2010.0 (Official). The same machine was used for the experiments on a Virtuoso [25] server, described in Section 5.4.

5.2 Optimized reformulation

In this section, we study well the fragmentation algorithm GCov (described in Section 4.2) performs in recommending a fragmentation that leads to a fast evaluation of the reformulated query.

For these experiments, we left the GCov algorithm to run up to completion *or* when reaching a 1 minute timeout. In practice, most runs stopped earlier than that, with the shortest taking 5 ms while the longest took around 10 seconds. This is obviously still quite high, especially since this time is counted as part of optimizing the query. We are aware of at least one source of inefficiency, namely repeated calls to our Reformulate algorithm and to the cost estimation module, where we have identified opportunities to reuse some results to save time. Work to make the search more efficient is thus ongoing.

3. For readability and without loss of information, the URIs starting with "http://www.lehigh.edu" were slightly shortened by eliminating a few /-separated steps

q	Q_{01}	Q_{02}	Q_{03}	Q_{04}	Q_{05}	Q_{06}	Q_{07}	Q_{08}	Q_{09}	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_{15}
$\|q^{ref}\|$	136	136	34	3384	130	1220	156	123	492	8496	221	221	1105	26	376
$\|q(\texttt{db})\|$ (1M)	123	123	41	26048	982	5537	0	719	269	0	47268	1530	88	4041	20205
$\|q(\texttt{db})\|$ (100M)	123	123	41	2432964	92026	523319	0	719	269	0	4409039	142337	7773	376792	1883960

q	Q_{16}	Q_{17}	Q_{18}	Q_{19}	Q_{20}	Q_{21}	Q_{22}	Q_{23}	Q_{23}	Q_{25}	Q_{26}	Q_{27}	Q_{28}	Q_{29}	Q_{30}
$\|q^{ref}\|$	28458	650	65	940	2444	697	2788	697	65	752	52	156	2256	156	318096
$\|q(\texttt{db})\|$ (1M)	0	5364	5388	47348	60342	107610	228086	60342	16134	100	12	19	5	1	0
$\|q(\texttt{db})\|$ (100M)	0	501063	503395	4425553	5632454	10041493	21289440	5632454	1510695	11820	1508	1463	5	1	495

Table 1: Characteristics of the queries used in our study.

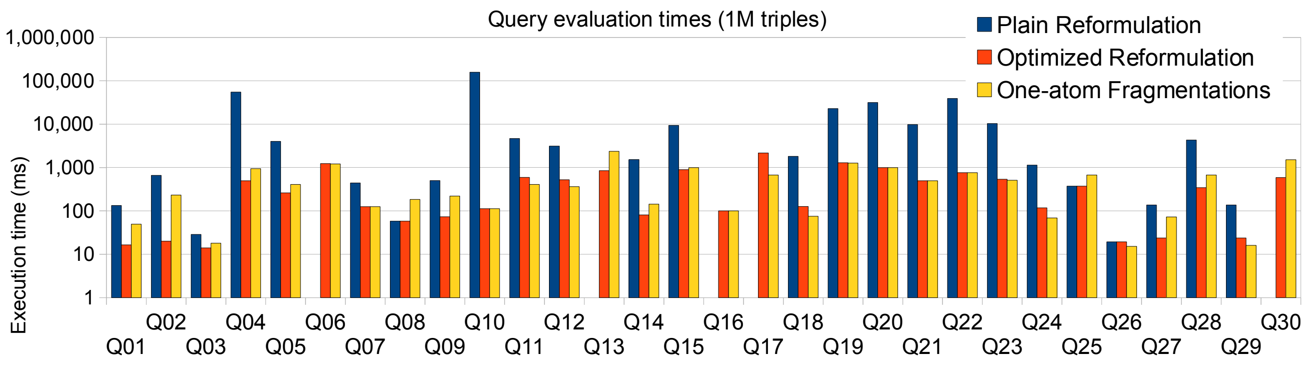


Figure 4: Query answering through plain, optimized and one-atom fragmentation reformulation.

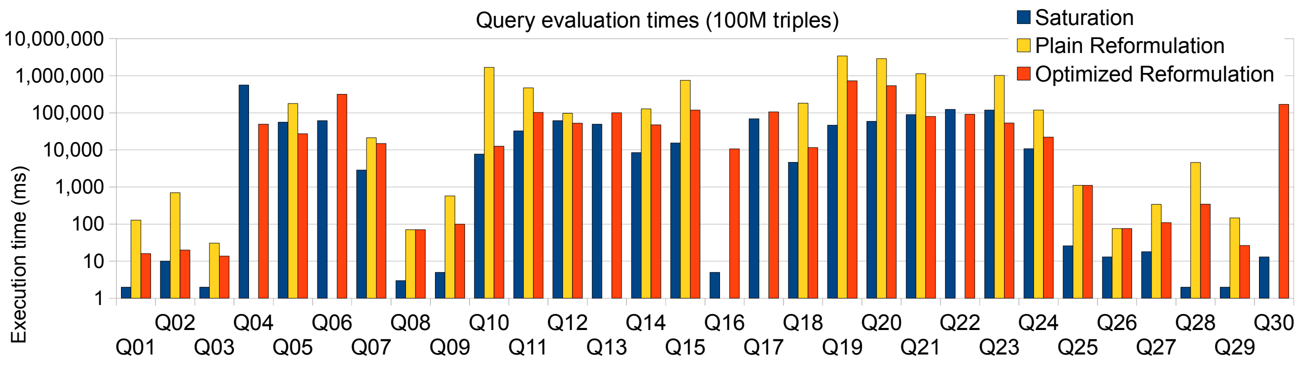


Figure 5: Query answering through saturation, plain and optimized reformulation.

Is an optimizer needed? The first question we ask is whether exploring the space of alternatives is actually needed, or could one just rely on a simple (fixed) fragmentation? Keeping all atoms in one fragment corresponds to the plain reformulation option, whose drawbacks we have illustrated in our motivating examples. Another simple alternative is to put each atom alone in one fragment (which we call *one-atom fragmentation* in the sequel). Figure 4 shows the evaluation time on 1M triples, for these two simple options together with our GCov-optimized reformulation (again, observe the logarithmic time axis). We see that optimized reformulation is in all but two cases (Q_{17} and Q_{18}, where the difference is not very large) the best of the three alternatives. While plain reformulation is overall the worst, one-atom fragmentation can still be more than one order of magnitude slower than the optimized one, as for instance in the case of Q_{02}.

(Optimized) reformulation compared with saturation. Figure 5 shows the evaluation times of: (*i*) queries over the saturated table; (*ii*) the plain reformulated query as produced by the algorithm of [3], (*iii*) the GCov-optimized reformulation. Notice the logarithmic vertical axis. Missing bars correspond to executions which could not complete on the Postgres server and that we stopped after 1 hour.

Figure 5 shows that GCov makes the evaluation of reformulated queries feasible in some cases where through plain reformulation it is not; this is, for instance, the case of Q_{15}, Q_{20} and Q_{21}. Further, for the queries where plain reformulation succeeds (and thus we can compare it with the optimized one), the GCov-chosen fragmentation is systematically faster than the plain reformulation, by up to two orders of magnitude.

SQL-specific discussion: reformulated query syntax. To complete our discussion of optimized reformulated query performance, we include here a short discussion of the impact of the SQL syntax used to express such queries, on their performance.

A cover-based JUCQ reformulation of a query q may be expressed in SQL in several ways, as the following example shows:

Example 5 (**SQL variants**). *Consider the JUCQ*

$q(x)$:- $q_1^{SQL} \bowtie_x (q_2^{SQL} \cup q_3^{SQL})$, *where*

$$\begin{array}{ll} q_1(x) & \text{:- } x \; \textit{dblp:author} \; \textit{ns1:Ann} \\ q_2(x) & \text{:- } x \; \text{rdf:type} \; \textit{Article} \\ q_3(x) & \text{:- } x \; \text{rdf:type} \; \textit{Book} \end{array}$$

Two distinct SQL syntaxes for this query are provided below. The first one, shown in Figure 6, defines q_3^{SQL} on one hand, and $(q_1^{SQL} \cup q_2^{SQL})$ on the other hand, as Common Table Expressions (or CTEs, in short), then it uses them in the main query.

An alternative syntax, shown in Figure 7, nests q_3^{SQL} and $(q_1^{SQL} \cup q_2^{SQL})$ within the FROM clause.

```
WITH authors AS (
            SELECT DISTINCT s FROM triples AS p3
            WHERE p3.p='dblp:Author'
             AND p3.o='ns1:Ann'),
      publications AS (
             SELECT DISTINCT s FROM triples AS p1
             WHERE p1.p='rdf:type'
             AND p1.o='dblp:Article'
             UNION
             SELECT DISTINCT s FROM triples AS p2
             WHERE p2.p='rdf:type'
             AND p2.o='dblp:Book'
      )
SELECT DISTINCT authors.s FROM authors, publications
WHERE authors.s=publications.s
```

Figure 6: CTE SQL syntax for the JUCQ in Example 5.

```
SELECT DISTINCT authors.s FROM
          (SELECT DISTINCT s FROM triples AS p3
           WHERE p3.p='dblp:Author' AND p3.o='ns1:Ann')
          AS authors,
          (SELECT DISTINCT s FROM triples AS p1
           WHERE p1.p='rdf:type' AND p1.o='dblp:Article'
           UNION
           SELECT DISTINCT s FROM triples AS p2
          WHERE p2.p='rdf:type' AND p2.o='dblp:Book')
          AS publications
WHERE authors.s=publications.s
```

Figure 7: Nested SQL syntax for the JUCQ in Example 5.

The two SQL syntaxes illustrated in Example 5 correspond to the following methods of composing the SQL query corresponding to the JUCQ:

— Define each union subquery as a *Common Table Expression* (or CTE, in short), used by the top-level join query. CTEs can be thought of as temporary tables that exist just for the duration of processing one query, and are supported by all major RDBMSs such as Oracle [26], DB2 [27], SQLServer [28] or Postgres [29].

— Defining each union subquery as a *nested query* and use it in the FROM clause of the top-level join query.

Figure 8 shows the evaluation time of the two SQL syntactic variants for our optimized reformulation as well as for the one-atom fragmentation. It can be seen that the CTE syntax is almost always faster (Q_{13} for the GCov-optimized reformulation being the lonely exception). However, the one-atom fragmentation evaluation times demonstrate that in some cases the speed-up obtained by using the CTE syntax is quite significant, above two orders of magnitude.

5.3 Trade-offs between saturation- and reformulation-based query answering

In this section, we compare the benefits and drawbacks of the two approaches by a quantitative analysis of all the costs involved. When relying on saturation, one has to take into account the time to saturate the database, maintain it when schema/data tuples are added to/deleted from the database, and saturation-based query answering. When reformulation is used, the only cost to consider is the one of (possibly optimized) reformulation-based query answering.

Table 2 summarizes the times to perform various operations related to saturation in the context of instance and schema updates.

To assess the trade-offs between reformulation and saturation, and following [3], we rely on a set of *thresholds* which quantify how many times must a query run, for the fixed cost of database saturation to be amortized (for *saturation to pay off*). More specifically, the *saturation threshold* of a query q, or $\mathbf{st}(q)$, the smallest integer n such that:

$$n \times \mathbf{t^{ref}}(q) > n \times \mathbf{t^{sat}}(q) + \mathbf{t_{sat+}}$$

In other words, n is the minimum number of times one needs to run q in order for the whole saturation cost to amortize.

Figure 9 shows the query specific saturation thresholds for the plain and optimized reformulation, in the case of the 1 million and 100 million triples datasets. Notice that for our optimized reformulation algorithm the thresholds increase (often by an order or magnitude), meaning that it takes much longer to compensate for the initial table saturation cost, making saturation a less convenient option. Also notice that some thresholds are missing in the case of plain reformulation marking the cases where the RDBMS was unable to run the reformulated queries.

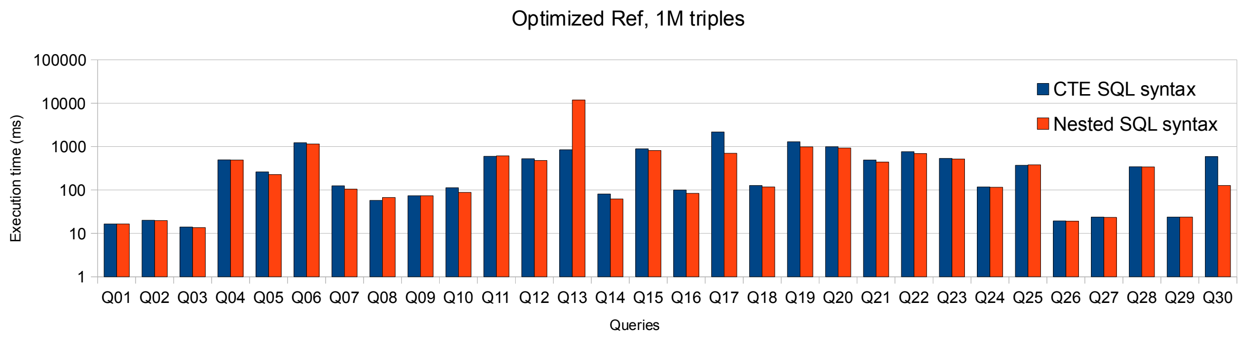
Similarly, a second set of thresholds shows how many times q should run in order for the *maintenance overhead due to one instance update* to pay off. We formalize this as follows.

Let $\mathbf{t}^+_{\mathsf{triple}}$ be the time to insert one statement in $\mathsf{triple}(s, p, o)$, and $\mathbf{t}^+_{\mathsf{sat}}$ be the time to propagate the insertion of one triple to the sat relation. Then, the *saturation threshold for an instance insertion*, denoted $\mathbf{st}^+_\mathbf{i}(q)$, is the smallest n for which:

$$n \times \mathbf{t^{ref}}(q) + \mathbf{t}^+_{\mathsf{triple}} > n \times \mathbf{t^{sat}}(q) + \mathbf{t}^+_{\mathsf{sat}}$$

In other words, $\mathbf{st}^+_\mathbf{i}(q)$ is the minimum number of times one needs to run q in order for the maintenance overhead due to the insertion of one triple (recall Table 2) to amortize. Figure 10(a) shows the query specific thresholds for instance insertions. Maintaining the saturation after a one triple insertion does not create a high computation overhead, therefore the insertion cost is amortized in one query execution for the majority of the tested queries. Exceptions here are the cases where the query evaluation time through reformulation is lower than the one obtained when using saturation. In such cases, the insertion cost can never be amortized.

Similarly, we define the *saturation threshold for an instance deletion*, denoted $\mathbf{st}^-_\mathbf{i}(q)$. Figure 10(b) shows the



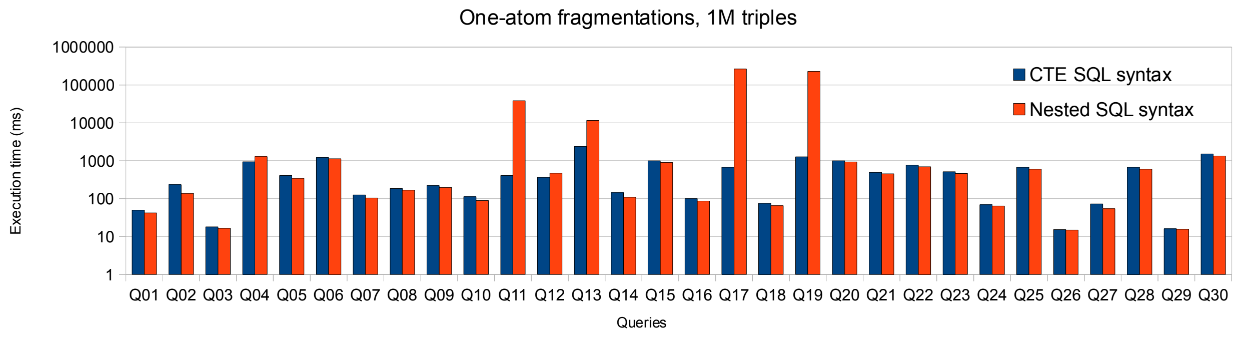


Figure 8: Impact of SQL syntax on GCov-optimized reformulation (top) and one-atom fragmentation (bottom).

Operation	Time (ms) 1 M	Time (ms) 10 M	Time (ms) 100 M
a) saturate triple table allowing maintenance	$60,376$	$595,870$	$5,728,652$
b) insert into triple table	6.97	8.62	44.50
c) delete from triple table	6.51	8.54	38.94
d) insert into and maintain saturation table	9.78	9.77	49.64
e) delete from and maintain saturation table	122.83	$1,075.67$	$34,682.84$
f) insert into the schema and maintain saturation table	$1,408.85$	$21,717.78$	$364,285.57$
g) delete from the schema and maintain saturation table	$1,951.12$	$32,265.10$	$1,210,486.43$

Table 2: LUBM saturation, instance and schema update times.

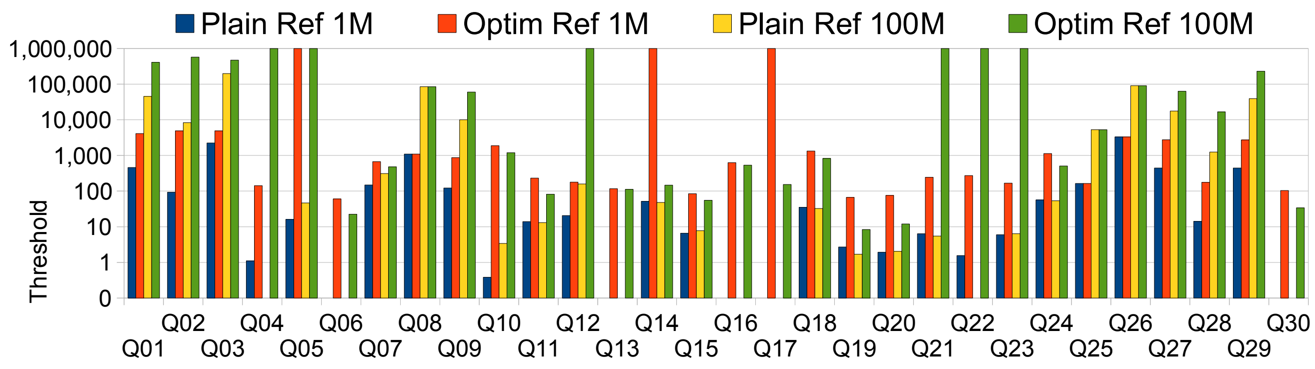
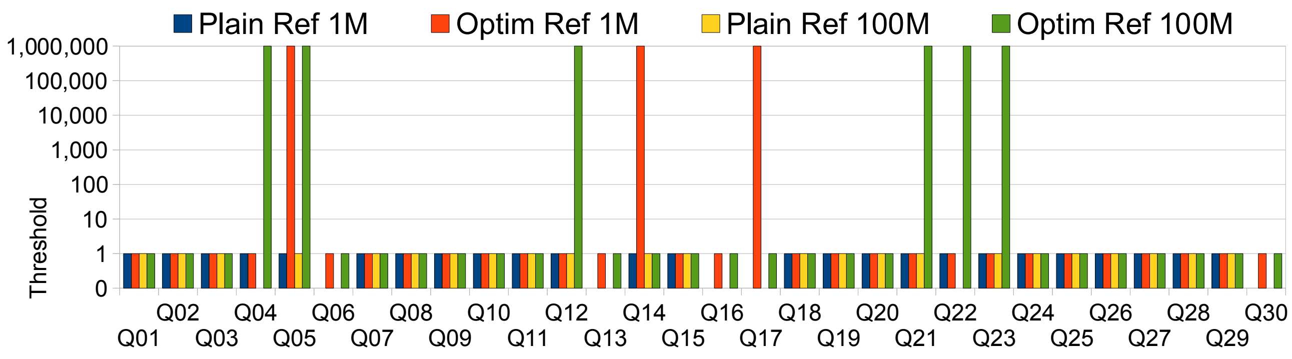


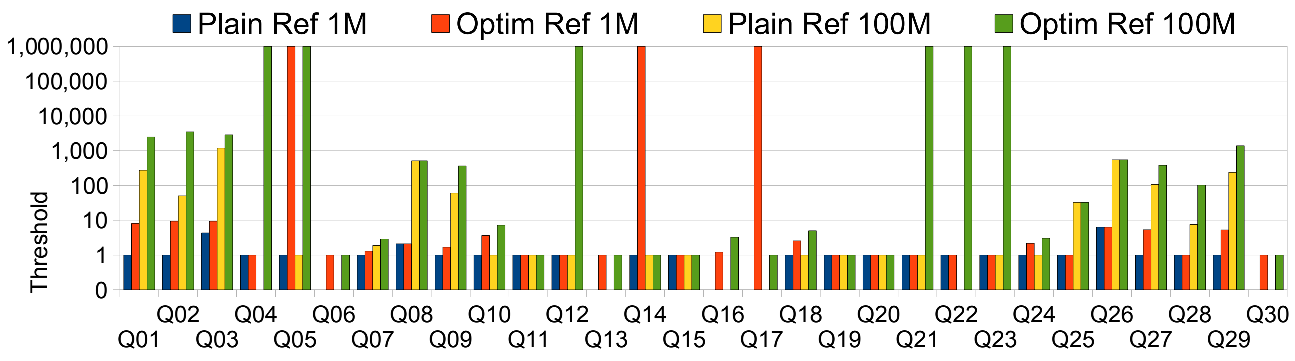
Figure 9: Saturation threshold.

threshold for instance deletion. We observe here the same trend as for saturation: the threshold increases as we optimize query reformulations, proving the interest of our reformulation alternatives.

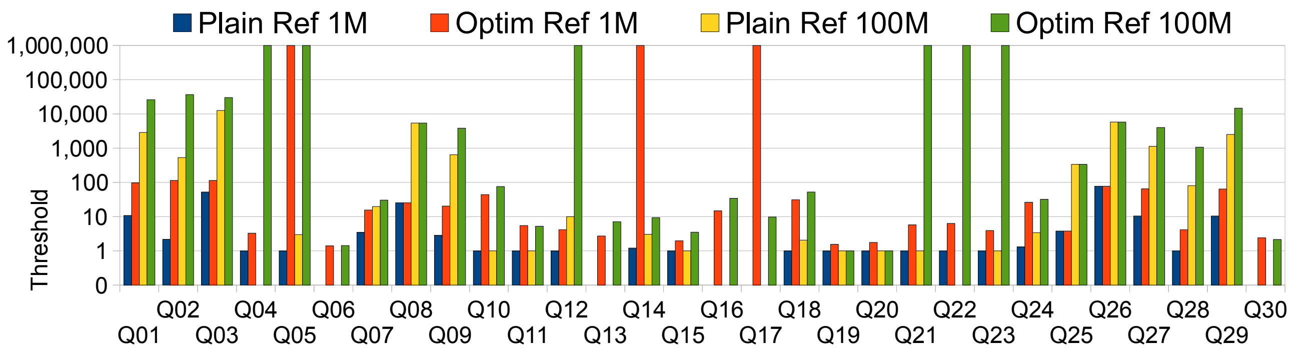
Similar to $\mathbf{st}(q)$, we define the *saturation threshold for a schema insertion* $\mathbf{st}_{\mathbf{s}}^{+}(q)$ and *deletion* $\mathbf{st}_{\mathbf{s}}^{-}(q)$, as the minimum number of times one needs to run q in order for the schema update cost to amortize. These thresholds (Figure 10(c) and 10(d)) again illustrate the improved performance of our reformulation alternatives.



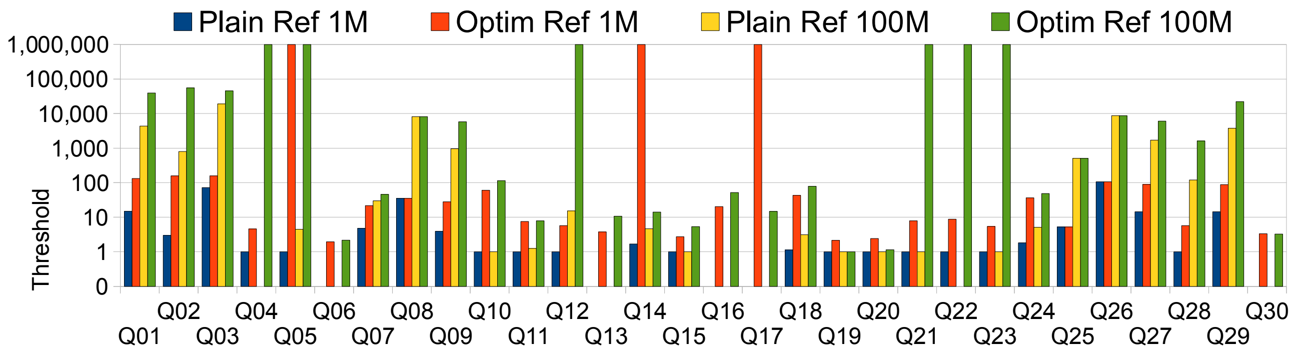
(a) Saturation threshold for one **instance insertion**.



(b) Saturation threshold for one **instance deletion**.



(c) Saturation threshold for one **schema insertion**.



(d) Saturation threshold for one **schema deletion**.

Figure 10: Saturation thresholds for one triple updates.

5.4 Comparison with query answering through Virtuoso

To investigate the interest of our RDBMS-based query answering technique, and to compare it with a quite different approach, we have also loaded the LUBM datasets within Virtuoso using the version 6.1.6 of the open source multi threaded edition. We loaded the already saturated LUBM datasets on the Virtuoso server and evaluated our 30 queries through Virtuoso' engine.

Figure 11 compares the query answering times through: Virtuoso evaluation over the saturated dataset on one hand, and Postgres-based saturation, respectively optimized reformulation (through the GCov algorithm) on the other hand, for two of the database sizes that we consider (note the logarithmic time axis).

We first comment on the *saturation-based alternatives*, that is those corresponding to the evaluation through Virtuoso and Postgres, on the saturated database. In almost all cases, the Postgres-based solution is faster than the Virtoso one, by up to three orders of magnitude (Q_{30} on 100 million triples); only for the query Q_{04}, Virtuoso is faster than Postgres and saturation-based query answering, by one order of magnitude. This validates the interest of a Postgres-based solution (where SQL query evalution is leveraged to handle large data volumes) to the problem of RDF query answering, and demonstrates its performance is competitive to that of a well-known dedicated RDF engine.

Second, comparing these saturation-based solution with the ones based on our optimized reformulation, we notice that optimized reformulation is overall comparable with saturation especially for expensive queries, whereas for some of the least expensive ones (such as Q_{08}, Q_{09} or Q_{16}, while the trend is globally followed, even optimized reformulation is still a few orders of magnitude slower than saturation-based solutions. This must be put into perspective according to the following two observations.

First, saturation-based strategies leverage the benefits of pre-computing all the instances of each atom from the original query, thus when considering a single query evaluation, it is to be expected that saturation is faster. Our focus in this work was to make *reformulation possible* (even in cases where it was not) and *as efficient as possible* (not to outperform saturation in all cases, although in some cases we do, e.g., Q_4, Q_{12} and Q_{22} on 100M triples in Figure 11 etc.).

Second, reformulation is especially interesting when the database changes frequently, and optimized reformulation strengthens that advantage, as demonstrated by the sometimes dramatic increase in the saturation threshods (Section 5.3).

5.5 Experiment conclusion

From our experiments we draw the following conclusions.

First, exploring the space of alternative JUCQs is interesting, and our proposed algorithm GCov does it efficiently, as witnessed by its performance superior to that of either plain reformulation or one-atom reformulation (Figure 4). The fragmentations identified by GCov make possible the evaluation of queries where plain reformulation is simply too expensive to evaluate (as illustrated by the missing bars in Figure 5); further, GCov-optimized reformulation may be up to two orders of magnitude faster than plain reformulation.

Second, given the particular shape of (plain) reformulated queries, which are joins of unions, when evaluation is made through an SQL engine, two syntaxes can be used. In our experiments with Postgres, the common table expression (CTE) option is best understood and evaluated by the server. Given that CTEs are supported in many systems (and also considering the associated implementation, suited for factorizing repeated sub-expressions), we believe it is the most reliable option.

Third, optimized reformulation is overall getting closer to the performance of saturation-based query answering, although the latter remains more efficient in most cases, as expected. However, when the saturation needs to be maintained due to data and schema updates, the number of runs needed for saturation to pay off (the so-called *thresholds* we studied in Section 5.3) may be quite high. As expected, insertions are easier to handle than deletions, and schema updates more difficult to handle than data updates. Many RDF data management scenarios (in particular integration of distinct datasets, or operational databases where data is continuosly added and deleted) involve frequent changes to the database; (optimized) reformulation is an attractive option for these.

6. RELATED WORK

The problem addressed in our work can be stated as *answering conjunctive queries against RDF data, in the presence of RDFS constraints.*

A first dimension characterizing the problem is the RDF dialect considered. Previous works have focused on the description logic [4] fragment of RDF and the relational conjunctive SPARQL subset [5, 6, 7], and extensions thereof [8, 9, 10, 11]. This work is placed in the *database fragment of RDF* [3] which is currently the most expressive dialect of RDF for which reformulation-based query answering algorithms are available.

A second classification dimension is whether the reasoning required by query answering takes place: (*i*) *statically* (independently of a query being asked), or (*ii*) *dynamically (at runtime)*, dictated by the needs of the query. Works belonging to each of these classes are discussed below.

Static reasoning (saturation). When using static reasoning (or, equivalently, saturation), all the implicit triples are computed and explicitly added to the database; query answering then reduces to query evaluation on the saturated database. Well-known SPARQL compliant RDF platforms such as 3store [30], Jena [31], OWLIM [32], Sesame [33], Oracle Semantic Graph [34] support saturation-based query answering, based on (a subset of) RDF entailment rules.

RDF platforms originating in the data management community, such as Hexastore [23] or RDF-3X [22], ignore entailed triples and only provide query *evaluation* on top of the RDF graph, which is assumed to be saturated. Thus, query answering in the presence of implicit triples in such systems also belongs to the saturation category.

Parallel RDF saturation algorithms have been proposed in the literature. In WebPie [35], an RDF graph is stored in a distributed file system and the saturation of the graph is computed using MapReduce jobs. A similar approach, based on C/Message Passing Interface, is presented in [36].

The drawbacks of saturation w.r.t. updates have been pointed out in [37], which proposes a *truth maintenance* technique implemented in Sesame. It relies on the storage and management of the *justifications* of entailed triples

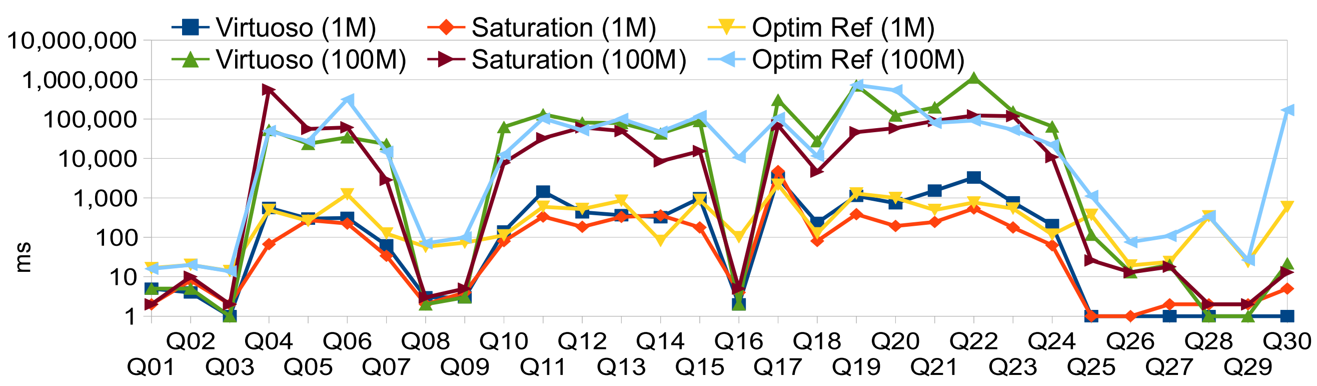


Figure 11: Query answering through Virtuoso and Postgres (via saturation, respectively, optimized reformulation).

(which triples beget them). While efficient on graphs with few entailed triples, the technique is pegged by the high overhead of handling justifications when their number and size grow. Therefore, [38] proposes to compute only the relevant justifications w.r.t. an update, at maintenance time. This technique is implemented in OWLIM, however [32] points out that schema-level deletions can lead to poor performance. A more efficient saturation maintenance technique is provided in [3] based on the *number of times* triples are entailed; this is easier to store and manipulate. A distinct yet related problem is finding which triples to delete from an RDF graph, so that an implicit triple no longer holds [39].

Run-time reasoning (reformulation). The alternative approach is based on run-time reasoning, or *reformulating* a given BGP query into a target language, whose evaluation (by an appropriate run-time engine) computes the query answer. Among the well-established RDF data management systems, the only ones supporting run-time reasoning are Virtuoso (which supports only the rdfs:subClassOf and rdfs:subPropertyOf RDFS rules) and AllegroGraph [40] which supports the four RDFS rules but whose reasoning implementation is incomplete [4].

Simple dialects of SQL have been used as the reformulation language in [3] and also in Virtuoso. The former work corresponds to one point in the space of alternatives considered in the present work, namely: the fragmentation consisting of the set of all the query atoms. In contrast, in Virtuoso, the only alternative considered corresponds to another point in the space of alternatives considered in the present work, namely: the fragmentation consisting of one set per query atom. Our technique may make more complex choices, and our experiments have shown that these alternative may lead to very significant performance improvements.

A distinct approach is to use Datalog as a target reformulation language. For instance, Presto [41, 42] reformulates queries in a DL-Lite setting into non-recursive Datalog programs. These works rely on DL-Lite formalisms, strictly more expressive from a semantic constraint viewpoint than the RDFS constraints we consider. Thus, their method could be easily transferred (restricted) to the DL fragment of RDF which, as previously mentioned, is a subset of the database fragment of RDF that we consider. However, reformulation methods under DL-Lite constraints and using Datalog as a target language did not consider cost-driven performance optimization based on data statistics and a query evaluation cost model as we do in our work.

From a *database optimization* perspective, the performance advantage we gain by clustering selective atoms next to very large ones is akin to the semi-join reducers technique, well-known from the distributed database context [43]. It has been shown e.g., in [44] that semi-join reducers can also be beneficial in a centralized context by reducing the overall join effort. In this work, we use a technique reminiscent of semi-joins in order to pick the best *query-level* formulation of a large (union of joins) query, to make its evaluation possible and efficient; this contrasts with the traditional usage of semi-joins *at the level of algebraic plans.* On one hand, working at the plan level enables one to intelligently combine traditional joins and semi-joins to obtain the best performance. On the other hand, producing (as we do) an output at the *query (syntax)* level (recall Figure 1) enables us to take advantage of any existing system, and of its optimizer which will figure out the best way to evaluate such queries, a task at which many systems are good once we brought the query to a "reasonable" shape. Further, expressing optimized reformulations as queries allows us *not* to (re-)explore the search space of join orders etc. together with the (already large) space of possible fragmentations.

7. CONCLUSION

An important class of applications requiring scalable data processing and flexible semantic constraints originates in the Semantic Web.

Our work is placed in the setting of query answering against RDF graphs in the presence of RDF Schema constraints. Two methods have been investigated in this context. First, one may derive all consequences of the constraints, or, equivalently in our context, compute all the entailed or derived triples, and store them in the database next to the explicit data; this is termed saturation. On a saturated database, query answering is reduced to query evaluation. Alternatively, one can leave the database unchanged and reformulate queries asked against the data, so that the reformulated query, when evaluated (through standard query evaluation techniques) against the database, returns the same answer as if the original query was evaluated on the saturated database.

While the performance of saturation has been the focus of

4. As stated at http://franz.com/agraph/support/documentation/current/reasoner-tutorial.html#header2-13.

many previous works, little or no effort has been invested in *making reformulation-based query answering more efficient*; this is the goal of the present work.

We have identified a space of alternative JUCQ reformulations, whose evaluation (based on a standard, semantics-unaware query processor) may be (*i*) feasible even when plain reformulation is not, and (*ii*) more efficient, in some cases by orders of magnitude. Further, we have presented a cost model for such JUCQ alternatives, and proposed an anytime greedy cost-based algorithm capable of identifying such efficient alternatives. We have shown that through this optimization, reformulation-based query answering is a better alternative than saturation-based answering, and may actually reach (and overcome) its performance in some cases.

Generally speaking, saturation-based query answering has an obvious performance advantage, due to the significant amount of precomputed results; thus, it is probably preferable in static contexts where the RDF database changes little if at all, and less interesting in dynamic ones where changes to the data and schema are frequent.

Our work has focused in particular on conjunctive query reformulation for the database fragment of RDF, and we have mostly experimented with Postgres. However, *our study applies in a broader setting*, as outlined below.

First, as we have shown, the performance of our Postgres-based solution is comparable with (and typically better than) an efficient native RDF system, namely Virtuoso.

Second, while different RDBMSs may implement other optimization strategies, the observation that the evaluation of plain reformulated queries is really challenging is independent of the system being tested, and in particular of Postgres that we used. Recall from Table 1 that these are very large queries, with many repeated subexpressions, and their evaluation is inherently hard.

Third, our approach improves the performance of reformulation-based query answering in any setting where the plain reformulated query is evaluated by a conjunctive query processor, which could for instance be a SPARQL endpoint, not necessarily an RDBMS.

Fourth and finally, as we have discussed in the introduction, reformulated queries such as the ones we consider can be produced by other reformulation algorithms, potentially using different (classes of) constraints.

As part of our future work, we plan to enlarge the set of platforms on which to study reformulation-based query answering, and to investigate heuristic search algorithms for identifying efficient fragmentations faster, and possibly identifying better ones.

8. REFERENCES

[1] The World Wide Web Consortium (W3C). Resource description framework. http://www.w3.org/RDF.

[2] The World Wide Web Consortium (W3C). SPARQL protocol and RDF query language. http://www.w3.org/TR/rdf-sparql-query.

[3] François Goasdoué, Ioana Manolescu, and Alexandra Roatis. Efficient query answering against dynamic RDF databases. In *EDBT*, 2013.

[4] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[5] Philippe Adjiman, François Goasdoué, and Marie-Christine Rousset. SomeRDFS in the semantic web. *JODS*, 8, 2007.

[6] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *Journal of Automated Reasoning (JAR)*, 39(3), 2007.

[7] Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Ontological queries: Rewriting and optimization. In *ICDE*, 2011. Keynote.

[8] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. Foundations of RDF databases. In *Reasoning Web*, 2009.

[9] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. View selection in semantic web databases. *PVLDB*, 2011.

[10] Zoi Kaoudi, Iris Miliaraki, and Manolis Koubarakis. RDFS reasoning and query answering on DHTs. In *ISWC*, 2008.

[11] Jacopo Urbani, Frank van Harmelen, Stefan Schlobach, and Henri Bal. QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases. In *ISWC*, 2011.

[12] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank van Harmelen, and Henri E. Bal. WebPIE: A web-scale parallel inference engine using MapReduce. *J. Web Sem.*, 10:59–75, 2012.

[13] PostgreSQL. http://www.postgresql.org.

[14] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[15] Tomasz Imielinski and Witold Lipski Jr. Incomplete information in relational databases. *JACM*, 31(4), 1984.

[16] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.

[17] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. View selection in semantic web databases. *PVLDB*, 5(2), 2011.

[18] François Picalausa, Yongming Luo, George H.L. Fletcher, Jan Hidders, and Stijn Vansummeren. A structural approach to indexing triples. In *The Semantic Web: Research and Applications*, pages 406–421, 2012.

[19] Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, and Pierre Senellart. *Web Data Management.* Cambridge University Press, 2011.

[20] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[21] T. Hearne and C. Wagner. Minimal covers of finite sets. *Discrete Mathematics*, 5:247–251, 1973.

[22] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *VLDBJ*, 19(1):91–113, 2010.

[23] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple indexing for Semantic Web data management. *PVLDB*, 2008.

[24] Java^{TM}. http://docs.oracle.com/javase/7/docs/.

[25] Orri Erling and Ivan Mikhailov. RDF Support in the Virtuoso DBMS. *Networked Knowledge - Networked Media*, pages 7–24, 2009.

[26] Oracle CTEs (subquery_factoring_clause). http://docs.oracle.com/cd/E11882_01/server.112/e26088/statements_10002.htm#SQLRF55267.

[27] DB2 CTEs. http://www-01.ibm.com/support/knowledgecenter/#!/SSEPEK_11.0.0/com.ibm.db2z11.doc.sqlref/src/tpc/db2z_sql_commontableexpression.dita.

[28] SQL server CTEs. http://technet.microsoft.com/en-us/library/ms190766(v=sql.105).aspx.

[29] Postgres CTEs. http://www.postgresql.org/docs/9.2/static/queries-with.html.

[30] 3store. http://www.aktors.org/technologies/3store.

[31] Apache jena^{TM}: Java framework for building semantic web applications. http://jena.apache.org.

[32] Owlim. http://owlim.ontotext.com.

[33] Sesame. http://www.openrdf.org.

[34] Oracle semantic graphs. http://docs.oracle.com/cd/E16655_01/appdev.121/e17895/toc.htm, 2014.

[35] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Scalable Distributed Reasoning using MapReduce. In *8th International Semantic Web Conference (ISWC)*, 2009.

[36] Jesse Weaver and James A. Hendler. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In *ISWC*, 2009.

[37] Jeen Broekstra and Arjohn Kampman. Inferencing and truth maintenance in RDF Schema: Exploring a naive practical approach. In *PSSS Workshop*, 2003.

[38] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. OWLIM: A family of scalable semantic repositories. *Semantic Web*, 2(1), 2011.

[39] Claudio Gutierrez, Carlos A. Hurtado, and Alejandro A. Vaisman. RDFS update: From theory to practice. In *ESWC*, 2011.

[40] AllegroGraph RDFStore Web 3.0 Database. http://franz.com/agraph/allegrograph, 2014.

[41] Riccardo Rosati and Alessandro Almatelli. Improving query answering over DL-Lite ontologies. In *KR*, 2010.

[42] Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. MASTRO: A reasoner for effective ontology-based data access. In *ORE*, 2012.

[43] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[44] Konrad Stocker, Reinhard Braumandl, Alfons Kemper, and Donald Kossmann. Integrating semi-join-reducers into state-of-the-art query processors. In *ICDE*, 2001.

Q01(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Employee”, ?X “http://www.lehigh.edu/univ-bench.owl#worksFor” “http://www.Department0.University0.edu”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” ?Y
Q02(?X ?Y ?U ?V ?W) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Employee”, ?X “http://www.lehigh.edu/univ-bench.owl#worksFor” “http://www.Department0.University0.edu”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” ?Y, ?X “http://www.lehigh.edu/univ-bench.owl#name” ?U, ?X “http://www.lehigh.edu/univ-bench.owl#emailAddress” ?V, ?X “http://www.lehigh.edu/univ-bench.owl#telephone” ?W
Q03(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/Univ.owl#Employee”, ?X “http://www.lehigh.edu/univ-bench.owl#worksFor” “http://www.Department0.University0.edu”, ?X “http://www.lehigh.edu/univ-bench.owl#doctoralDegreeFrom” ?Y
Q04(?X ?Y ?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?Y, ?U “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#University”, ?X “http://www.lehigh.edu/univ-bench.owl#doctoralDegreeFrom” ?U, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” ?Z
Q05(?X ?Y ?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Student”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?Z “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Course”, ?X “http://www.lehigh.edu/univ-bench.owl#advisor” ?Y, ?Y “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?Z, ?X “http://www.lehigh.edu/univ-bench.owl#takesCourse” ?Z
Q06(?X ?W ?Y ?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?W, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?Z “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Course”, ?X “http://www.lehigh.edu/univ-bench.owl#advisor” ?Y, ?Y “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?Z, ?X “http://www.lehigh.edu/univ-bench.owl#takesCourse” ?Z
Q07(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” ?Y, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” ?Y
Q08(?X) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Person”, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” “http://www.Department0.University0.edu”
Q09(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Person”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” ?Y, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” “http://www.Department0.University0.edu”
Q10(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Professor”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Professor”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” ?U, ?Y “http://www.lehigh.edu/univ-bench.owl#degreeFrom” ?V, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” ?V, ?Y “http://www.lehigh.edu/univ-bench.owl#memberOf” ?U

Table 3: Queries Q1-Q10.

Q11(?W ?X ?Y) :- ?W “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Publication”, ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateStudent”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?W “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?X, ?W “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?Y
Q12(?W ?X ?Y) :- ?W “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Publication”, ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateStudent”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?X “http://www.lehigh.edu/univ-bench.owl#advisor” ?Y, ?W “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?X, ?W “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?Y
Q13(?W ?X ?Y) :- ?W “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Publication”, ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateStudent”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?Z “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Course”, ?X “http://www.lehigh.edu/univ-bench.owl#advisor” ?Y, ?Y “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?Z, ?X “http://www.lehigh.edu/univ-bench.owl#takesCourse” ?Z, ?W “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?X, ?W “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?Y
Q14(?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Student”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateStudent”, ?Z “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?X “http://www.lehigh.edu/univ-bench.owl#advisor” ?Z, ?Y “http://www.lehigh.edu/univ-bench.owl#advisor” ?Z
Q15(?Z ?W) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/~zhp2/univ-bench.owl#Student”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateStudent”, ?Z “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?W, ?X “http://www.lehigh.edu/univ-bench.owl#advisor” ?Z, ?Y “http://www.lehigh.edu/univ-bench.owl#advisor” ?Z
Q16(?X) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#University”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Publication”, ?U “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?Z, ?Y “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?U, ?U “http://www.lehigh.edu/univ-bench.owl#memberOf” ?X
Q17(?Z) :- ?Z “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Student”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateStudent”, ?U “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Course”, ?V “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Course”, ?Z “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?U, ?Z “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?V, ?X “http://www.lehigh.edu/univ-bench.owl#takesCourse” ?U, ?Y “http://www.lehigh.edu/univ-bench.owl#takesCourse” ?V
Q18(?X) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateCourse”, ?Z “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Course”, ?X “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?Y, ?X “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?Z

Table 4: Queries Q11-Q18.

Q19(?X ?W) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?W, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateCourse”, ?Z “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Course”, ?X “http://www.lehigh.edu/univ-bench.owl#takesCourse” ?Y, ?X “http://www.lehigh.edu/univ-bench.owl#takesCourse” ?Z
Q20(?X ?Z ?W) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?Z “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?W, ?Z “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?X
Q21(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Person”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Publication”, ?Y “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?X
Q22(?X ?Y ?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Person”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Publication”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” ?Z, ?Y “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?X
Q23(?X ?Y ?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Person”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Publication”, ?X “http://www.lehigh.edu/univ-bench.owl#doctoralDegreeFrom” ?Z, ?Y “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?X
Q24(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Course”, ?X “http://www.lehigh.edu/univ-bench.owl#doctoralDegreeFrom” ?Z, ?X “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?Y
Q25(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?Y, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” “http://www.University0.edu”
Q26(?X) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” “http://www.University0.edu”
Q27(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” “http://www.University532.edu”, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” ?Y
Q28(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?Y, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” “http://www.University532.edu”, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” “http://www.Department1.University7.edu”
Q29(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” “http://www.University532.edu”, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” “http://www.Department1.University7.edu”
Q30(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?U, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?V, ?X “http://www.lehigh.edu/univ-bench.owl#mastersDegreeFrom” “http://www.University532.edu”, ?Y “http://www.lehigh.edu/univ-bench.owl#doctoralDegreeFrom” “http://www.University532.edu”, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” ?Z, ?Y “http://www.lehigh.edu/univ-bench.owl#memberOf” ?Z

Table 5: Queries Q19-30.