Understanding and Improving Reformulation-Based Query Answering Performance in RDF

Damian Bursztyn, François Goasdoué, Ioana Manolescu

Understanding and Improving Reformulation-Based Query Answering Performance in RDF

Damian Bursztyn
INRIA & U. Paris-Sud, France
damian.burstzyn@inria.fr

François Goasdoué
U. Rennes 1 & INRIA, France
fg@irisa.fr

Ioana Manolescu
INRIA & U. Paris-Sud, France
ioana.manolescu@inria.fr

ABSTRACT

Answering queries over Semantic Web data, i.e., RDF graphs, must account for both *explicit* and *implicit* data, entailed by the explicit data and the *semantic constraints* holding on them. Two main query answering techniques have been devised, namely *Saturation*-based (SAT) which precomputes and adds to the graph all implicit information, and *Reformulation*-based (REF) which reformulates the query based on the graph constraints, so that evaluating the reformulated query directly against the explicit data (i.e., without considering the constraints) produces the query answer.

While SAT is well known, REF has received less attention so far. In particular, reformulated queries often perform poorly if the query is complex. Our demonstration [6] showcases a large set of REF techniques, including but not limited to one we proposed recently [5]. The audience will be able to 1. test them against different datasets, constraints and queries, as well as different well-established systems, 2. analyze and understand the performance challenges they raise, and 3. alter the scenarios to visualize the impact on performance. In particular, we show how a *cost-based* REF *approach* allows avoiding reformulation performance pitfalls.

1. INTRODUCTION

The efficient management of complex, semantic-rich Web data is a hot topic within the Databases, Semantic Web, and Knowledge Representation communities. In particular, the former has produced many techniques for storing, indexing, querying and updating such data, e.g., [4, 13, 14, 17], while the latter have mostly focused on expressive semantic languages to describe the meaning of the data, e.g., [2, 3, 7]. Currently, technical interest seems to be split between the experts in "query evaluation", which consider large databases and complex queries, but tend to ignore the data semantics, and the experts in "reasoning", whose main focus is on the knowledge description formalisms. As an unfortunate consequence, reasoning is rarely considered in database systems and prototypes handling Semantic Web data. This makes them ill-adapted to real-life applications, which are rich in *constraints* describing the properties of the data [16]; such constraints must be taken into account in order to compute *correct* results, and do so *efficiently*.

One possible reason for disregarding semantics is that for popular data models (such as the W3C's Resource Description Framework, or RDF in short) and associated constraint languages (such as RDF Schema, or RDFS), *constraints can be compiled in the database*, by materializing in the data all possible consequences of the constraints. For instance, if a constraint states that any *Manager* is an *Employee*, given a database D, one can build another database D' by adding to D an *Employee* instance for each *Manager* from D. This can be seen as making *explicit* in D', the instances of *Employee* which were *implicit* in D; the process is called materialization or *saturation*. To *answer* a query over the original database D under the above constraint, one can just *evaluate* the query over D', ignoring the constraint (since its effects are fully reflected in D'). We use SAT to designate the *saturation-based query answering technique* outlined above.

SAT is rather simple and well-understood. However, the saturation needs to be maintained after changes in the data and/or constraints, which may incur a performance penalty. Further, Semantic Web data is often found not in a single repository, but in a set of independent ones, typically called *RDF endpoints*; a set of well-known endpoints are listed in the Linked Open Data Cloud[1]. Data in each such independent source may or may not be saturated; further, implicit facts may be due to the presence of one fact in one endpoint, and a constraint in another. Computing the complete (distributed) set of consequences in this setting is unfeasible, especially considering that such sources often return only restricted answers (e.g., the first 50) to a query, to avoid overloading their servers.

The alternative technique is based on *query reformulation*. It leaves the database unchanged, but changes the given query Q into a query Q' which, evaluated over the original database D, returns the answer of Q against D', reflecting both the implicit and the explicit data. In the simple example above, if Q asks for all the *Employee*s, it is reformulated into Q' returning the union of all *Employee*s and all *Manager*s. We term this technique *reformulation-based query answering*, and denote it REF. While it has obvious advantages (it does not require credentials or space to store implicit data, nor the effort to maintain saturation), depending on the language in which the reformulated query is

BDA 29 septembre 2015, ÃŐle de Porquerolles, France.

[1] `http://linkeddatacatalog.dws.informatik.uni-mannheim.de/state/LODCloudDiagram.html`

expressed, which we term a *reformulation strategy*, reformulation may lead to very large queries, whose evaluation is inefficient or even infeasible, making REF non practical in general.

Our demonstration aims at *showcasing to the audience the performance challenges* raised by reformulation-based query answering. We will prepare a *set of scenarios (data, constraints, and queries)* and allow the audience to *experiment with a variety of* REF *strategies*, and evaluating them through performant relational database management systems (RDMBSs, in short): DB2, Postgres and MySQL. To enlarge the comparison, we also include native RDF systems using their own fixed (incomplete) REF strategy, Virtuoso and AllegroGraph, as well as a query answering technique based on translating scenarios to Datalog programs and resorting to the LogicBlox engine for evaluation. In particular, we show that (*i*) *a fixed reformulation strategy may lead to very bad performance or simply fail* - on moderate-size databases and simple constraints - on all the systems, because *reformulated queries may be syntactically huge* and (*ii*) *a cost-based query reformulation approach allows avoiding such performance pitfalls* and makes REF feasible - and efficient - in the same setting(s).

2. RELATED WORK

A thorough discussion of RDF REF and SAT can be found in [5, 10]; we recall the most relevant works here. Most RDF data management systems use SAT, either providing a saturation service, like 3store, OWLIM, Sesame, etc., or by simply assuming that RDF graphs have been saturated prior to loading. RDF platforms built on top of RDBMSs [4], or RDBMS-style engines, e.g., [13, 14, 17] fall in this category.

REF has also been the topic of many works [8, 15, 18, 19], including ours [9]. Existing techniques apply to the Description Logics (DLs) [3] fragment of RDF, the conjunctive subset of SPARQL and extensions thereof [2, 7, 15, 19], including the "database fragment" of RDF we introduced in [9], the most expressive RDF fragment for which REF techniques are known. Only a few RDF data management systems, such as AllegroGraph, Stardog or Virtuoso, use reformulation, in some cases incomplete (ignoring some RDFS constraints) [10].

A query is typically reformulated into an equivalent *large union of conjunctive queries (*`UCQ`*)* w.r.t. the RDF Schema constraints [7, 8, 9, 15, 19], or in a language currently not well supported by available engines, e.g., nested SPARQL [2]. The technique of [18], when translated to the RDF setting, reformulates a conjunctive query into a join of unions of atomic queries, called a *semi-conjunctive query* (`SCQ`).

In our recent work [5], we devised a novel strategy for improving REF performance and robustness. Instead of reformulating into a fixed `UCQ` or `SCQ`, we have identified a *space of alternative reformulations*, corresponding to an *enlarged reformulation language* consisting of *joins of unions of conjunctive* queries (denoted `JUCQ`s in the sequel). `UCQ` and `SCQ` reformulations are each just a point in this space. The evaluation performance of distinct `JUCQ`s from this space may differ by *several orders of magnitude*; we devised a cost model and used it in a greedy search algorithm to find out the `JUCQ` whose evaluation is likely to be most efficient.

SAT and REF are combined in [19]; the resulting reformulated query may still be large, thus hard to evaluate.

Assertion	Triple	Relational notation
Class	`s` rdf:type `o`	`o(s)`
Property	`s p o`	`p(s,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 1: RDF (top) & RDFS (bottom) statements.

3. PRELIMINARIES

3.1 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 triples, as per the W3C's RDF specification, using uniform resource identifiers (URIs), typed or un-typed literals (constants), and *blank nodes* (unknown URIs or literals) corresponding to a form of incomplete information.

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

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

For example, the RDF graph `G` shown below describes a book, identified by doi_1: its author (a blank node $\_{:}b_1$ related to the author name), title and date of publication.

$$\mathtt{G} = \begin{array}{l}\{\text{doi}_1 \text{ rdf:type Book},\ \text{doi}_1 \text{ writtenBy } \_{:}b_1,\\ \ \text{doi}_1 \text{ hasTitle "El Aleph"},\\ \ \_{:}b_1 \text{ hasName "J. L. Borges"},\\ \ \text{doi}_1 \text{ publishedIn "1949"}\}\end{array}$$

RDF Schema allows enhancing the descriptions in RDF graphs by means of *RDFS triples*, declaring *semantic constraints* between the classes and the properties used in those graphs. Figure 1 (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 1) are interpreted under the open-world assumption (OWA) [1].

RDF entailment. *Implicit triples* may be part of the RDF graph even though they are not explicitly present in it. W3C names *RDF entailment* the mechanism through which, based on the 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). For instance, assume that the RDF graph `G` above is extended with the following constraints.

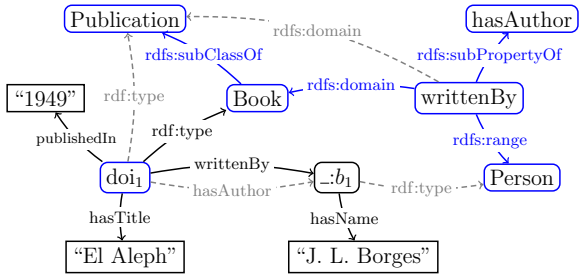

Figure 2: Sample RDF graph.

- books are publications:
  Book rdfs:subClassOf Publication
- writing something means being an author:
  writtenBy rdfs:subPropertyOf hasAuthor
- writtenBy is a relation between books and people:
  writtenBy rdfs:domain Book and
  writtenBy rdfs:range Person

The resulting graph is depicted in Figure 2. 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 $\mathtt{G}$, which is the RDF graph $\mathtt{G}^{\infty}$ defined as the fixed-point obtained by repeatedly applying $\vdash^{i}_{\text{RDF}}$ rules on $\mathtt{G}$.

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

3.2 Conjunctive Queries

We consider the widely used SPARQL dialect consisting of (unions of) *basic graph pattern* (BGP) queries, a.k.a. conjunctive queries (CQs), widely considered in research but also in real-world applications [16]. A BGP is a set of *triple patterns*, or triples/atoms in short. Each triple has a subject, property and object, some of which can be variables.

Notations. We use the CQ 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 in $t_1, \ldots, t_\alpha$; for boolean queries $\bar{x}$ is empty. The head of q is $q(\bar{x})$, its body is $t_1, \ldots, t_\alpha$; x, y, z, etc. denote variables.

Query answering. The evaluation of a CQ q against $\mathtt{G}$ has access only to $\mathtt{G}$'s explicit triples, thus may lead to an incomplete answer. The (complete) answer of q against $\mathtt{G}$ is obtained by the evaluation of q against $\mathtt{G}^{\infty}$. For instance, the query below asks for the names of authors of books somehow connected to the literal 1949:

$q(x_3)$:- x_1 hasAuthor x_2, x_2 hasName x_3, x_1 x_4 "1949"

Its answer against the graph in Figure 2 is $q(\mathtt{G}^{\infty}) = \{\langle$"J. L. Borges"$\rangle\}$. Note that evaluating q only against $\mathtt{G}$ leads to the empty answer, which is obviously incomplete.

3.3 Reformulation-based query answering

The *database (DB)* fragment of RDF [9] is the most expressive RDF fragment for which both *saturation-* and *reformulation-based query answering techniques* have been defined. Its name comes from the fact that query answering against any graph from this fragment can be easily implemented on top of any RDBMS.

The DB fragment is defined by: (i) *Restricting RDF entailment* to the RDF Schema constraints only (Figure 1), a.k.a. RDFS entailment. While simple, these allow expressing many practical application domain (ontological) constraints. (ii) *Not restricting RDF graphs in any way.* In other words, any triple allowed by the RDF specification is also allowed in the DB fragment.

The associated query reformulation algorithm devised in [9] exhaustively applies a set of 13 reformulation rules based on RDFS constraints. Starting from a CQ query q to answer against db, the algorithm produces a UCQ reformulation q^{ref} using the constraints in a backward-chaining fashion, which retrieves the complete answer to q out of the (non-saturated) db: $q(\mathtt{db}^{\infty}) = q^{\text{ref}}(\mathtt{db})$.

Triple	#answers	#reformulations	#answers after reformulation
(t_1)	18,999,082	188	33,328,108
(t_2)	0	4	3,223
(t_3)	396	3	683

Table 1: Characteristics of the sample query q_1.

4. OPTIMIZED REFORMULATION

We illustrate performance challenges raised by the evaluation of state-of-the-art reformulated queries, and how our cost-based approach [5] allows tackling them.

Example 1. *Let* q_1 *be the query:*

$$q_1(x,y) \text{ :- } \begin{array}{ll} x \text{ rdf:type } y, & (t_1)\\ x \text{ ub:degreeFrom "http://www.Univ532.edu"}, & (t_2)\\ x \text{ ub:memberOf "http://www.Dept1.Univ7.edu"} & (t_3)\end{array}$$

Table 1 gives some intuition on the difficulty of answering q_1 *over an* 10^8 *triples LUBM [12] benchmark dataset, evaluated through PostgreSQL 9.3.2 on a 8-core Intel Xeon (E5506) 2.13 GHz machine with 16GB RAM, using Mandriva Linux release 2010.0 (Official).*

The state-of-the-art REF *technique reformulating a* CQ *into an* UCQ *computes the answer to* q_1 *by evaluating a reformulated query* q'_1*, which is a union of* 2,256 *conjunctive queries, each of which consists of three triples (one for the reformulation of each triple in the original* q_1*). This query* q'_1 *appears in Table 2, where all the triples* t_1, t_2, t_3 *are reformulated together by a* CQ *to* UCQ *reformulation algorithm denoted* $(.)^{ref}$*. Observe that in* q'_1*, many sub-expressions are repeated; for instance, the join over the single triples resulting from the reformulation of triples* (t_2) *and* (t_3) *will appear for each of the* 188 *reformulations of triple* (t_1)*. Evaluating* q'_1 *on the* 100 *million triples LUBM dataset takes more than* ***6 seconds****.*

Alternatively, one could consider the equivalent query $q''_1 = (t_1)^{ref} \bowtie (t_2)^{ref} \bowtie (t_3)^{ref}$*, which joins the* CQ *to* UCQ *reformulation of each query's triple. In other terms,* q''_1 *first reformulates each triple (into, respectively, a union of* 188*,* 4*, and* 3 *queries), and then joins these unions. This query corresponds to the simple semi-conjunctive queries (*SCQ*) alternative proposed in [18]. While this avoids the repeated work, its performance is much worse: it takes about* ***1074 seconds*** *to evaluate.*

Let us now consider the following equivalent query $q'''_1 = (t_1, t_3)^{ref} \bowtie (t_2)^{ref}$ *where* t_1, t_2, t_3 *are the triples of the query* q_1*. Evaluating* q'''_1 *in the same experimental setting*

	Joins of UCQs	#reformulations	exec.time (ms)
q_1'	$(t_1,t_2,t_3)^{ref}$	2,256	6,387
q_1''	$(t_1)^{ref} \bowtie (t_2)^{ref} \bowtie (t_3)^{ref}$	195	1,074,026
	$(t_1,t_2)^{ref} \bowtie (t_3)^{ref}$	755	1,968
	$(t_1)^{ref} \bowtie (t_2,t_3)^{ref}$	200	846,710
q_1'''	$(t_1,t_3)^{ref} \bowtie (t_2)^{ref}$	568	554
	$(t_1,t_2)^{ref} \bowtie (t_1,t_3)^{ref}$	1,316	2,734
	$(t_1,t_2)^{ref} \bowtie (t_2,t_3)^{ref}$	764	2,289
	$(t_1,t_3)^{ref} \bowtie (t_2,t_3)^{ref}$	576	588

Table 2: Sample reformulations of q_1.

Triple	#answers	#reformulations	#answers after reformulation
(t_1)	18,999,082	188	33,328,108
(t_2)	18,999,082	188	33,328,108
(t_3)	476	1	476
(t_4)	509	1	509
(t_5)	7,299,701	3	7,803,096
(t_6)	7,299,701	3	7,803,096

Table 3: Characteristics of the sample query q_2.

*takes **554 ms**, more than **10 times faster** than the initial reformulation. The performance improvement of q_1''' over q_1'' is due to the intelligent grouping of the triples t_1 and t_3 together. Such grouping of triples reduce the cardinality of the respective reformulated queries. Thus, $(t_1,t_3)^{ref}$ has 2,045 answers and 564 reformulations. Table 2 shows the number of reformulations and execution time for all the eight possible JUCQs.*

Example 2. *Let q_2 be the query:*

$q_2(x,u,y,v,z)$:-
- x rdf:type u, (t_1)
- y rdf:type v, (t_2)
- x *ub:mastersDegreeFrom* "$http://www.Univ532.edu$", (t_3)
- y *ub:doctoralDegreeFrom* "$http://www.Univ532.edu$", (t_4)
- x *ub:memberOf* z (t_5)
- y *ub:memberOf* z (t_6)

Statistics on the query triples, when evaluated over a 100 million triples LUBM dataset, appear in Table 3.

*The CQ to UCQ reformulation of q_2 leads to a query q_2' corresponding to a union of 318,096 CQs, which **could not be evaluated** in our experimental setting: this huge query could not even be parsed [5].*

*Now consider the query $q_2'' = (t_1)^{ref} \bowtie (t_2)^{ref} \bowtie (t_3)^{ref} \bowtie (t_4)^{ref} \bowtie (t_5)^{ref} \bowtie (t_6)^{ref}$, where $t_1,\ldots,t_6$ are the triples of q_2; this corresponds to the SCQ reformulation proposed in [18]. q_2'' is equivalent to q_2', and in our same experimental setting, it is evaluated in **229 seconds**. This is due to the large results of the (syntactically small) subqueries $(t_1)^{ref},\ldots,(t_6)^{ref}$ (especially the first two with 33,328,108 results each), which required some time to join.*

*Finally, consider the query $q_2''' = (t_1,t_3)^{ref} \bowtie (t_3,t_5)^{ref} \bowtie (t_2,t_4)^{ref} \bowtie (t_4,t_6)^{ref}$, also equivalent to q_2'. Evaluating q_2''' takes **524 ms**, more than **430 times faster** than q_2''. The performance advantage of q_2''' is due to intelligently grouping triples, so that the subquery corresponding to each triple group can be efficiently evaluated and returns results of manageable size. In particular, the largest-result query triples (t_1) and (t_2) had been grouped with (t_3) and (t_4) respectively, resulting in smaller intermediate results of 2,296 and 2,475 rows respectively, and improving the perfomance. Grouping triples (t_3) and (t_4) with the (t_5) and (t_6) respectively, yields analogous performance improvements.*

As the above examples shows, enlarging the query reformulation language from the state-of-the-art UCQs [7, 8, 9, 11, 15, 19] or of SCQs [18], to that of *joins* of UCQs (or JUCQs, in short), has a great performance improvement potential.

Query covering is a technique we introduced [5] for exploring a space of JUCQ reformulations of a given query. The idea is to *cover* a query q with (possibly overlapping) subqueries; for instance, $\{\{t_1,t_3\},\{t_3,t_5\},\{t_2,t_4\},\{t_4,t_6\}\}$ is a cover of our query q_2, corresponding exactly to the query q_2''' in Table 3, which has the shortest evaluation time.

As shown in [5], *each cover naturally leads to a query answering strategy*: reformulating each cover subquery using any CQ-to-UCQ algorithm, and joining the results of these reformulated queries, yields the answer to the original query.

Greedy cost-based cover selection (GCov). To select the cover leading to the most eficient evaluation, we rely on a *cost estimation function* c which, for a JUCQ q, returns the cost of evaluating it through an RDBMS storing the database. Function c may reflect any (combination of) query evaluation costs, such as I/O, CPU etc.; in [5] we computed c based on database textbook formulas.

Our *greedy cost-based cover search algorithm*, named GCov, starts with a cover where each atom is alone in a fragment, and adds an atom to a fragment (leading to a new cover) if the cost model suggests the new cover may lead to a more efficient query answering strategy. This (i) makes Ref feasible in cases when the reformulated queries built by previous reformulation algorithms simply fail, and (ii) strongly improves Ref performance in the other cases, as our experiments have shown [5] on three different RDBMSs.

5. DEMONSTRATION OUTLINE

Our demo [6] analyzes reformulation-based query answering, with a particular focus on performance and completeness.

A first dimension of the problem is the query reformulation strategy. Since UCQ and SCQ reformulations are JUCQ ones obtained from particular query covers, our demo **represents them by the corresponding covers**, which are well suited to a graphical visualization.

A second dimension is the data management platform. (i) We use **well-established RDBMSs**, namely **PostgreSQL** 9.3.2, **DB2** Express-C 10.5 and **MySQL** Server 5.6.20, on top of which queries can be answered using any cover: a fixed one (i.e., a UCQ or SCQ), a user-chosen one with the help of our GUI (a JUCQ), or a best one w.r.t. cost (a best performing JUCQ). (ii) We demonstrate the same alternatives on top of the state-of-the-art **RDF-3X** research prototype [17]. (iii) Our demo integrates the popular RDF platforms **Virtuoso** and **AllegroGraph** using their own (incomplete) Ref strategy. *These systems and reformulation strategies are representative of the state of the art for* Ref. In addition, we showcase a simple encoding of the RDF data, constraints and queries into Datalog programs to be evaluated by the **LogicBlox** engine. This can be viewed as another answering technique Dat, an alternative to Ref and Sat.

The third important aspect is (sub)query evaluation **costs**, which depends on the characteristics of data and constraints. We will rely on **real and synthetic RDF data sets**, such as French statistical (INSEE) and geographical (IGN) data, DBLP, and LUBM.

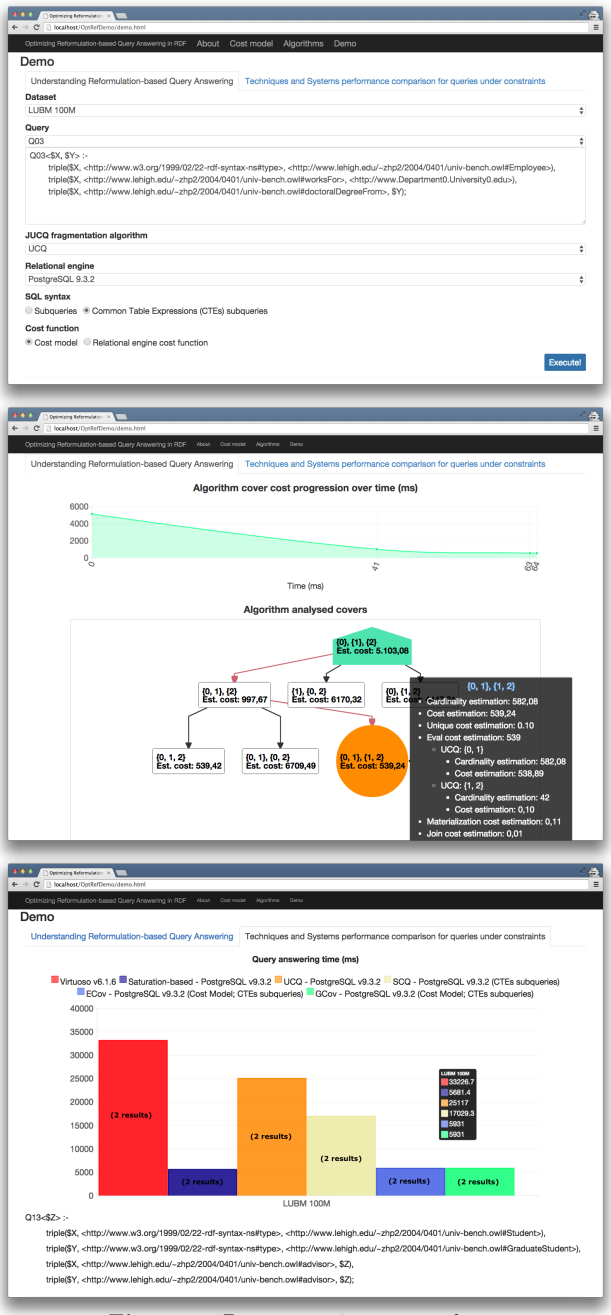

Figure 3: Demonstration screen shots.

The demo attendee experience is as follows. **1.** Pick an RDF graph (data and constraints), and visualize its statistics (value distributions for subject, property and object, for attribute pairs etc.). **2.** Select a query (upper part of Figure 3) and answer it through a chosen system and query cover, or through all the available systems, to compare their performance and completeness (bottom of Figure 3). **3.** Observe the evaluation runtime and inspect: the chosen query plan; cardinalities and costs of (sub)queries; and (if the cover was selected by `GCov`) the space of explored alternatives, and their estimated costs (center of Figure 3). **4.** Choose (from a pre-defined set) or propose modifications to the available RDF data and constraints, and re-run steps **1.-3.** to see the impact on Ref performance (constraints and query modifications, in particular, may have a dramatic impact).

6. REFERENCES

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

[2] M. Arenas, C. Gutierrez, and J. Pérez. Foundations of RDF databases. In *Reasoning Web*, 2009.

[3] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The DL Handbook: Theory, Implem., and Applications*. Cambridge Univ. Press, 2003.

[4] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient RDF store over a relational database. In *SIGMOD*, 2013.

[5] D. Bursztyn, F. Goasdoué, and I. Manolescu. Optimizing reformulation-based query answering in RDF. In *EDBT*, 2015.

[6] D. Bursztyn, F. Goasdoué, and I. Manolescu. Reformulation-based query answering in RDF: alternatives and performance". In *VLDB*, 2015.

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

[8] G. D. Giacomo, D. Lembo, M. Lenzerini, A. Poggi, R. Rosati, M. Ruzzi, and D. Savo. MASTRO: A reasoner for effective ontology-based data access. In *ORE*, 2012.

[9] F. Goasdoué, I. Manolescu, and A. Roatiş. Efficient query answering against dynamic RDF databases. In *EDBT*, 2013.

[10] F. Goasdoué, I. Manolescu, and A. Roatiş. RDF data mgmt.: Reasoning on Web data (tutorial). In *ICDE*, 2015.

[11] G. Gottlob, G. Orsi, and A. Pieris. Query rewriting and optimization for ontological databases. *ACM TODS*, 39(3):25, 2014.

[12] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3), 2005.

[13] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *SIGMOD*, 2014.

[14] M. Hammoud, D. A. Rabbou, R. Nouri, S.-M.-R. Behehti, and S. Sakr. DREAM: Distributed RDF engine with adaptive query planner and minimal communication. In *PVLDB*, 2015.

[15] Z. Kaoudi, I. Miliaraki, and M. Koubarakis. RDFS reasoning and query answering on DHTs. In *ISWC*, 2008.

[16] D. Lanti, M. Rezk, G. Xiao, and D. Calvanese. The NPD benchmark: Reality check for OBDA systems. In *EDBT*, 2015.

[17] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDBJ*, 19(1), 2010.

[18] M. Thomazo. Compact rewriting for existential rules. *IJCAI*, 2013.

[19] J. Urbani, F. van Harmelen, S. Schlobach, and H. Bal. QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases. In *ISWC*, 2011.