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 poor reformulations, 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., [1, 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 Employees, it is reformulated into $Q'$ returning the union of all Employees and all Managers. 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

(c) 2015, Copyright is with the authors. Published in the Proceedings of the BDA 2015 Conference (September 29-October 2, 2015, Ile de Porquerolles, France). Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

http://linkeddatacatalog.dws.informatik.uni-mannheim.de/state/luwclouddiagram.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 (RDBMSs, 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 [2, 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, e.g., 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 [5]. Existing techniques apply to the Description Logics (DLs) fragment of RDF, the conjunctive subset of SPARQL and extensions thereof [2, 5, 15, 19], including the “database fragment” of RDF we introduced in [2], 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 [19], when translated to the RDF setting, reformulates a conjunctive query into a union 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 JUCQs in the sequel). UCQ and SCQ reformulations are each just a point in this space. The evaluation performance of distinct JUCQs 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 [18]; the resulting reformulated query may still be large, thus hard to evaluate.

### 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 doi1; its author (a blank node $b_1$ related to the author name), title and date of publication.

$$G = \{\text{doi1 rdf:type Book, doi1 writtenBy } b_1, \text{doi1 hasTitle "El Aleph"}, \text{doi1 hasPublisher "J. L. Borges"}, \text{doi1 publishedIn "1949"}\}$$

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 $\vdash_{\text{RDF}}$ 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 $G \vdash_{\text{RDF}} s$ 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.
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 $q(x) := t_1, \ldots, t_n$, where $\{t_1, \ldots, t_n\}$ is a BGP; the query head variables $x$ are called distinguished variables, and are a subset of the variables in $t_1, \ldots, t_n$; for boolean queries $x$ is empty. The head of $q$ is $q(x)$, its body is $t_1, \ldots, t_n$: $x, y, z, \ldots$ denote variables.

Query answering. The evaluation of a CQ $q$ against $G$ has access only to $G$’s explicit triples, thus may lead to an incomplete answer. The (complete) answer of $q$ against $G$ is obtained by the evaluation of $q$ against $G^\infty$. For instance, the query below asks for the names of authors of books somehow obtained by the evaluation of $q$. The (complete) answer of $q$ is $\{\text{J. L. Borges}\}$. Note that evaluating $q$ only against $G$ leads to the empty answer, which is obviously incomplete.

3.3 Reformulation-based query answering

The database (DB) fragment of RDF [6] 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 $G$, 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-saturation) $q^{ref}(db) = q^{ref}(db)$.

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:

$$q_1(x, y) := x \text{ rdf:type } y, x \text{ ubdegree} \text{From } \text{ "http://www.Univ532.edu"}, t_1$$

$$t_2 \text{ ubmemberOf} \text{From } \text{ "http://www.Dept1.Univ7.edu"}, t_3$$

Table 1 gives some intuition on the difficulty of answering $q_1$ over an 10$^6$ triples LUBM [12] benchmark dataset, evaluated through PostgreSQL 9.3.2 on an 8-core Intel Xeon (E5506) 2.13 GHz machine with 16GB RAM, using Mandriva Linux release 2010.0 (Official).

<table>
<thead>
<tr>
<th>Triple</th>
<th>#answers</th>
<th>#reformulations</th>
<th>#answers after reformulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(t_1)$</td>
<td>18,999,082</td>
<td>188</td>
<td>33,328,108</td>
</tr>
<tr>
<td>$(t_2)$</td>
<td>0</td>
<td>4</td>
<td>3,223</td>
</tr>
<tr>
<td>$(t_3)$</td>
<td>390</td>
<td>3</td>
<td>685</td>
</tr>
</tbody>
</table>

Table 1: Characteristics of the sample query $q_1$. 

The state-of-the-art RDF technique reformulating a CQ into an UCQ computes the answer to $q_1$, by reformulating the 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 1, where all the triples $t_1, t_2, t_3$ are reformulated together by a CQ to UCQ reformulation algorithm denoted $\text{UCQ}^{\text{ref}}$. 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 $q'' = (t_1)^{ref} \bowtie (t_2)^{ref} \bowtie (t_3)^{ref}$, which joins the CQ to UCQ reformulation of each query’s triple. In other terms, $q''$ 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 [17]. 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 $q''' = (t_1, t_2)^{ref} \bowtie (t_2, t_3)^{ref}$ where $t_1, t_2, t_3$ are the triples of the query $q_1$. Evaluating $q'''$ in the same experimental setting
corresponding to a union of \(\{q_1, q_2, q_5, q_6\}\) in our experimental setting: this huge query can be efficiently evaluated and returns results of manageable size. In particular, the largest-result query \((t_1, t_2)\) had been grouped with \((t_3, t_4)\) respectively, resulting in smaller intermediate results of 2,296 and 2,475 rows respectively, and improving the performance. Grouping triples \((t_2, t_3)\) and \((t_4)\) with the \((t_1)\) 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, \(\{(t_1, t_3), \{t_1, t_5\}, \{t_2, t_4\}, \{t_4, t_6\}\}\) is a cover of our query \(q_1\), corresponding exactly to the query \(q''_1\) 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 efficient 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 \(\text{Ref}\) feasible in cases when the reformulated queries built by previous reformulation algorithms simply fail, and (ii) strongly improves \(\text{Ref}\) performance in the other cases, as our experiments have shown [5] on three different RDBMSs.

## 5. DEMONSTRATION OUTLINE

Our demo [5] 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) \(\text{Ref}\) strategy. *These systems and reformulation strategies are representative of the state of the art for \(\text{Ref}\).* 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 \(\text{Ref}\) 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.
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