Dynamic Querying of Mass-Storage RDF Data with Rule-Based Entailment Regimes*

Giovambattista Ianni[1], Thomas Krennwallner[2], Alessandra Martello[1], and Axel Polleres[3]

[1] Dipartimento di Matematica, Università della Calabria, I-87036 Rende (CS), Italy
{ianni,a.martello}@mat.unical.it
[2] Institut für Informationssysteme 184/3, Technische Universität Wien, Austria
tkren@kr.tuwien.ac.at
[3] Digital Enterprise Research Institute, National University of Ireland, Galway
axel.polleres@deri.org

Abstract. RDF Schema (RDFS) as a lightweight ontology language is gaining popularity and, consequently, tools for scalable RDFS inference and querying are needed. SPARQL has become recently a W3C standard for querying RDF data, but it mostly provides means for querying simple RDF graphs only, whereas querying with respect to RDFS or other entailment regimes is left outside the current specification. In this paper, we show that SPARQL faces certain unwanted ramifications when querying ontologies in conjunction with RDF datasets that comprise multiple named graphs, and we provide an extension for SPARQL that remedies these effects. Moreover, since RDFS inference has a close relationship with logic rules, we generalize our approach to select a custom ruleset for specifying inferences to be taken into account in a SPARQL query. We show that our extensions are technically feasible by providing benchmark results for RDFS querying in our prototype system GiaBATA, which uses Datalog coupled with a persistent Relational Database as a back-end for implementing SPARQL with dynamic rule-based inference. By employing different optimization techniques like magic set rewriting our system remains competitive with state-of-the-art RDFS querying systems.

1 Introduction

Thanks to initiatives such as DBPedia or the Linked Open Data project,[4] a huge amount of machine-readable RDF [1] data is available, accompanying pervasive ontologies describing this data such as FOAF [2], SIOC [3], or YAGO [4].

A vast amount of Semantic Web data uses rather small and lightweight ontologies that can be dealt with rule-based RDFS and OWL reasoning [5–7], in contrast to the full power of expressive description logic reasoning. However, even if many practical use cases do not require complete reasoning on the terminological level provided by DL-reasoners, the following tasks become of utter importance. First, a Semantic Web system should be able to handle and evaluate (possibly complex) queries on large amounts of RDF instance data. Second, it should be able to take into account implicit knowledge found by ontological inferences as well as by additional custom rules involving built-ins

* This work has been partially supported by the Italian Research Ministry (MIUR) project Interlink II04CG8AGG, the Austrian Science Fund (FWF) project P20841, by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2).

[4] http://dbpedia.org/ and http://linkeddata.org/

or even nonmonotonicity. The latter features are necessary, e.g., for modeling complex mappings [8] between different RDF vocabularies. As a third point, joining the first and the second task, if we want the Semantic Web to be a solution to – as Ora Lassila formulated it – *those problems and situations that we are yet to define*,[5] we need triple stores that allow dynamic querying of different data graphs, ontologies, and (mapping) rules harvested from the Web. The notion of *dynamic* querying is in opposition to *static* querying, meaning that the same dataset, depending on context, reference ontology and entailment regime, might give different answers to the same query. Indeed, there are many situations in which the dataset at hand and its supporting class hierarchy cannot be assumed to be known upfront: think of distributed querying of remotely exported RDF data.

Concerning the first point, traditional RDF processors like Jena (using the default configuration) are designed for handling large RDF graphs in memory, thus reaching their limits very early when dealing with large graphs retrieved from the Web. Current RDF Stores, such as YARS [9], Sesame, Jena TDB, ThreeStore, AllegroGraph, or OpenLink Virtuoso[6] provide roughly the same functionality as traditional relational database systems do for relational data. They offer query facilities and allow to import large amounts of RDF data into their persistent storage, and typically support SPARQL [10], the W3C standard RDF query language. SPARQL has the same expressive power as non-recursive Datalog [11, 12] and includes a set of built-in predicates in so called **`filter`** expressions.

However, as for the second and third point, current RDF stores only offer limited support. OWL or RDF(S) inference, let alone custom rules, are typically fixed in combination with SPARQL querying (cf. Section 2). Usually, dynamically assigning different ontologies or rulesets to data for querying is neither supported by the SPARQL specification nor by existing systems. Use cases for such dynamic querying involve, e.g., querying data with different versions of ontologies or queries over data expressed in related ontologies adding custom mappings (using rules or "bridging" ontologies).

To this end, we propose an extension to SPARQL which caters for knowledge-intensive applications on top of Semantic Web data, combining SPARQL querying with dynamic, rule-based inference. In this framework, we overcome some of the above mentioned limitations of SPARQL and existing RDF stores. Moreover, our approach is easily extensible by allowing features such as aggregates and arbitrary built-in predicates to SPARQL (see [8, 14]) as well as the addition of custom inference and mapping rules. The contributions of our paper are summarized as follows:

- We introduce two additional language constructs to the normative SPARQL language. First, the directive **`using ontology`** for dynamically coupling a dataset with an arbitrary RDFS ontology, and second *extended dataset clauses*, which allow to specify datasets with named graphs in a flexible way. The **`using ruleset`** directive can be exploited for adding to the query at hand proper rulesets which might used for a variety of applications such as encoding mappings between entities, or encoding custom entailment rules, such as RDFS or different rule-based OWL fragments.
- We present the GiaBATA system [15], which demonstrates how the above extensions can be implemented on a middle-ware layer translating SPARQL to Datalog and SQL. Namely, the system is based on known translations of SPARQL to Datalog rules. Arbitrary, possibly recursive rules can be added flexibly to model arbitrary ontological inference regimes,

[5] `http://www.lassila.org/publications/2006/SCAI-2006-keynote.pdf`

[6] See `http://openrdf.org/`, `http://jena.hpl.hp.com/wiki/TDB/`, `http://threestore.sf.net/`, `http://agraph.franz.com/allegrograph/`, `http://openlinksw.com/virtuoso/`, respectively.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix rel: <http://purl.org/vocab/relationship/>.
...
rel:friendOf rdfs:subPropertyOf foaf:knows.
foaf:knows rdfs:domain foaf:Person.
foaf:knows rdfs:range foaf:Person.
foaf:homepage rdf:type owl:inverseFunctionalProperty.
...
```

(a) Graph G_M (<http://example.org/myOnt.rdfs>), a combination of the FOAF&Relationship ontologies.

```
<http://bob.org#me> foaf:name "Bob"; a foaf:Person;
  foaf:homepage <http://bob.org/home.html>;
  rel:friendOf [ foaf:name "Alice";
                 rdfs:seeAlso <http://alice.org> ].
```

(b) Graph G_B (<http://bob.org>)

```
<http://alice.org#me> foaf:name "Alice"; a foaf:Person;
rel:friendOf [ foaf:name "Charles" ],
             [ foaf:name "Bob";
               foaf:homepage <http://bob.org/home.html> ].
```

(c) Graph G_A (<http://alice.org>)

Fig. 1: An ontology and two data graphs

vocabulary mappings, or alike. The resulting program is compiled to SQL where possible, such that only the recursive parts are evaluated by a native Datalog implementation. This hybrid approach allows to benefit from efficient algorithms of deductive database systems for custom rule evaluation, and native features such as query plan optimization techniques or rich built-in functions (which are for instance needed to implement complex **filter** expressions in SPARQL) of common database systems.

• We compare our GiaBATA prototype to well-known RDF(S) systems and provide experimental results for the LUBM [16] benchmark. Our approach proves to be competitive on both RDF and dynamic RDFS querying without the need to pre-materialize inferences.

In the remainder of this paper we first introduce SPARQL along with RDF(S) and partial OWL inference by means of some motivating example queries which existing systems partially cannot deal in a reasonably manner in Section 2. Section 3 sketches how the SPARQL language can be enhanced with custom ruleset specifications and arbitrary graph merging specifications. We then briefly introduce our approach to translate SPARQL rules to Datalog in Section 4, and how this is applied to a persistent storage system. We evaluate our approach with respect to existing RDF stores in Section 5, and then conclusions are drawn in Section 6.

2 SPARQL and some Motivating Examples

Similar in spirit to structured query languages like SQL, which allow to extract, combine and filter data from relational database tables, SPARQL allows to extract, combine and filter data from RDF graphs. The semantics and implementation of SPARQL involves, compared to SQL, several peculiarities, which we do not focus on in this paper, cf. [10, 18, 11, 19] for details. Instead, let us just start right away with some illustrating example motivating our proposed extensions of SPARQL; we assume two data graphs describing data about our well-known friends Bob and Alice shown in Fig. 1(b)+(c). Both graphs refer to terms in a combined ontology defining the FOAF and Relationship[7] vocabularies, see Fig. 1(a) for an excerpt.

On this data the SPARQL query (1) intends to extract names of persons mentioned in those graphs that belong to friends of Bob. We assume that, by means of `rdfs:seeAlso` statements, Bob provides links to the graphs associated to the persons he is friend with.

```
select ?N from <http://example.org/myOnt.rdfs>
          from <http://bob.org>
          from named <http://alice.org>                          (1)
where { <http://bob.org#me> foaf:knows ?X . ?X rdfs:seeAlso ?G .
        graph ?G { ?P rdf:type foaf:Person; foaf:name ?N } }
```

[7] http://vocab.org/relationship/

Here, the **from** and **from named** clauses specify an RDF dataset. In general, the *dataset* $DS = (G, N)$ of a SPARQL query is defined by (i) a *default graph* G obtained by the RDF merge [20] of all graphs mentioned in **from** clauses, and (ii) a set $N = \{(u_1, G_1), \ldots, (u_k, G_k)\}$ of *named graphs*, where each pair (u_i, G_i) consists of an IRI u_i, given in a **from named** clause, paired with its corresponding graph G_i. For instance, the dataset of query (1) would be $DS_1 = (\ G_M \uplus G_B, \{($`<http://alice.org>`$, G_A)\})$, where $\uplus$ denotes merging of graphs according to the normative specifications.

Now, let us have a look at the answers to query (1). Answers to SPARQL **select** queries are defined in terms of multisets of partial variable substitutions. In fact the answer to query (1) is empty when – as typical for current SPARQL engines – only simple RDF entailment is taken into account, and query answering then boils down to simple graph matching. Since neither of the graphs in the default graph contain any triple matching the pattern `<http://bob.org#me> foaf:knows ?X` in the **where** clause, the result of (1) is empty. When taking subproperty inference by the statements of the ontology in G_M into account, however, one would expect to obtain three substitutions for the variable ?N: {?N/`"Alice"`, ?N/`"Bob"`, ?N/`"Charles"`}. We will explain in the following why this is not the case in standard SPARQL.

In order to obtain the expected answer, firstly SPARQL's basic graph pattern matching needs to be extended, see [10, Section 12.6]. In theory, this means that the graph patterns in the **where** clause needs to be matched against an enlarged version of the original graphs in the dataset (which we will call the *deductive closure* $Cl(\cdot)$) of a given entailment regime. Generic extensions for SPARQL to entailment regimes other than simple RDF entailment are still an open research problem,[8] due to various problems: (i) for (non-simple) RDF entailment regimes, such as full RDFS entailment, $Cl(G)$ is infinite, and thus SPARQL queries over an empty graph G might already have infinite answers, and (ii) it is not yet clear which should be the intuitive answers to queries over inconsistent graphs, e.g. in OWL entailment, etc. In fact, SPARQL restricts extensions of basic graph pattern matching to retain finite answers. Not surprisingly, many existing implementations implement finite approximations of higher entailment regimes such as RDFS and OWL [6, 5, 21]. E.g., the RDF Semantics document [20] contains an informative set of entailment rules, a subset of which (such as the one presented in Section 3.2 below) is implemented by most available RDF stores. These rule-based approximations, which we focus on in this paper, are typically expressible by means of Datalog-style rules. These latter model how to infer a finite closure of a given RDF graph that covers sound but not necessarily complete RDF(S) and OWL inferences. It is worth noting that Rule-based entailment can be implemented in different ways: rules could be either dynamically evaluated upon query time, or the closure wrt. ruleset $\mathcal{R}$, $Cl_{\mathcal{R}}(G)$, could be materialized when graph G is loaded into a store. Materialization of inferred triples at loading time allows faster query responses, yet it has drawbacks: it is time and space expensive and it has to be performed once and statically. In this setting, it must be decided upfront

(a) which ontology should be taken into account for which data graph, and
(b) to which graph(s) the inferred triples "belong", which particularly complicates the querying of named graphs.

As for exemplifying (a), assume that a user agent wants to issue another query on graph G_B with only the FOAF ontology in mind, since she does not trust the Relationship ontology. In the realm of FOAF alone, `rel:friendOf` has nothing to deal with

[8] For details, cf. `http://www.polleres.net/sparqltutorial/`, Unit 5b.

foaf:knows. However, when materializing all inferences upon loading G_M and G_B into the store, bob:me foaf:knows _:a would be inferred from $G_M \uplus G_B$ and would contribute to such a different query. Current RDF stores prevent to dynamically parameterize inference with an ontology of choice at query time, since indeed typically all inferences are computed upon loading time *once and for all*.

As for (b), queries upon datasets including named graphs are even more problematic. Query (1) uses G_B in order to find the IRI identifiers for persons that Bob knows by following rdfs:seeAlso links and looks for persons and their names in the *named* RDF graphs found at these links. Even if rule-based inference was supported, the answer to query (1) over dataset DS_1 is just {?N/"Alice"}, as "Alice" is the only (explicitly) asserted foaf:Person in G_A. Subproperty, domain and range inferences over the G_M ontology do not propagate to G_A, since G_M is normatively prescribed to be merged into the default graph, but not into the named graph. Thus, there is no way to infer that "Bob" and "Charles" are actually names of foaf:Persons within the named graph G_A. Indeed, SPARQL does not allow to merge, on demand, graphs into the named graphs, thus there is no way of combining G_M with the named graph G_A.

To remedy these deficiencies, we suggest an extension of the SPARQL syntax, in order to allow the specification of datasets more flexibly: it is possible to group graphs to be merged in parentheses in **from** and **from named** clauses. The modified query, obtaining a dataset $DS_2 = (\, G_M \uplus G_B, \{(\texttt{http://alice.org}, G_M \uplus G_A)\})$ looks as follows:

```
select ?N
  from (<http://example.org/myOnt.rdfs> <http://bob.org/>)
  from named <http://alice.org>
     (<http://example.org/myOnt.rdfs> <http://alice.org/>)
where { bob:me foaf:knows ?X . ?X rdfs:seeAlso ?G .
        graph ?G { ?X foaf:name ?N . ?X a foaf:Person . } }
```
(2)

For ontologies which should apply to the whole query, i.e., graphs to be merged into the default graph as well as any specified named graph, we suggest a more convenient shortcut notation by adding the keyword **using ontology** in the SPARQL syntax:

```
select ?N
  using ontology <http://example.org/myOnt.rdfs>
  from <http://bob.org/>
  from named <http://alice.org/>
where { bob:me foaf:knows ?X . ?X foaf:seeAlso ?G .
        graph ?G { ?X foaf:name ?N . ?X a foaf:Person. } }
```
(3)

Hence, the **using ontology** construct allows for coupling the entire given dataset with the terminological knowledge in the myOnt data schema. As our investigation of currently available RDF stores (see Section 5) shows, none of these systems easily allow to merge ontologies into named graphs or to dynamically specify the dataset of choice.

In addition to parameterizing queries with ontologies in the dataset clauses, we also allow to parameterize the ruleset which models the entailment regime at hand. Per default, our framework supports a standard ruleset that "emulates" (a finite subset of) the RDFS semantics. This standard ruleset is outlined in Section 3 below. Alternatively, different rule-based entailment regimes, e.g., rulesets covering parts of the OWL semantics á la ter Horst [5], de Bruijn [22, Section 9.3], OWL2 RL [17] or other custom rulesets can be referenced with the **using ruleset** keyword. For instance, the following query returns the solution {?X/<http://alice.org#me>, ?Y/<http://bob.org#me>}, by doing equality reasoning over inverse functional properties such as foaf:homepage when the

FOAF ontology is being considered:

```
select ?X ?Y
 using ontology <http://example.org/myOnt.rdfs>
 using ruleset rdfs
 using ruleset <http://www.example.com/owl-horst>                    (4)
 from <http://bob.org/>
 from <http://alice.org/>
where { ?X foaf:knows ?Y }
```

Query (4) uses the built-in RDFS rules for the usual subproperty inference, plus a ruleset implementing ter Horst's inference rules, which might be available at URL `http://www.example.com/owl-horst`. This ruleset contains the following additional rules, that will "equate" the blank node used in G_A for "Bob" with `<http://bob.org#me>`:[9]

```
?P rdf:type owl:iFP . ?S1 ?P ?O . ?S2 ?P ?O .  → ?S1 owl:sameAs ?S2.
?X owl:sameAs ?Y                               → ?Y owl:sameAs ?X.
?X ?P ?O . ?X owl:sameAs ?Y                    → ?Y ?P ?O.              (5)
?S ?X ?O . ?X owl:sameAs ?Y                    → ?S ?Y ?O.
?S ?P ?X . ?X owl:sameAs ?Y                    → ?S ?P ?Y.
```

3 A Framework for Using Ontologies and Rules in SPARQL

In the following, we will provide a formal framework for the SPARQL extensions outlined above. In a sense, the notion of *dynamic querying* is formalized in terms of the dependence of BGP pattern answers $[\![P]\!]^{\mathcal{O},\mathcal{R}}$ from a variable ontology $\mathcal{O}$ and ruleset $\mathcal{R}$. For our exposition, we rely on well-known definitions of RDF datasets and SPARQL. Due to space limitations, we restrict ourselves to the bare minimum and just highlight some standard notation used in this paper.

Preliminaries. Let I, B, and L denote pairwise disjoint infinite sets of IRIs, blank nodes, and RDF literals, respectively. A *term* is an element from $I \cup B \cup L$. An *RDF graph* G (or simply *graph*) is defined as a set of *triples* from $I \cup B \times I \cup B \times I \cup B \cup L$ (cf. [18, 12]); by $blank(G)$ we denote the set of blank nodes of G.[10]

A *blank node renaming* θ is a mapping $I \cup B \cup L \to I \cup B \cup L$. We denote by $t\theta$ the application of θ to a term t. If $t \in I \cup L$ then $t\theta = t$, and if $t \in B$ then $t\theta \in B$. If (s, p, o) is a triple then $(s, p, o)\theta$ is the triple $(s\theta, p\theta, o\theta)$. Given a graph G, we denote by $G\theta$ the set of all triples $\{t\theta \mid t \in G\}$. Let G and H be graphs. Let θ^G_H be an arbitrary blank node renaming such that $blank(G) \cap blank(H\theta^G_H) = \emptyset$. The *merge of* G *by* H, denoted $G \uplus H$, is defined as $G \cup H\theta^G_H$.

An *RDF dataset* $D = (G_0, N)$ is a pair consisting of exactly one unnamed graph, the so-called default graph G_0, and a set $N = \{\langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$ of named graphs, coupled with their identifying URIs. The following conditions hold: (i) each G_i $(0 \leq i \leq n)$ is a graph, (ii) each u_j $(1 \leq j \leq n)$ is from I, and (iii) for all $i \neq j$, $\langle u_i, G_i\rangle, \langle u_j, G_j\rangle \in N$ implies $u_i \neq u_j$ and $blank(G_i) \cap blank(G_j) = \emptyset$.

The syntax and semantics of SPARQL can now be defined as usual, cf. [10, 18, 12] for details. For the sake of this paper, we restrict ourselves to **select** queries as shown in the example queries (1)–(4) and just provide an overview of the necessary concepts. A query in SPARQL can be viewed as a tuple $Q = (V, D, P)$, where V is the set of variables mentioned in the **select** clause, D is an RDF dataset, defined by means of **from** and **from named** clauses, and P is a *graph pattern*, defined in the **where** clause.

[9] We use `owl:iFP` as shortcut for `owl:inverseFunctionalProperty`.

[10] Note that we allow *generalized RDF graphs* that may have blank nodes in property position.

Graph patterns are in the simplest case sets of RDF triples (s, p, o), where terms and variables from an infinite set of variables Var are allowed, also called *basic graph patterns (BGP)*. More complex graph patterns can be defined recursively, i.e., if P_1 and P_2 are graph patterns, $g \in I \cup Var$ and R is a filter expression, then P_1 **`optional`** P_2, P_1 **`union`** P_2, P_1 **`filter`** R, and **`graph`** $g\, P_1$ are graph patterns.

Graph pattern matching. Queries are evaluated by matching graph patterns against graphs in the dataset. In order to determine a query's solution, in the simplest case BGPs are matched against the *active graph* of the query, which is one particular graph in the dataset, identified as shown next.

Solutions of BGP matching consist of multisets of bindings for the variables mentioned in the pattern to terms in the active graph. Partial solutions of each subpattern are joined according to an algebra defining the **`optional`**, **`union`** and **`filter`** operators, cf. [10, 18, 12]. For what we are concerned with here, the most interesting operator though is the **`graph`** operator, since it changes the active graph. That is, the active graph is the default graph G_0 for any basic graph pattern not occurring within a **`graph`** sub pattern. However, in a subpattern **`graph`** $g\, P_1$, the pattern P_1 is matched against the named graph identified by g, if $g \in I$, and against any named graph u_i, if $g \in Var$, where the binding u_i is returned for variable g. According to [12], for a RDF dataset D and active graph G, we define $[\![P]\!]_G^D$ as the multiset of tuples constituting the *answer* to the graph pattern P. The solutions of a query $Q = (V, D, P)$ is the projection of $[\![P]\!]_G^D$ to the variables in V only.

3.1 SPARQL with Extended Datasets

What is important to note now is that, by the way how datasets are syntactically defined in SPARQL, the default graph G_0 can be obtained from merging a group of different source graphs, specified via several **`from`** clauses – as shown, e.g., in query (1) – whereas in each **`from named`** clause a single, separated, named graph is added to the dataset. That is, **`graph`** patterns will always be matched against a separate graph only. To generalize this towards dynamic construction of groups of merged named graphs, we introduce the notion of an *extended dataset*, which can be specified by enlarging the syntax of SPARQL with two additional dataset clauses:

- For $i, i_1, \ldots, i_m$ distinct IRIs ($m \geq 1$), the statement "**`from named`** $i(i_1 \ldots i_m)$" is called *extended dataset clause*. Intuitively, $i_1 \ldots i_m$ constitute a group of graphs to be merged: the merged graph is given i as identifying IRI.
- For $o \in I$ we call the statement "**`using ontology`** o" an *ontological dataset clause*. Intuitively, o stands for a graph that will merged with all graphs in a given query.

Extended RDF datasets are thus defined as follows. A *graph collection* $\mathcal{G}$ is a set of RDF graphs. An *extended RDF dataset* $\mathcal{D}$ is a pair $(\mathcal{G}_0, \{\langle u_1, \mathcal{G}_1\rangle, \ldots, \langle u_n, \mathcal{G}_n\rangle\})$ satisfying the following conditions: (i) each $\mathcal{G}_i$ is a nonempty graph collection (note that $\{\emptyset\}$ is a valid nonempty graph collection), (ii) each u_j is from I, and (iii) for all $i \neq j$, $\langle u_i, \mathcal{G}_i\rangle, \langle u_j, \mathcal{G}_j\rangle \in D$ implies $u_i \neq u_j$ and for $G \in \mathcal{G}_i$ and $H \in \mathcal{G}_j$, $blank(G) \cap blank(H) = \emptyset$. We denote $\mathcal{G}_0$ as $dg(\mathcal{D})$, the *default graph collection* of $\mathcal{D}$.

Let $\mathcal{D}$ and $\mathcal{O}$ be an extended dataset and a graph collection, resp. The ordinary RDF dataset obtained from $\mathcal{D}$ and $\mathcal{O}$, denoted $D(\mathcal{D}, \mathcal{O})$, is defined as

$$\left(\biguplus_{g \in dg(\mathcal{D})} g \uplus \biguplus_{o \in \mathcal{O}} o, \left\{ \langle u, \biguplus_{g \in \mathcal{G}} g \uplus \biguplus_{o \in \mathcal{O}} o \rangle \mid \langle u, \mathcal{G} \rangle \in \mathcal{D} \right\} \right) .$$

We can now define the semantics of extended and ontological dataset clauses as follows. Let F be a set of ordinary and extended dataset clauses, and O be a set of ontological dataset clauses. Let $graph(g)$ be the graph associated to the IRI g: the extended RDF dataset obtained from F, denoted $edataset(F)$, is composed of:

(1) $\mathcal{G}_0 = \{graph(g) \mid$ "**from** g" $\in F\}$. If there is no **from** clause, then $\mathcal{G}_0 = \emptyset$.
(2) A named graph collection $\langle u, \{graph(u)\}\rangle$ for each "**from named** u" in F.
(3) A named graph collection $\langle i, \{graph(i_1), \dots, graph(i_m)\}\rangle$ for each "**from named** $i(i_1 \dots i_m)$" in F.

The graph collection obtained from O, denoted $ocollection(O)$, is the set $\{graph(o) \mid$ "**using ontology** o" $\in O\}$. The ordinary dataset of O and F, denoted $dataset(F, O)$, is the set $D(edataset(F), ocollection(O))$.

Let $\mathcal{D}$ and $\mathcal{O}$ be as above. The *evaluation* of a graph pattern P over $\mathcal{D}$ and $\mathcal{O}$ having active graph collection $\mathcal{G}$, denoted $[\![P]\!]_{\mathcal{G}}^{\mathcal{D},\mathcal{O}}$, is the evaluation of P over $D(\mathcal{D}, \mathcal{O})$ having active graph $G = \biguplus_{g\in\mathcal{G}} g$, that is, $[\![P]\!]_{\mathcal{G}}^{\mathcal{D},\mathcal{O}} = [\![P]\!]_{G}^{D(\mathcal{D},\mathcal{O})}$.

Note that the semantics of extended datasets is defined in terms of ordinary RDF datasets. This allows to define the semantics of SPARQL with extended and ontological dataset clauses by means of the standard SPARQL semantics. Also note that our extension is conservative, i.e., the semantics coincides with the standard SPARQL semantics whenever no ontological clauses and extended dataset clauses are specified.

3.2 SPARQL with Arbitrary Rulesets

Extended dataset clauses give the possibility of merging arbitrary ontologies into any graph in the dataset. The second extension herein presented enables the possibility of dynamically deploying and specifying rule-based entailments regimes on a per query basis. To this end, we define a generic $\mathcal{R}$-entailment, that is RDF entailment associated to a parametric ruleset $\mathcal{R}$ which is taken into account when evaluating queries. For each such $\mathcal{R}$-entailment regime we straightforwardly extend BGP matching, in accordance with the conditions for such extensions as defined in [10, Section 12.6].

We define an *RDF inference rule* r as the pair $(\mathcal{A}nte, \mathcal{C}on)$, where the *antecedent* $\mathcal{A}nte$ and the *consequent* $\mathcal{C}on$ are basic graph patterns such that $\mathcal{V}(\mathcal{C}on)$ and $\mathcal{V}(\mathcal{A}nte)$ are non-empty, $\mathcal{V}(\mathcal{C}on) \subseteq \mathcal{V}(\mathcal{A}nte)$ and $\mathcal{C}on$ does not contain blank nodes.[11] As in Example (5) above, we typically write RDF inference rules as

$$\mathcal{A}nte \rightarrow \mathcal{C}on \ . \tag{6}$$

We call sets of inference rules *RDF inference rulesets*, or *rulesets* for short.

Rule Application and Closure. We define *RDF rule application* in terms of the immediate consequences of a rule r or a ruleset $\mathcal{R}$ on a graph G. Given a BGP P, we denote as $\mu(P)$ a pattern obtained by substituting variables in P with elements of $I \cup B \cup L$. Let r be a rule of the form (6) and G be a set of RDF triples, then:

$$T_r(G) = \{\mu(\mathcal{C}on) \mid \exists\mu \text{ such that } \mu(\mathcal{A}nte) \subseteq G\}.$$

[11] Unlike some other rule languages for RDF, the most prominent of which being CONSTRUCT statements in SPARQL itself, we forbid blank nodes; i.e., existential variables in rule consequents which require the "invention" of new blank nodes, typically causing termination issues.

Accordingly, let $T_{\mathcal{R}}(G) = \bigcup_{r\in\mathcal{R}} T_r(G)$. Also, let $G_0 = G$ and $G_{i+1} = G_i \cup T_{\mathcal{R}}(G_i)$ for $i \geq 0$. It can be easily shown that there exists the smallest n such that $G_{n+1} = G_n$; we call then $Cl_{\mathcal{R}}(G) = G_n$ the *closure* of G with respect to ruleset $\mathcal{R}$.

We can now further define *$\mathcal{R}$-entailment* between two graphs G_1 and G_2, written $G_1 \models_{\mathcal{R}} G_2$, as $Cl_{\mathcal{R}}(G_1) \models G_2$. Obviously for any finite graph G, $Cl_{\mathcal{R}}(G)$ is finite. In order to define the semantics of a SPARQL query wrt. *$\mathcal{R}$-entailment* we now extend graph pattern matching in $[\![P]\!]_G^D$ towards respecting $\mathcal{R}$.

Definition 1 (extended basic graph pattern matching for $\mathcal{R}$-entailment). *Let D be a dataset and G be an active graph. The* solution of a BGP P wrt. $\mathcal{R}$-entailment*, denoted $[\![P]\!]_G^{D,\mathcal{R}}$, is $[\![P]\!]_{Cl_{\mathcal{R}}(G)}^D$.*

The solution $[\![P]\!]_G^{D,\mathcal{R}}$ naturally extends to more complex patterns according to the SPARQL algebra. In the following we will assume that $[\![P]\!]_G^{D,\mathcal{R}}$ is used for graph pattern matching. Our extension of basic graph pattern matching is in accordance with the conditions for extending BGP matching in [10, Section 12.6]. Basically, these conditions say that any extension needs to guarantee finiteness of the answers, and defines some conditions about a "*scoping graph*." Intuitively, for our extension, the scoping graph is just equivalent to $Cl_{\mathcal{R}}(G)$. We refer to [10, Section 12.6] for the details.

To account for this generic SPARQL BGP matching extension parameterized by an RDF inference ruleset $\mathcal{R}_Q$ per SPARQL query Q, we introduce another novel language construct for SPARQL:

- For $r \in I$ we call "**`using ruleset`** r" a *ruleset clause*.

Analogously to IRIs denoting graphs, we now assume that an IRI $r \in I$ may not only refer to graphs but also to rulesets, and denote the corresponding ruleset by $ruleset(r)$. Each query Q may contain zero or more ruleset clauses, and we define the query ruleset $\mathcal{R}_Q = \bigcup_{r\in R} ruleset(r)$, where R is the set of all ruleset clauses in Q.

The definitions of solutions of a query and the evaluation of a pattern in this query on active graph G is now defined just as above, with the only difference that answer to a pattern P are given by $[\![P]\!]_G^{D,\mathcal{R}_Q}$.

We observe that whenever $\mathcal{R} = \emptyset$, then $\mathcal{R}$-entailment boils down to simple RDF entailment. Thus, a query without ruleset clauses will just be evaluated using standard BGP matching. In general, our extension preserve full backward compatibility.

Proposition 1. *For $\mathcal{R} = \emptyset$ and RDF graph G, $[\![P]\!]_G^{D,\mathcal{R}} = [\![P]\!]_G^D$.*

Analogously, one might use *$\mathcal{R}$-entailment* as the basis for RDFS entailment as follows. We consider here the ρDF fragment of RDFS entailment [6]. Let $\mathcal{R}_{RDFS}$ denote the ruleset corresponding to the minimal set of entailment rules (2)–(4) from [6]:

```
?P rdfs:subPropertyOf ?Q . ?Q rdfs:subPropertyOf ?R .  → ?P rdfs:subPropertyOf ?R.
?P rdfs:subPropertyOf ?Q . ?S ?P ?O .                  → ?S ?Q ?O.
?C rdfs:subClassOf ?D . ?D rdfs:subClassOf ?E .        → ?C rdfs:subClassOf ?E.
?C rdfs:subClassOf ?D . ?S rdf:type ?C .               → ?S rdf:type ?D.
?P rdfs:domain ?C . ?S ?P ?O .                         → ?S rdf:type ?C.
?P rdfs:range ?C . ?S ?P ?O .                          → ?O rdf:type ?C.
```

Since obviously $G \models_{RDFS} Cl_{\mathcal{R}_{RDFS}}(\mathcal{G})$ and hence $Cl_{\mathcal{R}_{RDFS}}(\mathcal{G})$ may be viewed as a finite approximation of RDFS-entailment, we can obtain a reasonable definition for defining a BGP matching extension for RDFS by simply defining $[\![P]\!]_G^{D,RDFS} = [\![P]\!]_G^{D,\mathcal{R}_{RDFS}}$. We allow the special ruleset clause **`using ruleset`** `rdfs` to conveniently refer to this

particular ruleset. Other rulesets may be published under a Web dereferenceable URI, e.g., using an appropriate RIF [23] syntax.

Note, eventually, that our rulesets consist of positive rules, and as such enjoy a natural monotonicity property.

Proposition 2. *For rulesets $\mathcal{R}$ and $\mathcal{R}'$, such that $\mathcal{R} \subseteq \mathcal{R}'$, and graph G_1 and G_2, if $G_1 \models_{\mathcal{R}} G_2$ then $G_1 \models_{\mathcal{R}'} G_2$.*

Entailment regimes modeled using rulesets can thus be enlarged without retracting former inferences. This for instance would allow to introduce tighter RDFS-entailment approximations by extending $\mathcal{R}_{RDFS}$ with further axioms, yet preserving inferred triples.

4 Translating SPARQL into Datalog and SQL

Our extensions have been implemented by reducing both queries, datasets and rulesets to a common ground which allows arbitrary interoperability between the three realms. This common ground is Datalog, wherein rulesets naturally fit and SPARQL queries can be reduced to. Subsequently, the resulting combined Datalog programs can be evaluated over an efficient SQL interface to an underlying relational DBMS that works as triple store.

From SPARQL to Datalog. A SPARQL query Q is transformed into a corresponding Datalog program D_Q. The principle is to break Q down to a series of Datalog rules, whose body is a conjunction of atoms encoding a graph pattern. D_Q is mostly a plain Datalog program in dlvhex [24] input format, i.e. Datalog with *external predicates* in the dlvhex language. These are explained along with a full account of the translation in [11, 19]. Main challenges in the transformation from SPARQL to Datalog are (i) faithful treatment of the semantics of joins over possibly unbound variables [11], (ii) the multiset semantics of SPARQL [19], and also (iii) the necessity of Skolemization of blank nodes in **`construct`** queries [8]. Treatment of **`optional`** statements is carried out by means of an appropriate encoding which exploits negation as failure. Special external predicates of dlvhex are used for supporting some features of the SPARQL language: in particular, importing RDF data is achieved using the external $\&rdf$ predicate, which can be seen as a built-in referring to external data. Moreover, SPARQL **`filter`** expressions are implemented using the dlvhex external $\&eval$ predicate in D_Q.

Let us illustrate this transformation step by an example: the following query A asking for persons who are not named "Alice" and optionally their email addresses

```
select * from <http://alice.org/>
 where { ?X a foaf:Person. ?X foaf:name ?N.                      (7)
         filter ( ?N != "Alice") optional { ?X foaf:mbox ?M } }
```

is translated to the program D_A as follows:

```
(r1) "triple"(S,P,O,default)     :- &rdf[ "alice.org" ](S,P,O).
(r2) answer1(X_N,X_X,default)    :- "triple"(X_X,"rdf:type","foaf:Person",default),
                                    "triple"(X_X,"foaf:name",X_N,default),
                                    &eval[" ?N != 'Alice' ","N", X_N ](true).
(r3) answer2(X_M,X_X,default)    :- "triple"(X_X,"foaf:mbox",X_M,default).
(r4) answer_b_join_1(X_M,X_N,X_X,default)  :- answer1(X_N,X_X,default),
                                              answer2(X_M,X_X,default).
(r5) answer_b_join_1(null,X_N,X_X,default) :- answer1(X_N,X_X,default),
                                              not answer2_prime(X_X,default).
(r6) answer2_prime(X_X,default) :- answer1(X_N,X_X,default),
                                   answer2(X_M,X_X,default).
(r7) answer(X_M,X_N,X_X)        :- answer_b_join1(X_M,X_N,X_X,default).
```

where the first rule `(r1)` computes the predicate `"triple"` taking values from the built-in predicate *&rdf*. This latter is generally used to import RDF statements from the specified URI. The following rules `(r2)` and `(r3)` compute the solutions for the filtered basic graph patterns { `?X a foaf:Person. ?X foaf:name ?N. filter (?N != "Alice")` } and { `?X foaf:mbox ?M` }. In particular, note here that the evaluation of **`filter`** expressions is "outsourced" to the built-in predicate *&eval*, which takes a filter expression and an encoding of variable bindings as arguments, and returns the evaluation value (`true`, `false` or `error`, following the SPARQL semantics). In order to emulate SPARQL's **`optional`** patterns a combination of join and set difference operation is used, which is established by rules `(r4)–(r6)`. Set difference is simulated by using both *null* values and *negation as failure*. According to the semantics of SPARQL, one has to particularly take care of variables which are joined and possibly unbound (i.e., set to the `null` value) in the course of this translation for the general case. Finally, the dedicated predicate *answer* in rule `(r7)` collects the answer substitutions for Q. D_Q might then be merged with additional rulesets whenever Q contains **`using ruleset`** clauses.

From Datalog to SQL. For this step we rely on the system DLV^{DB} [25] that implements Datalog under stable model semantics on top of a DBMS of choice. DLV^{DB} is able to translate Datalog programs in a corresponding SQL query plan to be issued to the underlying DBMS. RDF Datasets are simply stored in a database D, but the native dlvhex $\&rdf$ and $\&eval$ predicates in D_Q cannot be processed by DLV^{DB}directly over D. So, D_Q needs to be post-processed before it can be converted into suitable SQL statements.

Rule `(r1)` corresponds to loading persistent data into D, instead of loading triples via the $\&rdf$ built-in predicate. In practice, the predicate `"triple"` occurring in program D_A is directly associated to a database table TRIPLE in D. This operation is done off-line by a loader module which populates the TRIPLE table accordingly, while `(r1)` is removed from the program. The $\&eval$ predicate calls are recursively broken down into `WHERE` conditions in SQL statements, as sketched below when we discuss the implementation of **`filter`** statements.

After post-processing, we obtain a program D'_Q, which DLV^{DB}allows to be executed on a DBMS by translating it to corresponding SQL statements. D'_Q is coupled with a mapping file which defines the correspondences between predicate names appearing in D'_Q and corresponding table and view names stored in the DBMS D.

For instance, the rule `(r4)` of D_A, results in the following SQL statement issued to the RDBMS by DLV^{DB}:

```
INSERT INTO answer_b_join_1
 SELECT DISTINCT answer2_p2.a1, answer1_p1.a1, answer1_p1.a2, 'default'
 FROM answer1 answer1_p1, answer2 answer2_p2
 WHERE (answer1_p1.a2=answer2_p2.a2)
 AND (answer1_p1.a3='default')
 AND (answer2_p2.a3='default')
 EXCEPT (SELECT * FROM answer_b_join_1)
```

Whenever possible, the predicates for computing intermediate results such as `answer1`, `answer2`, `answer_b_join_1`, ..., are mapped to SQL views rather than materialized tables, enabling dynamic evaluation of predicate contents on the DBMS side.[12]

Schema rewriting. Our system allows for customizing schemes which triples are stored in. It is known and debated [26] that in choosing the data scheme of D several aspects have to be considered, which affect performance and scalability when handling large-scale

[12] For instance, recursive predicates require to be associated with permanent tables, while remaining predicates are normally associated to views.

RDF data. A widely adopted solution is to exploit a single table storing quadruples of form (s, p, o, c) where s, p, o and c are, respectively, the triple subject, predicate, object and context the triple belongs to. This straightforward representation is easily improved [27] by avoiding to store explicitly string values referring to URIs and literals. Instead, such values are replaced with a corresponding hash value.

Other approaches suggest alternative data structures, e.g., property tables [27, 26]. These aim at denormalizing RDF graphs by storing them in a flattened representation, trying to encode triples according to the hidden "schema" of RDF data. Similarly to a traditional relational schema, in this approach D contains a table per each known property name (and often also per class, splitting up the `rdf:type` table).

Our system gives sufficient flexibility in order to program different storage schemes: while on higher levels of abstraction data are accessible via the 4-ary *triple* predicate, a schema rewriter module is introduced in order to match D'_Q to the current database scheme. This module currently adapts D'_Q by replacing constant IRIs and literals with their corresponding hash value, and introducing further rules which translate answers, converting hash values back to their original string representation.

Magic sets. Notably, DLV^{DB} can post-process D'_Q using the magic sets technique, an optimization method well-known in the database field [28]. The optimized program mD'_Q tailors the data to be queried to an extent significantly smaller than the original D'_Q. The application of magic sets allows, e.g., to apply entailment rules $\mathcal{R}_{RDFS}$ only on triples which might affect the answer to Q, preventing thus the full computation and/or materialization of inferred data.

Implementation of `filter` statements. Evaluation of SPARQL **`filter`** statements is pushed down to the underlying database D by translating filter expressions to appropriate SQL views. This allows to dynamically evaluate filter expressions on the DBMS side. For instance, given a rule $r \in D_Q$ of the form

```
h(X,Y) :- b(X,Y), &eval[f_Y](bool).
```

where the `&eval` atom encodes the **`filter`** statement (*f_Y* representing the filter expression), then r is translated to

```
h(X,Y) :- b'(X,Y).
```

where `b'` is a fresh predicate associated via the mapping file to a database view. Such a view defines the SQL code to be used for the computation of *f_Y*, like

```
CREATE VIEW B' AS ( SELECT X,Y FROM B WHERE F_Y )
```

where *F_Y* is an appropriate translation of the SPARQL **`filter`** expression *f_Y* at hand to an SQL Boolean condition,[13] while B is the DBMS counterpart table of the predicate b.

5 Experiments

In order to illustrate that our approach is practically feasible, we present a quantitative performance comparison between our prototype system, GiaBATA, which implements the approach outlined before, and some state-of-the-art triple stores. The test were done on an Intel P4 3GHz machine with 1.5GB RAM under Linux 2.6.24. Let us briefly outline the main features and versions of the triple stores we used in our comparison.

[13] A version of this translation can be found in [29].

AllegroGraph works as a database and application framework for building Semantic Web applications. The system assures persistent storage and RDFS++ reasoning, a semantic extension including the RDF and RDFS constructs and some OWL constructs (`owl:sameAs`, `owl:inverseOf`, `owl:TransitiveProperty`, `owl:hasValue`). We tested the free Java edition of AllegroGraph 3.2 with its native persistence mechanism.[14]

ARQ is a query engine implementing SPARQL under the Jena framework.[15] It can be deployed on several persistent storage layers, like filesystem or RDBMS, and it includes a rule-based inference engine. Being based on the Jena library, it provides inferencing models and enables (incomplete) OWL reasoning. Also, the system comes with support for custom rules. We used **ARQ** 2.6 with RDBMS backend connected to PostgreSQL 8.3.

GiaBATA [15] is our prototype system implementing the SPARQL extensions described above. GiaBATA is based on a combination of the DLV^{DB} [25] and dlvhex [24] systems, and caters for persistent storage of both data and ontology graphs. The former system is a variant of DLV [13] with built-in database support. The latter is a solver for HEX-programs [24], which features an extensible plugin system which we used for developing a rewriter-plugin able to translate SPARQL queries to HEX-programs. The tests were done using development versions of the above systems connected to PostgreSQL 8.3.

Sesame is an open source RDF database with support for querying and reasoning.[16] In addition to its in-memory database engine it can be coupled with relational databases or deployed on top of file systems. Sesame supports RDFS inference and other entailment regimes such as OWL-Horst [5] by coupling with external reasoners. Sesame provides an infrastructure for defining custom inference rules. Our tests have been done using Sesame 2.3 with persistence support given by the native store.

First of all, it is worth noting that all systems allow persistent storage on RDBMS. All systems, with the exception of ours, implement also direct filesystem storage. All cover RDFS (actually, disregarding axiomatic triples) and partial or non-standard OWL fragments. Although all the systems feature some form of persistence, both reasoning and query evaluation are usually performed in main memory. All the systems, except AllegroGraph and ours, adopt a persistent materialization approach for inferring data.

All systems – along with basic inference – support named graph querying, but, with the exception of GiaBATA, combining both features results in incomplete behavior as described in Section 2. Inference is properly handled as long as the query ranges over the whole dataset, whereas it fails in case of querying explicit default or named graphs. That makes querying of named graphs involving inference impossible with standard systems.

For performance comparison we rely on the LUBM benchmark suite [16]. Our tests involve the test datasets LUBMn for $n \in \{1, 5, 10, 30\}$ with LUBM30 having roughly four million triples (exact numbers are reported in [16]). In order to test the additional performance cost of our extensions, we opted for showing how the performance figures change when queries which require RDFS entailment rules (LUBM Q4-Q7) are considered, w.r.t. queries in which rules do not have an impact (LUBM Q1-Q3, see Appendix of [16] for the SPARQL encodings of $Q1$–$Q7$). Experiments are enough for comparing performance trends, so we didn't consider at this stage larger instances of LUBM. Note that evaluation times include the data loading times. Indeed, while former performance benchmarks do

[14] System available at `http://agraph.franz.com/allegrograph/`.
[15] Distributed at `https://jena.svn.sourceforge.net/svnroot/jena/ARQ/`.
[16] System available at `http://www.openrdf.org/`.

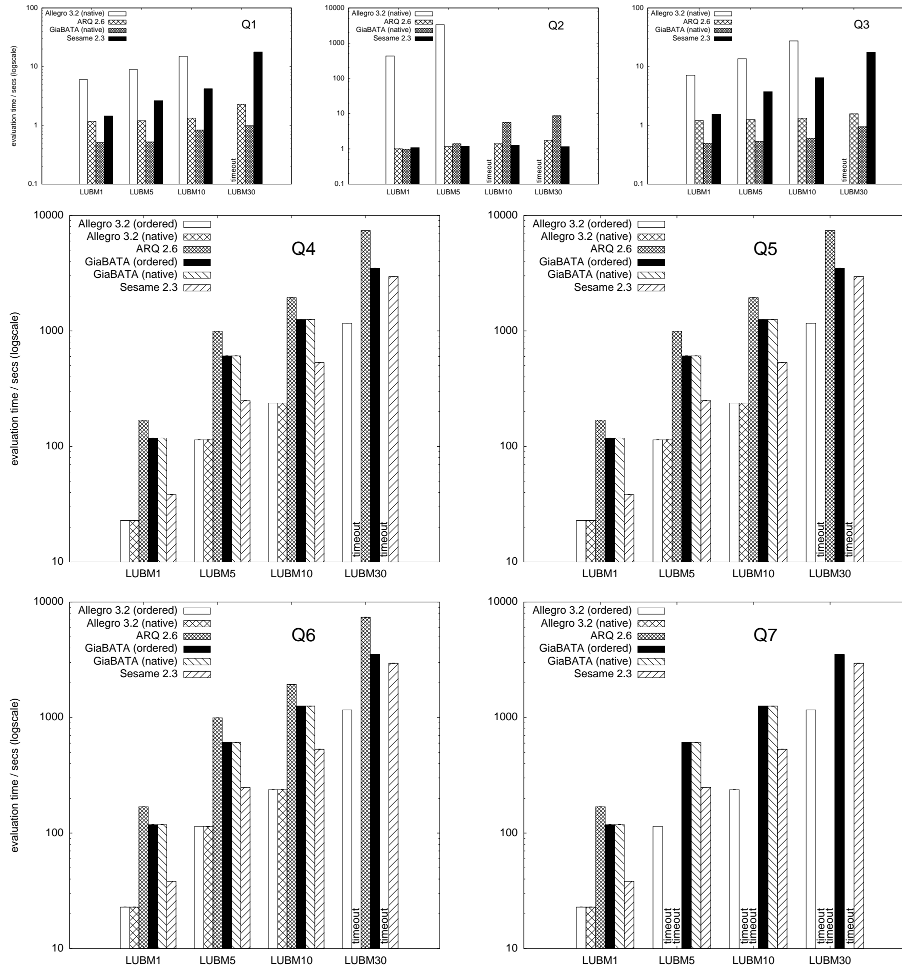

Fig. 2: Evaluation

not take this aspect in account, from the semantic point of view, pre-materialization-at-loading computes inferences needed for complete query answering under the entailment of choice. On further reflection, dynamic querying of RDFS moves inference from this materialization to the query step, which would result in an apparent advantage for systems that rely on pre-materialization for RDFS data. Also, the setting of this paper assumes materialization cannot be performed *una tantum*, since inferred information depends on the entailment regime of choice, and on the dataset at hand, on a *per query* basis. We set a 120min query timeout limit to all test runs.

Our test runs include the following system setup: (i) “Allegro (native)” and “Allegro (ordered)” (ii) “ARQ”; (iii) “GiaBATA (native)” and “GiaBATA (ordered)”; and (iv) “Sesame”. For (i) and (iii), which apply dynamic inference mechanisms, we use “(native)” and “(ordered)” to distinguish between executions of queries in LUBM’s native ordering and in a optimized reordered version, respectively. The GiaBATA test runs both

use Magic Sets optimization. To appreciate the cost of RDFS reasoning for queries $Q4$–$Q7$, the test runs for (i)–(iv) also include the loading time of the datasets, i.e., the time needed in order to perform RDFS data materialization or to simply store the raw RDF data.

The detailed outcome of the test results are summarized in Fig. 2. For the RDF test queries $Q1$–$Q3$, GiaBATA is able to compete for $Q1$ and $Q3$. The systems ARQ and Sesame turned out to be competitive for $Q2$ by having the best query response times, while Allegro (native) scored worst. For queries involving inference ($Q4$–$Q7$) Allegro shows better results. Interestingly, systems applying dynamic inference, namely Allegro and GiaBATA, query pattern reordering plays a crucial role in preserving performance and in assuring scalability; without reordering the queries simply timeout. In particular, Allegro is well-suited for queries ranging over several properties of a single class, whereas if the number of classes and properties increases ($Q7$), GiaBATA exhibits better scalability. Finally, a further distinction between systems relying on DBMS support and systems using native structures is disregarded, and since figures (in logarithmic scale) depict overall loading and querying time, this penalizes in specific cases those systems that use a DBMS.

6 Future Work and Conclusion

We presented a framework for dynamic querying of RDFS data, which extends SPARQL by two language constructs: **`using ontology`** and **`using ruleset`**. The former is geared towards dynamically creating the dataset, whereas the latter adapts the entailment regime of the query. We have shown that our extension conservatively extends the standard SPARQL language and that by selecting appropriate rules in **`using ruleset`**, we may choose varying rule-based entailment regimes at query-time. We illustrated how such extended SPARQL queries can be translated to Datalog and SQL, thus providing entry points for implementation and well-known optimization techniques. Our initial experiments have shown that although dynamic querying does more computation at query-time, it is still competitive for use cases that need on-the-fly construction of datasets and entailment regimes. Especially here, query optimization techniques play a crucial role, and our results suggest to focus further research in this direction. Furthermore, we aim at conducting a proper computational analysis as it has been done for Hypothetical datalog [30], in which truth of atoms is conditioned by hypothetical additions to the dataset at hand. Likewise, our framework allows to add ontological knowledge and rules to datasets before querying: note however that, in the spirit of [31], our framework allows for hypotheses (also called "premises") on a per query basis rather than a per atom basis.

References

1. Klyne, G., Carroll, J.J. (eds.): Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Rec. (February 2004)
2. Brickley, D., Miller, L.: FOAF Vocabulary Specification 0.91 (2007) `http://xmlns.com/foaf/spec/`.
3. Bojārs, U., Breslin, J.G., Berrueta, D., Brickley, D., Decker, S., Fernández, S., Görn, C., Harth, A., Heath, T., Idehen, K., Kjernsmo, K., Miles, A., Passant, A., Polleres, A., Polo, L., Sintek, M.: SIOC Core Ontology Specification (June 2007) W3C member submission.
4. Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: A Core of Semantic Knowledge. In: WWW2007, ACM (2007)

5. ter Horst, H.J.: Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. J. Web Semant. **3**(2–3) (2005) 79–115
6. Muñoz, S., Pérez, J., Gutiérrez, C.: Minimal deductive systems for RDF. In: ESWC'07. Springer (2007) 53–67
7. Hogan, A., Harth, A., Polleres, A.: Scalable authoritative owl reasoning for the web. Int. J. Semant. Web Inf. Syst. **5**(2) (2009)
8. Polleres, A., Scharffe, F., Schindlauer, R.: SPARQL++ for mapping between RDF vocabularies. In: ODBASE'07. Springer (2007) 878–896
9. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: ISWC'07. Springer (2007) 211–224
10. Prud'hommeaux, E., Seaborne, A. (eds.): SPARQL Query Language for RDF. W3C Rec. (January 2008)
11. Polleres, A.: From SPARQL to rules (and back). In: WWW2007. ACM (2007) 787–796
12. Angles, R., Gutierrez, C.: The expressive power of SPARQL. In: ISWC'08. Springer (2008) 114–129
13. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Log. **7**(3) (2006) 499–562
14. Euzenat, J., Polleres, A., Scharffe, F.: Processing ontology alignments with SPARQL. In: OnAV'08 Workshop, CISIS'08, IEEE Computer Society (2008) 913–917
15. Ianni, G., Krennwallner, T., Martello, A., Polleres, A.: A Rule System for Querying Persistent RDFS Data. In : ESWC'09. Springer (2009) 857–862
16. Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. J. Web Semant. **3**(2–3) (2005) 158–182
17. Motik, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C. (eds.): OWL 2 Web Ontology Language Profiles W3C Cand. Rec. (June 2009)
18. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In: ISWC'06. Springer (2006) 30–43
19. Polleres, A., Schindlauer, R.: dlvhex-sparql: A SPARQL-compliant query engine based on dlvhex. In: ALPSWS2007. CEUR-WS (2007) 3–12
20. Hayes, P.: RDF semantics. W3C Rec. (February 2004).
21. Ianni, G., Martello, A., Panetta, C., Terracina, G.: Efficiently querying RDF(S) ontologies with answer set programming. J. Logic Comput. **19**(4) (2009) 671–695
22. de Bruijn, J.: Semantic Web Language Layering with Ontologies, Rules, and Meta-Modeling. PhD thesis, University of Innsbruck (2008)
23. Boley, H., Kifer, M.: RIF Basic Logic Dialect. W3C Working Draft (July 2009)
24. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Effective integration of declarative rules with external evaluations for semantic web reasoning. In: ESWC'06. Springer (2006) 273–287
25. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. Theory Pract. Log. Program. **8**(2) (2008) 129–165
26. Theoharis, Y., Christophides, V., Karvounarakis, G.: Benchmarking Database Representations of RDF/S Stores. In: ISWC'05. Springer (2005) 685–701
27. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: VLDB. ACM (2007) 411–422
28. Beeri, C., Ramakrishnan, R.: On the power of magic. J. Log. Program. **10**(3-4) (1991) 255–299
29. Lu, J., Cao, F., Ma, L., Yu, Y., Pan, Y.: An Effective SPARQL Support over Relational Databases. In: SWDB-ODBIS. (2007) 57–76
30. Bonner, A.J.: Hypothetical datalog: complexity and expressibility. Theor. Comp. Sci. **76**(1) (1990) 3–51
31. Gutiérrez, C., Hurtado, C.A., Mendelzon, A.O.: Foundations of semantic web databases. In: PODS 2004, ACM (2004) 95–106