Quest, an OWL 2 QL Reasoner for Ontology-Based Data Access

Mariano Rodríguez-Muro and Diego Calvanese

KRDB Research Centre for Knowledge and Data
Free University of Bozen-Bolzano, Bolzano, Italy
{rodriguez,calvanese}@inf.unibz.it

Abstract. Ontology Based Data Access (OBDA) has drawn considerable attention from the OWL and RDF communities. In OBDA, instance data is accessed by means of mappings, which state the relationship between the data in a data source (e.g., an RDBMSs) and the vocabulary of an ontology. In this paper we present Quest, a new system for OBDA focused on fast and efficient reasoning with large ontologies and large volumes of data. Quest provides SPARQL query answering with OWL 2 QL/RDFS entailments and can function as a traditional OWL reasoner/triple store, or as a mediator, located on-top of a legacy data source linked to the ontology by means of mappings. In such configuration all data remains in the data source and is only accessed at run-time. Quest uses query rewriting techniques as the inference mechanism in both modes. In this paper we describe the architecture of Quest, and the optimization techniques it currently implements.

1 Introduction

Nowadays, the use of OWL and RDFS ontologies is widespread. One common use case for this technology is the exploitation of the vocabulary and semantics of ontologies during query answering over data extracted from existing legacy sources. The ontology vocabulary allows us to integrate the data from heterogenous sources into a coherent, global view. At the same time, the terminological part of an ontology (the TBox) allows for obtaining richer answers by means of OWL and RDFS inference. This scenario is called Ontology Based Data Access (OBDA).

The purpose of the current paper is to introduce *Quest*, a new system for OBDA whose key service is SPARQL query answering under the OWL 2 QL or RDFS entailment regimes. The development of Quest focuses on efficiency in the presence of very large volumes of data and very large ontologies. To achieve this, Quest uses highly optimized query rewriting techniques and relies on RDBMSs for ABox storage and query execution. Moreover, the SQL queries generated by Quest are tuned with respect to the capabilities of SQL engines to allow to exploit their optimizations during query evaluation. The rest of the paper is organized as follows: In Section 2, we introduce the notion of *OBDA models* and *mappings*, which are central to Quest. In Section 3, we provide a description of the query answering process in Quest as well as the optimizations it implements. In Section 4, we discuss Quest in the context of related systems. We conclude in Section 5 with future directions for the system. Quest is part of the **-ontop-** framework for OBDA. Further information about Quest and **-ontop-** can be found in the project's homepage [9]. This paper is a short overview of the system, for a full description of all techniques mentioned here we refer to a technical report [11].

2 OBDA models in Quest

A key aspect of OBDA is that data is obtained from one or more legacy systems, e.g., RDBMSs, XML systems, etc. To use this data in a Semantic Web application a user needs to perform an ETL process, i.e., (E)xtract the data from the sources, (T)ransform it into an OWL ABox (or as RDF triples), and (L)oad it into a query answering system (i.e., a reasoner or triple store). This practice has several drawbacks. First, it puts the burden of ETL on the client's application code, which increases software complexity of the application, and hence, the cost of development and maintenance. In addition, this generates duplicated data that consumes resources and introduces the problem that, in the event of an update on the data source(s), the system becomes out of synch. But more importantly, the fact that the query answering system is aware neither of the provenance of the data, nor of the way in which the data was extracted from the sources is an important missed opportunity for optimization during inference and/or query answering.

Example 1. Let $\mathcal{T}$ be an OWL TBox composed of the following 6 axioms:

$$
\begin{array}{ll}
\mathtt{SubClassOf(CoronaryPatient, CardiacArrestPatient)}, & \mathtt{domain(age, Patient)}, \\
\mathtt{SubClassOf(CardiacArrestPatient, Patient)}, & \mathtt{domain(id, Patient)}, \\
\mathtt{SubClassOf(CardiacArrest, HearthCondition)}, & \mathtt{domain(name, Patient)}, \\
\mathtt{SubClassOf(CardiacArrestPatient,} & \\
\quad \mathtt{ObjectSomeValuesFrom(affectedBy, CardiacArrest))} &
\end{array}
$$

and let our data source be a relational DB composed of two tables, *patient* and *condition*, organized as follows:

patient =

id (PK)	*name*	*age*
'x22'	'John'	56

condition =

id (PK)	*cond*
'x22'	22

such that *id* is a primary key over the patient table and, intuitively, the first 2 columns allow us to generate data for the `Patient` class and the data properties `id`, `name` and `age`; moreover, we know that for any row in the *condition* table, if $cond = 22$ then the corresponding *id* identifies a patient that had a cardiac arrest. If a system constructs an ABox using this specification, it might produce 5 ABox assertions, e.g., `Patient(x22)`, `id(x22, 'x22')`, `name(x22, 'John')`, `age(x22, 56)`, and `CardiacArrestPatient(x22)`.

More importantly, it is not possible to infer new ground assertions from this data and the TBox of the system, since all possible ground inferences are already explicit in the ABox. However, if we load these TBox and ABox into an inference engine, the system will inevitably generate redundant inferences, e.g., due to the `domain(name, Patient)` axiom, a system based on forward chaining would at least *attempt* to generate one assertion of the form `Patient(x22)` for each assertion of the form `name(x22, 'John')`. An optimized system would discard the redundant data, however, the inference would have been already done. ■

These redundant inferences, have a very high cost when the TBoxes and/or data are large, hindering dramatically the performance of inference at load time, or during query answering. More importantly, if the system has access to the source and information about the data extraction procedure, it would be possible to avoid redundancy.

Quest addresses the aforementioned issue by considering the formalization of the relationship between source and ontology as a first-class citizen in the system's lifetime. We call this formalization an *OBDA model*. The OBDA model is composed of a *data source definition* and a set of *mapping axioms*. The former provides the information that a system requires to access the data source, and the latter provides a formal specification of the relationship between the data in the *data source* and the vocabulary of an OWL 2 QL/RDFS ontology. Following [7], a mapping axiom in Quest is defined as a pair of an SQL query and an *ABox assertion template*. An assertion template is a set of RDF triples written in a turtle-like syntax in which the subject and object of the triples allow for variables that reference SQL columns of the result of the SQL query of the mapping, and for (Skolem) functions applied to them. Intuitively, a mapping axiom defines how the values in each row of the result of an SQL query can be used to generate a set of ABox assertions.

Example 2. Let the TBox and database be those of Example 1. Then the formalization of this scenario in a Quest OBDA model can be done using the following 2 mappings:

```
SELECT id, name, age FROM patient;
{<":patient/{$id}"> rdf:type :Patient;
                    :id $id; :name $name; :age $age^^xsd:int}
SELECT id FROM condition WHERE cond = 22;
{<":patient/{$id}"> rdf:type :CardiacArrestPatient.}
```

Note that by analyzing these mappings, and assuming that there is a foreign key constraint from the *id* attribute of the *condition* table to the same attribute in the *patient* table, a system can detect the containment relationship between the *id*s returned by the second SQL query and those returned by the first one, and optimize the reasoning process accordingly, e.g., avoiding the redundancy issues mentioned before. ■

Important features of the mapping language of Quest include: *(i)* typing of OWL data values, *(ii)* language tags for literal terms, *(iii)* support for URI's and literal constants in the templates, and *(iv)* functions to construct URI's, BNodes, and OWL data values from the SQL values returned by the SQL queries. At the moment, mappings can be created manually by using Quest's API, by writing a `.obda` file in Quest's own syntax, or by using **-ontopPro-** [9], a plugin for Protege 4 that offers many features to facilitate the creation of mappings. In the near future we will also add support for automated mapping generation, as well as support for compatible languages that have different syntaxes, e.g., R2RML and D2RQ.

3 Query answering in Quest

The main service in Quest is SPARQL query answering. The current version of Quest (v1.7-alpha) supports the conjunctive fragment of SPARQL plus `FILTER` expressions.

Support for `UNION`, `OPTIONAL`, and more advanced operators will be added in later versions. A distinguishing feature of Quest are the so called *ABox modes*, i.e., configurations that define how data assertions are given to the system (see Fig. 1). During system initialization, Quest will request a TBox that defines the vocabulary as well as the semantics of the system, then, assertional data is given as follows:

Virtual ABox mode. In this mode data assertions are defined as an OBDA model consisting of a single JDBC data source definition $\mathcal{D}$ and a set $\mathcal{M}$ of mappings for that source. In this mode Quest works in a purely on-the-fly manner, relying solely on SQL query rewriting to produce answers for the user's queries. Note that this allows Quest to always be up-to-date w.r.t. to the data source. Moreover, Quest allows the source to be a data federation system, e.g., Teiid, DB2 data federator, etc., for on-the-fly integration of multiple sources by means of a single ontology.

Classic ABox mode. In this mode Quest works similarly to a traditional OWL reasoner or triple store, in the sense that it will require as input ABox assertions (i.e., data triples). The interesting aspect of this mode is that Quest uses the same mechanism for query answering as the one used in virtual mode. That is, Quest will also use an OBDA model and a RDBMS during query answering; however, in this case the database (schema and data) and the mappings of the OBDA model are defined and managed by Quest itself. The client may choose to let Quest make use of H2, an in-memory database, for this purpose, or may provide Quest with the connection information to access an existing supported RDBMS (e.g., MySQL, PostgreSQL, DB2, or Oracle).

3.1 Quest initialization

Quest's initialization mechanism consists of several steps in which the input TBox and mappings get analyzed and optimized so as to allow the rewriting and unfolding algorithms to be fast, and the SQL queries produced to be minimal and efficient. We will now briefly describe each of these steps. Note that some steps are done exclusively in classic ABox mode and some exclusively in virtual ABox mode.

(i) **Vocabulary optimization.** The first optimization performed by Quest is over the input TBox $\mathcal{T}$. In this stage it will transform $\mathcal{T}$ into a TBox $\mathcal{T}^e$ in which the vocabulary has been simplified by detecting *semantic equivalence* of named classes and named properties. From each equivalence set, Quest will eliminate from $\mathcal{T}^e$ all but one class/property (the 'canonical form'). In addition to $\mathcal{T}^e$, this step also produces an

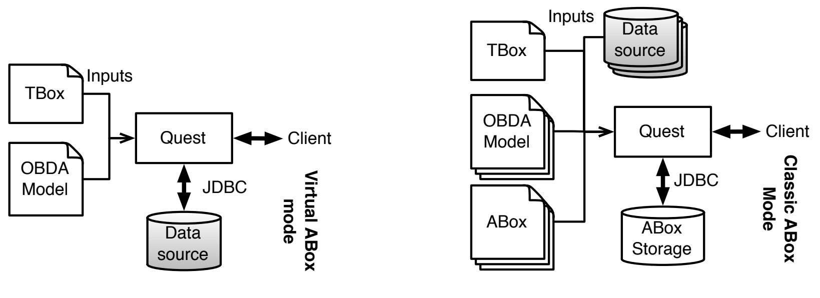

Fig. 1. Quest initialization and configuration based on the ABox mode

equivalence map, where a record of the removed vocabulary, together with the corresponding canonical forms, is kept in order to process queries and/or ABox assertions. Given the extensive use of synonyms for classes, properties, and inverse properties in OWL ontologies, this optimization, though conceptually simple, has a strong impact on the efficiency of the system, including the size of query rewritings. See [10] for details.

(*ii*) RDBMS and OBDA model definition (classic ABox mode). The technique used in this step is called *Semantic Index* repository [10]. The technique parts from the well known notion of *universal tables*, that is often used in triple stores, i.e., *subject*, *property*, *object* tables. Quest uses one table for each type of assertions, i.e., class, object, and data property assertions (in the last case one for each value domain). In these tables, the class or property is determined by a numeric id (*idx*). The assignment of ids to classes and properties is done taking into account the TBox $\mathcal{T}$ such that the value of the identifiers *encode* the implied hierarchical information of $\mathcal{T}$. For example, if the class `Mammal` is the top class of a tree hierarchy consisting of 1000 classes, our technique makes sure that `Mammal` has id 1 and all implied subclasses have ids from 2 to 1000. During processing of ABox assertions, each row will be inserted using those ids, such that the system will be able to create mappings that retrieve all implied ABox assertions by means of intervals in the `WHERE` clauses, e.g., to retrieve all instances of `Mammal`:

```
SELECT subject FROM classtriples WHERE id >= 1 AND id <= 1000;
{?subject rdf:type :Mammal;}
```

Note that this technique makes most of the TBox inferences redundant, since the mappings and database already capture the TBox semantics.

(*iii*) $\mathcal{T}$-Mappings (virtual ABox). This optimization focuses on the enforcement of TBox semantics on the OBDA model used for unfolding and SQL query generation. During this step Quest will create additional mappings to account for the semantics of the TBox. That is, if $\mathcal{T} \models A \sqsubseteq B$, this step will ensure that the data obtained from the mappings for A is also used in mappings for B. This allows Quest to cope with most entailments related to constants, including *subClassOf*, *subPropertyOf*, *domain*, *range*, and *inverseOf*. For example, if the TBox and mappings where those of Examples 1 and 2, this optimization would generate at least the following additional mapping:

```
SELECT id FROM condition WHERE cond = 22;
{<":patient/{$id}"> rdf:type :Patient.}
```

This technique, called *$\mathcal{T}$-Mappings* [8], makes most reasoning w.r.t. $\mathcal{T}$ redundant, similarly to the Semantic Index technique in classic ABox mode.

(*iv*) Mapping program optimization. In this step, Quest transforms the resulting OBDA model into a Datalog program, called the *mapping program*, that has the same semantics, but that is easier to manipulate. To do this, it translates each mapping axiom into a Datalog rule, where the head of the rule corresponds to the target template, and the body of the rule to the SQL query of the mapping. Function symbols are used to account for URI templates, datatype casting and similar operations. For example, the previous mapping is transformed into the following mapping rule:

$$\texttt{Patient}(\textit{template}(\texttt{":patient/\{\}"}, x)) \leftarrow \textit{condition}(x, y), y = 22$$

Note the use of the function $template$, which abstracts away the meaning of URI templates. Also note that to create the body of this rule it is necessary to obtain metadata about the schema of the database, and to parse the SQL query in the mapping. Moreover, each rule must have only one atom in the head, hence, this step requires to *split* mappings that contain multiple triple patterns in the target template into simple mappings, with only one pattern. Once the mapping program is setup, Quest optimizes it by removing any rules that are redundant with respect to query containment. At this stage, in virtual ABox mode, Quest also uses information about primary keys and other constraints obtained from a DB metadata analysis step.

***(v)* TBox redundancy elimination.** This optimization exploits information about completeness of the OBDA model of the system. Intuitively, Quest detects which inferences are already taken into account by the OBDA model and produces a new TBox in which only non-redundant inferences are entailed. The combination of this optimization with the $\mathcal{T}$-Mappings technique in virtual ABox mode, or the Semantic Index technique in classic ABox mode allows Quest to dramatically simplify the TBox used for rewriting.

***(vi)* Component setup.** Last, having set up/optimized the internal TBox and OBDA model, Quest will initialize the rewriting, unfolding and SQL execution engines and is ready for query answering.

3.2 Quest run-time

Query answering in Quest is done by means of query rewriting into SQL. Given an input query Q, three steps are performed at query answering time: *(i)* query rewriting, where Q is transformed into a new query Q' that takes into account the semantics of the TBox; *(ii)* query unfolding, where Q' is transformed into a single SQL query using the mappings of the OBDA model; and *(iii)* query evaluation, where the SQL query is executed over the data storage and the results are streamed to the client.

We provide now an intuitive description of the first two query execution steps. For the third one, we rely on standard relational technology.

***(i)* Query rewriting.** [3] Intuitively, query rewriting uses the TBox axioms to compute a set of queries that *encode* the semantics of the TBox, such that if we evaluate the union of these queries over the ABox we will obtain sound and complete answers. The query rewriting process works using the TBox axioms and unification to generate *more specific* queries from the original, *most general* query. In Quest, this process is iterative, and stops when no more queries can be generated.

Example 3. Let the TBox be as in Example 1, and let Q be the following SPARQL query that asks for the URI, name, and age of all patients with a hearth condition and age between 21 and 70:

```
SELECT ?p ?n ?a WHERE {
 ?p a Patient; name ?n; age ?a; affectedBy [a HeartCondition].
FILTER (?a >= 21 & ?a <= 70) }
```

From this query, making use of the `SubClassOf` axioms, the query rewriting algorithm of Quest generates the following rewriting:

```
SELECT ?p ?n ?a WHERE {
 {?p a Patient; name ?n; age ?a; affectedBy [a HeartCondition]} UNION
 {?p a Patient; name ?n; age ?a; affectedBy [a CardiacArrest]} UNION
 {?p a Patient; name ?n; age ?a; a CardiacArrestPatient} UNION ...
FILTER (?a >= 2 & ?a <= 70) }
```
■

Due to space limitations we cannot provide more details here. However, we note that the query rewriting algorithm is a variation of the PerfectRef algorithm [3] optimized in order to obtain a fast query rewriting procedure that generates a minimal amount of queries. In particular, it has been restructured to anticipate termination w.r.t. saturation and avoid *useless* unification of atoms, and has been extended with extensive use of query containment w.r.t. dependencies to minimize the size of the rewriting. Also note that the techniques introduced before (Semantic Index, $\mathcal{T}$-mappings, and TBox optimizations) have reduced the size of the TBox, and most queries produced by rewriting can also be eliminated by means of containment checks. In many cases, e.g., for grounded queries, the rewriting stage in Quest will generate just a few queries.

(ii) **Query unfolding.** Intuitively, query unfolding uses the mapping program (see Section 3.1.(*iv*)) to transform the rewritten query into SQL. First, Quest transforms the rewritten query into a Datalog program, by translating each subquery into a Datalog rule such that each graph pattern becomes a Datalog atom (`rdf:type` patterns become unary atoms, other patterns become binary atoms, `FILTER` expressions become (in-)equality atoms and the `SELECT` expression becomes the head of the rule). Then, the program is resolved [5] against the mapping program, until no resolution step can be applied. At each resolution step, Quest replaces an atom formulated in terms of TBox predicates, with the body of a mapping rule. In case there is no matching rule for an atom, that sub-query is logically empty and is eliminated. Finally, when no more resolution steps are possible, we have a new Datalog program, formulated in terms of database tables, that can be directly translated into a union of select-project-join SQL queries. Also at this step, Quest makes use of query containment w.r.t. dependencies to detect redundant queries and to eliminate redundant join operations in individual queries (i.e., using primary key metadata). Last, Quest always attempts to generate simple queries, with no sub-queries or structurally complex SQL. This is necessary to ensure that most RDBMS engines are able to generate efficient execution plans.

To complement the description of the unfolding procedure, we now provide an example of the SQL queries generated by Quest in virtual mode.

Example 4. Consider the query rewriting generated in Example 3 and the OBDA model from Example 1. Note that from the 3 subqueries in the rewritten query, only the last one can be non-empty, since the graph patterns involving the property `affectedBy` cannot be matched with any mapping in the OBDA model. Moreover, given that the column *id* is a primary key in the table *patient*, the first three simple graph patterns in this query can be satisfied using a single reference to the *patient* table. Then we have that Quest unfolds the rewritten query into the following SQL query:

```
SELECT '&base;/patient/' || v1.id as p, v1.name as name,
  v1.age as age
FROM patient v1 JOIN condition v2 ON v1.id = v2.id
WHERE v2.cond = 22 AND v1.age >= 21 AND v1.age <= 70
```

Note that this query already returns URI's as specified by the mappings, and that the conditions of the query are formulated over the original columns of the tables. This allows the RDBMS to exploit any index that may be available, and to obtain an efficient query execution plan. ■

4 Related work

With respect to support for on-the-fly query answering over relational sources, systems like Virtuoso and Mastro-i [2] are pioneers in the area. In comparison to Virtuoso, Quest reduces query answering to SQL query evaluation, generating a single SQL query that covers the data access and the reasoning process, while Virtuoso handles reasoning by means of iterative access to the data source, performing join and other operations on its own. This difference puts Quest closer to systems like Mastro-i than to Virtuoso. At the same time, Mastro-i and Quest are different in the adopted optimizations as well as in the support for OWL-oriented OBDA models. Also, Quest makes extensive use of TBox and mapping transformations, database metadata and query containment w.r.t. dependencies that, to the best of our knowledge, are not available in Mastro-i.

In the context of query rewriting systems we mention Requiem [6], QuOnto [1], Presto [12], and Owlgres [13]. These systems are comparable to Quest in that they rely on query rewriting w.r.t. to the TBox for query answering. At the same time, all systems differ from Quest in their user of the database layer, since none of these systems use the database for reasoning. In contrast, Quest embeds most of the TBox semantics into the database by means of the Semantic Index technique. This key difference allows Quest to produce smaller queries than all these systems, since it encodes inferences in succinct numeric intervals, instead of individual queries or Datalog rules. Our experiments have shown that in TBoxes with large hierarchies, Quest generates SQL queries that are orders of magnitude smaller than those produced by Owlgres, Quonto, or Requiem and significantly smaller than those of Presto (see [10] for details). At the same time, the UCQ-based rewriting technique implemented in Quest is by nature, slower than that of Presto (CNF vs. DNF). We believe that the ideal system should combine Quest's techniques with a *succinct rewriting* technique such as Presto's or the one in [4].

Last, also triple stores like Jena and Sesame provide similar functionality to Quest in classic ABox mode. These systems are fundamentally different in that their inference engines are based on *forward-chaining*, a rule-based inference technique that materializes ground inferences at load time. Initial experiments (see [10]) with Sesame 2.6 have shown that with large TBoxes (thousands of axioms) and large datasets (millions of triples), Quest can provide better performance than Sesame at load time (up to 50 times faster) and comparable performance at query answering time. At the same time, these engines support full SPARQL while, at the moment, Quest supports only fragments.

5 Conclusions

In this paper we have introduced Quest, a new query answering system with support for SPARQL query answering under the OWL 2 QL and RDFS entailment regimes. We have also described some of the more relevant aspects of the system w.r.t. OBDA, as well as some of the optimizations that it implements. The current system supports only the conjunctive query fragment of SPARQL 1.0 with FILTERs, and future work includes the extension of the query answering technique to allow for broader fragments of SPARQL 1.1. We will also look into the issue of extending the rewriting techniques implemented in Quest to support different OWL 2 fragments, e.g., OWL 2 EL and OWL 2 RL possibly in combination with fragments of SWRL.

Acknowledgements. This research has been partially supported by the EU under the ICT Collaborative Project ACSI (Artifact-Centric Service Interoperation), grant agreement n. FP7-257593.

References

1. A. Acciarri, D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, M. Palmieri, and R. Rosati. QuOnto: QUerying ONTOlogies. In *Proc. of the 20th Nat. Conf. on Artificial Intelligence (AAAI 2005)*, pages 1670–1671, 2005.
2. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, R. Rosati, M. Ruzzi, and D. F. Savo. The Mastro system for ontology-based data access. *Semantic Web J.*, 2(1):43–53, 2011.
3. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. of Automated Reasoning*, 39(3):385–429, 2007.
4. S. Kikot, R. Kontchakov, and M. Zakharyaschev. On (in)tractability of OBDA with OWL 2 QL. In *Proc. of the 24th Int. Workshop on Description Logic (DL 2011)*, 2011.
5. A. Leitsch. *The Resolution Calculus*. Springer, 1997.
6. H. Pérez-Urbina, B. Motik, and I. Horrocks. Tractable query answering and rewriting under description logic constraints. *J. of Applied Logic*, 8(2):186–209, 2010.
7. A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *J. on Data Semantics*, X:133–173, 2008.
8. M. Rodríguez-Muro and D. Calvanese. Dependencies: Making ontology based data access work in practice. In *Proc. of the 5th Alberto Mendelzon Int. Workshop on Foundations of Data Management (AMW 2011)*, volume 749 of *CEUR Electronic Workshop Proceedings,* `http://ceur-ws.org/`, 2011.
9. M. Rodriguez-Muro and D. Calvanese. -ontop- framework. Website, Apr. 2012. `http://obda.inf.unibz.it/protege-plugin/`.
10. M. Rodriguez-Muro and D. Calvanese. High performance query answering over *DL-Lite* ontologies. In *Proc. of the 13th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2012)*, 2012.
11. M. Rodriguez-Muro and D. Calvanese. The Quest system for OBDA. Technical report, KRDB Research Centre for Knowledge and Data, Free University of Bozen-Bolzano, 2012.
12. R. Rosati and A. Almatelli. Improving query answering over *DL-Lite* ontologies. In *Proc. of the 12th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2010)*, pages 290–300, 2010.
13. M. Stocker and M. Smith. Owlgres: A scalable OWL reasoner. In *Proc. of the 5th Int. Workshop on OWL: Experiences and Directions (OWLED 2008)*, volume 432 of *CEUR Electronic Workshop Proceedings,* `http://ceur-ws.org/`, 2008.