Data & Knowledge Engineering 72 (2012) 172–201

Contents lists available at SciVerse ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

The extension-based inference algorithm for *pD**

Övünç Öztürk [a,*], Tuğba Özacar [a], Murat Osman Ünalır [b]

[a] Department of Computer Engineering, Celal Bayar University, Muradiye, 45140, Manisa, Turkey
[b] Department of Computer Engineering, Ege University, Bornova, 35100, İzmir, Turkey

ARTICLE INFO

Article history:
Received 24 June 2010
Received in revised form 16 October 2011
Accepted 17 October 2011
Available online 25 October 2011

Keywords:
Ontology
Rule-based reasoning
Owl *pD**
Scalable reasoning

ABSTRACT

In this work, we present a scalable rule-based reasoning algorithm for the OWL *pD** language. This algorithm uses partial materialization and a syntactic ontology transformation (the extension-based knowledge model) to provide a fast inference. Because the materialized part of the ontology does not contain assertional data, the time consumed by the process, and the number of inferred triples, remain fixed with varying amounts of assertional data. The algorithm uses database reasoning and a query rewriting technique to handle the remaining inference. The extension-based knowledge model and the database reasoning prevent the expected decreases in query performances, which are the natural result of online reasoning during query time. This work also evaluates the efficiency of the proposed method by conducting experiments using LUBM and UOBM benchmarks.

1. Introduction

The Semantic Web extends the current Web by structuring information to better enable computers and people to work in cooperation. The Semantic Web is a Web of meaning: software agents understand what the entities on the Web mean, and can make use of that knowledge. In the most influential Semantic Web article [1], ontologies are proposed as a flexible solution to the problem of representing and sharing the meaning of specific knowledge [2]. A widely-cited paper [3] describes ontology as "an explicit specification of a conceptualization."

OWL is the W3C recommendation for creating and sharing ontologies on the Web. An OWL ontology consists of two parts: the intensional part, called the TBox, contains knowledge about classes, and relationships between the classes, and the extensional part, known as the ABox, contains knowledge about entities, and how they relate to the classes and the roles of the intensional part. TBox represents a very small percentage of the ontological data in most real world ontologies. In view of the number of Web pages, it is apparent that reasoning on the Semantic Web will have to deal with very large ABoxes [4]. The ABox is not only expected to be the largest part of an ontology but is also subject to frequent changes [5]. As the ABox of the ontology grows, repetitions in the inference procedure increase, and the inference process slows at an exponential rate. Therefore, the complexity of reasoning on the Semantic Web is closely related to the complexity of ABox reasoning, which is also called data complexity.

In a nutshell, reasoning performance is perhaps the single biggest challenge relating to reasoning and Semantic Web topics [6], especially when considering that the number of linked datasets is growing. For example, the LOD (Linked Open Data) community[1] aims to extend the Web with a data commons by publishing various open datasets as RDF (Resource Description Framework) on the Web and by setting RDF links between data items from different data sources. In 2007, datasets consisted of over two billion

* Corresponding author at: Department of Computer Engineering, Celal Bayar University, Muradiye, 45140, Manisa, Turkey. Tel.: +90 2362372886; fax: +90 2362372442.
E-mail addresses: ovunc.ozturk@bayar.edu.tr, ovunc.ozturk@gmail.com (Ö. Öztürk), tugba.ozacar@bayar.edu.tr, tugba.ozacar@gmail.com (T. Özacar), murat.osman.unalir@ege.edu.tr (M.O. Ünalır).

[1] http://linkeddata.org/.

0169-023X/$ – see front matter © 2011 Published by Elsevier B.V.
doi:10.1016/j.datak.2011.10.003

RDF triples, which were interlinked by over two million RDF links. By 2010 this had grown to 25 billion RDF triples, interlinked by approximately 395 million RDF links.

In this work, we propose a novel inference algorithm, namely the extension-based inference algorithm, for large-scale ontological datasets. This algorithm employs both forward and backward rule engines in conjunction. For further optimization, the algorithm uses extensions, which can be defined as partial sets of concept individuals. In [7], we used extensions with forward chaining algorithm for scaling up RDFS reasoning. In this work, we also use database reasoning and backward chaining, to increase the level of inference to pD^*, which weakens the standard iff-semantics of OWL and extends RDFS entailment (see Section 2, for if/iff semantics see [8]).

Our contributions in this work are as follows: (a) we define a syntactic transformation on pD^* ontologies, (b) we present a materialization technique, which uses this transformation to perform scalable reasoning on large ABoxes. This technique filters the ontology triples about individuals, and, as a consequence, time consumption of the reasoning process remains fixed even when the size of the instance data increases, (c) we exploit the transformation and database reasoning for performing scalable query answers, and (d) all of the notions in (a), (b), and (c) provide a reasonable complexity of reasoning and querying without sacrificing too much expressive power.

The rest of this paper is organized as follows: in Section 2, we give some background knowledge about the field of ontology reasoning. In Section 3, we focus on our modified knowledge representation formalism, "the extension-based knowledge model." In Section 4, we define extension-based reasoning and querying algorithms in detail. To further illustrate the extension-based inference algorithm, we provide an example of running our algorithm in Section 5. Section 6 presents some theoretical results about the complexity of the extension-based reasoning algorithm. We provide experimental results for our approach using LUBM and UOBM benchmarks in Section 7. In Section 8, we compare our work with state-of-the-art reasoning techniques. Finally, we offer our conclusions and future directions in Section 9.

2. Background knowledge

An ontology is an explicit specification of a conceptualization [3]. An ontology defines the terms used to describe and represent an area of knowledge [9]. An ontology has two components, the TBox and the ABox. These components are defined in [10], as follows:

- The TBox (assertions on concepts) stores assertions stating general properties of concepts and roles. For example, an assertion of this type is the statement that a concept represents a specialization of another concept. The TBox of an ontology is more resistant to change compared to extensional knowledge of the ABox.
- The ABox (assertions on individuals) comprises assertions on individual objects. A typical assertion in the ABox is the statement that an individual is an instance of a certain concept. The ABox is usually the largest part of an ontology and is also subject to frequent changes [5].

OWL (the Web Ontology Language) is a family of knowledge representation languages for authoring ontologies. OWL became a W3C (World Wide Web Consortium) Recommendation, namely a web standard, in February 2004. OWL is built on top of RDFS, RDF and XML. RDF and RDF Schema provide basic capabilities for describing vocabularies that describe resources. RDF Schema contains primitives for defining classes, properties, subclass/subproperty relations, class individuals, and relations between classes and class individuals. OWL extends these languages with a rich set of modeling constructors, which are presented in Appendix A.

Reasoning is used to infer information that is not explicitly represented in an ontology. Reference [11] divides reasoning strategies into two groups, as follows:

- *DL reasoning paradigm*: this paradigm is based on the notion of Classical Logics, such as Description Logics [12]. In this case, the semantics of OWL ontologies can be handled by DL reasoning systems, such as Pellet [13], RacerPro [14] and Fact++ [15], which reuse existing DL algorithms, such as tableaux-based algorithms [12].
- *Datalog paradigm*: in this case, a subset of the OWL semantics is transformed into rules that are used by a rule engine to infer implicit knowledge.

The DL reasoning engines have an inefficient instance reasoning performance, whereas rules are insufficient to model certain situations related to the open nature of the Semantic Web. Obviously, the selection of the most suitable modeling paradigm depends on the domain and on the needs of the application. There are also other efforts that work to combine both strategies. For example, CLIPS-OWL [16] incorporates the extensional results of DL reasoning in CLIPS production rule programs.

The extension-based inference algorithm is designed for rule-based reasoning, which applies entailment rules to the knowledge base to produce new facts. We present a definition of an entailment rule [17] that we follow in the rest of the paper:

Definition 1. An entailment rule for an ontology graph G is of the form,

$$\langle s_1\ p_1\ o_1\rangle\langle s_2\ p_2\ o_2\rangle\ldots\langle s_n\ p_n\ o_n\rangle\rightarrow\langle s'_1\ p'_1\ o'_1\rangle\langle s'_2\ p'_2\ o'_2\rangle\ldots\langle s'_m\ p'_m\ o'_m\rangle,$$

where $n \geq 1$, $m \geq 1$, s_i, s'_i, p_i and p'_i are RDF URI references or blank nodes, and o_i and o'_i are RDF URI references, blank nodes or literals. The triples on the left-side of the rule denote the condition of the entailment and the triples on the right-side denote the conclusion. The condition of the rule denotes the RDF triples that should exist in G, and the conclusion denotes the RDF triples that should be added in G. If $n = 0$, then all of the conclusion triples should always exist in G (axiomatic triples).

If $m = 0$, then the entailment denotes that the triple pattern of the body should be viewed as inconsistent (inconsistency entailment).

There are two types of rule-based reasoning algorithms, which are defined in [18], as follows:

- *Forward chaining (offline reasoning)*: to start from known facts (explicit statements) and to perform inference in an inductive fashion. An *inferred closure* is the extension of a KB (knowledge base) with all of the implicit facts that could be inferred from it, based on the enforced semantics. *Total materialization* is the inference strategy, which performs total forward chaining and computes the inferred closure of a KB. The advantages (+) and the disadvantages (−) of total materialization are listed below:
  + Query performance is relatively better, because no reasoning is required at the time of query answering.
  – Upload, storage, and addition and removal of new facts is relatively slow, because all the reasoning is performed during the upload. Moreover, all of the reasoning is computed from scratch after adding or removing a new fact.
  – The inference process requires considerable additional space (RAM, disk, or both).

  The other inference strategy is *partial materialization*, which selectively computes a proper subset of the inferred closure to reduce the disadvantages of total materialization.
- *Backward chaining (online reasoning)*: to start from a specific fact or a query and to verify it or get all possible results (bindings for free variables), using deductive reasoning.

Both of the above methods have advantages and disadvantages. Backward chaining has smaller storage requirements but is slow in query answering. On the other hand, forward chaining is fast on query answering but has huge memory requirements. There are other inference strategies that combine the two strategies and avoid the disadvantages of both. These have proven to be efficient in many contexts [18].

Yet another issue regarding rule-based reasoning is to guarantee the completeness and the decidability of reasoning. There are some works (such as RDF MT and RDFS(FA) [19]) that define sublanguages of the OWL and RDF languages and that reduce the complexity and the time-consumption of reasoning. In this paper, we describe a reasoning engine for the pD^* language that weakens the standard iff-semantics of OWL and extends RDFS entailment. pD^* Entailment is largely defined by means of "if conditions", and extends RDFS with datatypes and a property-related fragment of OWL (see Appendix B).

Due to the huge size of the Semantic Web ontologies, it will be necessary to use database technology to provide persistence to the knowledge described by the ontologies, as well as scalability to the queries and reasoning on the knowledge [20]. Therefore, relational databases are extensively used as an efficient means for storing ontologies. Database-based ontology repositories can be divided into three major categories [21]: (a) *generic RDF stores*, which mainly use a relational table of three columns (Subject, Property, Object) to store all triples (e.g., Jena [22] and Oracle [23]); (b) *binary table-based stores*, whose schema changes with ontologies (e.g., DLDB-OWL [19]). In this kind of repository, a class table stores all instances belonging to the same class, and a property table stores all triples that have the same property; and (c) *improved triple stores*, such as Minerva [24], OntoMinD [25] and Sesame, manage different types of triples using different tables.

It is also interesting to note that, there are some works on efficient reasoning with modular ontologies in light of the fact that reasoning engines need to only process the knowledge bases of the relevant modules (e.g., [26]).

3. Extension-based knowledge model

3.1. Extension-based knowledge model constructs

The extension-based knowledge model works on a simple principle, *creating groups for individuals of a concept that is the extension or denotation of the concept*. In this model, we define four types of grouping constructs:

hasClassExtension relates every class to one of its class extensions, which holds certain individuals (either explicit or implicit individuals) of the class. A class extension is related to each of its members via a "contains" predicate. *hasClassExtension* has two subproperties: *hasExplicitClassExtension* and *hasInferredClassExtension*. *hasExplicitClassExtension* relates the class to its unique explicit class extension, which holds all explicit individuals of the class. The property *hasInferredClassExtension* relates the class to one of its inferred class extensions, which holds a part of the inferred individuals of the class. Each inferred class extension holds the individuals, which belong to one of the subclasses of the class. The union of the inferred class extensions constitutes the individuals, which belong to the subclasses of the class.

Fig. 1 shows three classes and their explicit extensions. C_1 has two subclasses, C_2 and C_3. E_1 holds the explicit individuals of C_1, E_2 holds the explicit individuals of C_2, and E_3 holds the explicit individuals of C_3. In this case, there are *hasExplicitClassExtension* relations between C_1 and E_1, C_2 and E_2, and C_3 and E_3. The implicit *hasInferredClassExtension* relations between class C_1 and the extensions E_2 and E_3 are inferred from *hasExplicitClassExtension* relations.

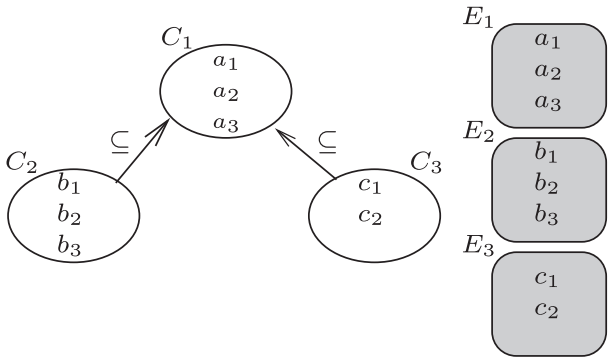

Fig. 1. The relations between classes and their extensions.

hasSubjectExtension/hasObjectExtension relates every property (except "type") to one of its subject/object extensions, which holds the subjects/objects of certain individuals (either explicit or implicit individuals) of the property. A subject/object extension is related to each of its members via a "contains" predicate. *hasSubjectExtension/hasObjectExtension* has two subproperties: *hasExplicitSubjectExtension/hasExplicitObjectExtension* and *hasInferredSubjectExtension/hasInferredObjectExtension*. *hasExplicitSubjectExtension/hasExplicitObjectExtension* relates the property to its unique explicit subject/object extension, which holds subjects/objects of all explicit property individuals. *hasInferredSubjectExtension/hasInferredObjectExtension* relates the property to one of its inferred subject/object extensions, which holds the subjects/objects of a subproperty of the property. The union of the inferred subject/object extensions constitutes the individuals, which belong to the subproperties of the property.

Fig. 2 shows three properties and their explicit extensions. P_1 has two subproperties, P_2 and P_3. S_1 holds the subjects of explicit individuals of P_1. S_2 holds the subjects of explicit individuals of P_2, and S_3 holds the subjects of explicit individuals of P_3. Similarly, O_1 holds the objects of explicit individuals of P_1; O_2 holds the objects of explicit individuals of P_2, and O_3 holds the objects of explicit individuals of P_3. In this case, there are *hasExplicitSubjectExtension/hasExplicitObjectExtension* relations between P_1 and S_1/O_1, P_2 and S_2/O_2, and P_3 and S_3/O_3. The implicit *hasInferredSubjectExtension/hasInferredObjectExtension* relations between property P_1 and extensions S_2/O_2 and S_3/O_3 are inferred from *hasExplicitSubjectExtension/hasExplicitObjectExtension* relations.

hasPropertyExtension relates every property (except "type") to one of its extensions, which symbolizes certain individuals (either explicit or implicit individuals) of the property. The construct *hasPropertyExtension* has two subproperties: *hasExplicitPropertyExtension* and *hasInferredPropertyExtension*. *hasExplicitPropertyExtension* relates the property to its unique explicit property extension, which symbolizes all explicit individuals of the property. *hasInferredPropertyExtension* relates the property to one of its inferred property extensions, which symbolizes the certain inferred individuals of the property. Each inferred property extension holds the individuals, which belong to one of the subproperties of the property. The union of the inferred property extensions constitutes the individuals, which belong to the subproperties of the property. This grouping construct differs from the others in that it lacks a concrete extension. Its extension is an empty and virtual list, which is not related to any of its items with a "contains" predicate. The reason for not keeping the individuals of this extension is to avoid a large increase in the triple count, after transformation.

3.2. Extension-based knowledge model transformation algorithm

Transforming an OWL ontology to the extension-based knowledge model using Algorithm 1 involves a syntactic ontology transformation and does not change the semantics of the ontology language.

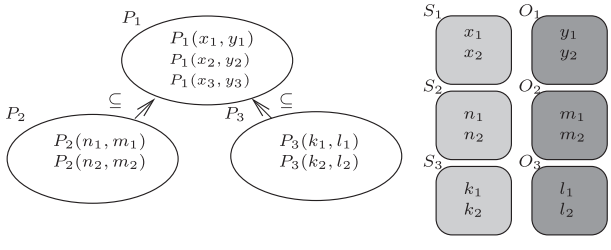

Fig. 2. The relations between properties and their subject/object extensions.

Table 1
The columns of *contains* and *statements* tables.

Table name	The name of columns
contains	*extension_name, resource*
statements	*subject, predicate, object*

Algorithm 1. The transformation algorithm.

```
for each triple (spo) in ontology{
   if predicate p is "rdf: type"{
      if (class extension of object o is not defined)
         add triple (o hasExplicitClassExtension ec)
      substitute (s rdf: type o) with (ec contains s)
   }
   else
   {
      if (subject extension of predicate p is not defined)
         add triple (p hasExplicitSubjectExtension es)
      add triple (es contains s)
      if (object extension of predicate p is not defined)
         add triple (p hasExplicitObjectExtension eo)
      add triple (eo contains o)
   }
}
```

3.3. Extension-based knowledge model database schema

The extension-based knowledge model uses a generic RDF store, which is mainly constituted of two database tables: *contains* and *statements* (Table 1). The *contains* table maps each extension member to its extension, and the *statements* table stores all of the remaining triples in the ontology. In addition to these two tables, there are also auxiliary database tables, which are described in Section 4.2.3.

4. Extension-based reasoning and querying

Fig. 3 shows the extension-based inference process. A syntactic transformation (see Section 3.2) is applied to the raw ontological data. Both ontology schema and instance data are transformed to their equivalents in the extension-based knowledge model. The reasoning on ontology schema is performed in the main memory by the forward chaining process; then, both closures of the schema and the instance data are moved to the database. The rest of the inference is completed using database reasoning and backward chaining (via a query rewriting mechanism, which will be described in Section 4.3).

4.1. Extension-based knowledge model entailment rules

The extension-based knowledge model uses a set of entailment rules, which contain the transformed pD^* entailment rules and additional rules that involve relationships between concepts and their extensions.

4.1.1. pD^* Entailment rules

pD^* Semantics extends the "if-semantics" of RDFS to a subset of the OWL vocabulary.[2] pD^* Provides reasonable computational properties without sacrificing too much expressive power. The original pD^* entailment rule sets [8] are given in Tables 24 and 25 in Appendix B. These rules are shown to be sound and complete with respect to the pD^* semantics. In this section, we define a guideline to transform pD^* entailment rules into their equivalents in the extension-based knowledge model.

This transformation involves substituting conditions of pD^* rules, which match members of an extension, for conditions that will match the extension itself. Not every rule is transformed in this way. The transformed rules are executed during the *forward chaining* process. The rules, which cannot be transformed, are applied during *expanding the extensions* or during *query answering*. Fig. 4 shows rule patterns (*P1* through *P9*) that are used to classify pD^* entailment rules.

[2] RDF and OWL differ in the way in which their semantics is defined. The semantics of RDF and RDFS is defined using if conditions, whereas the semantics of OWL uses many if-and-only-if conditions. A semantics that uses iff conditions in its specification is more powerful, in the sense that it leads to more entailments [8].

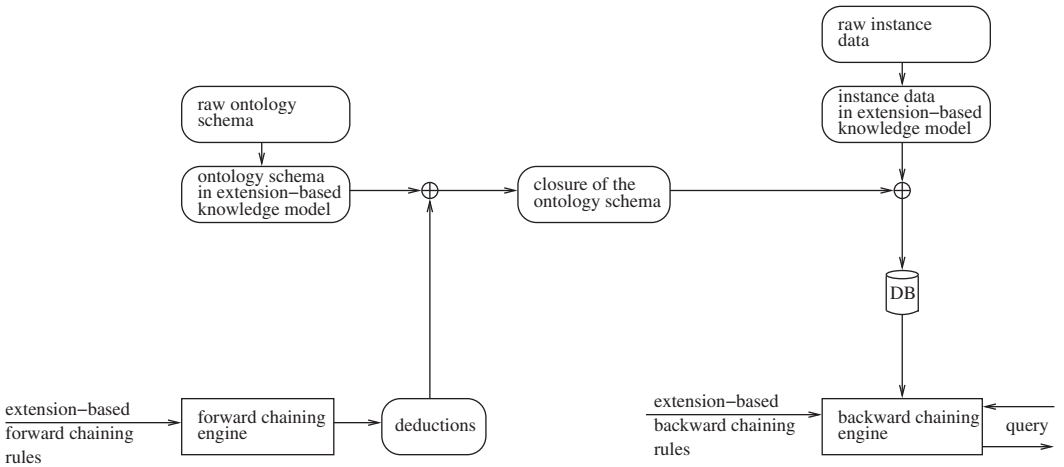

Fig. 3. The extension-based inference process.

A pattern is the conjunction of the following statements (s_1 through s_7), where R is the rule that matches the pattern, c is a condition of R, $subj_c$ is the subject of c, $pred_c$ is the predicate of c, obj_c is the object of c, lhs_R is the left-hand-side of R, rhs_R is the right-hand-side of R, $Cond_{ABox}$ is the set of all possible conditions involving ABox, and $Cond_{TBox}$ is the set of all possible conditions involving TBox:

- S_1:$\forall c \in lhs_R(c \in Cond_{TBox}$: all conditions in the LHS (left hand-side) of the rule involve TBox.
- S_2:$\exists c_1 \in lhs_R(pred_{c_1} = rdf:type)$: there is at least one condition with a "type" predicate in the LHS of the rule.
- S_3:$\forall c \in lhs_R(c \in Cond_{ABox})$: all conditions in the LHS of the rule involve ABox.
- S_4:$\exists c_1 \in lhs_R(c_1 \in Cond_{ABox} \wedge pred_{c_1} \neq rdf:type)$: there is at least one condition matching with a property individual in the LHS of the rule.
- S_5:$\forall c \in rhs_R(pred_c = rdf: type)$: the condition on the RHS (right hand-side) of the rule has "type" predicate.
- s_6:$\exists c_1 \in lhs_R(pred_{c_1} = owl:hasValue \vee pred_{c_1} = owl:someValuesFrom \vee pred_{c_1} = owl:allValuesFrom)Z$: there is at least one condition involving property restrictions in the LHS of the rule.

Table 2
The patterns that identify when the rules are executed.

ID	Rules matched with the pattern	[X]
P1	*rdfs5, rdfs11, rdfp12a, rdfp12b,rdfp12c, rdfp13a, rdfp13b, rdfp13c*	$\mathcal{F}_n$
P2	*rdfs6, rdfs8, rdfs10, rdfp9, rdfp10*	$\mathcal{F}_t$
P3	*rdfs9, rdfs4a, rdfs4b*	$\mathcal{F}_t$
P4	*rdfs7x*	$\mathcal{F}_t$
P5	*rdfp14bx*	$\mathcal{E}$
P6	*rdfp14a, rdfp15, rdfp16*	$\mathcal{Q}$
P7	*rdfs2, rdfs3*	$\mathcal{F}_t$
P8	*rdfp5a, rdfp5b, rdfp6, rdfp7,rdfp11*	$\mathcal{E}$
P9	*rdfp1, rdfp2, rdfp3, rdfp4, rdfp8ax, rdfp8bx*	$\mathcal{F}_t, \mathcal{E}$

Table 3
The D^* entailment rules after transformation.

rdfs2: *p domain u* $\wedge$ *p hasSubjectExtension e* $\Rightarrow$ *u hasInferredClassExtension e*
rdfs3: p range u $\wedge$ *p hasObjectExtension e* $\Rightarrow$ *u hasInferredClassExtension e*
rdfs4a : *p hasSubjectExtension e* $\Rightarrow$ *Resource hasInferredClassExtensione*
rdfs4b : *p hasObjectExtension e* $\Rightarrow$ *Resource hasInferredClassExtension e*
rdfs5 : *v subPropertyOf w* $\wedge$ *w subPropertyOf u* $\Rightarrow$ *v subPropertyOf u*
rdfs6 : *v hasExplicitPropertyExtension e* $\Rightarrow$ *v subPropertyOf v*
rdfs7x.I : *p subPropertyOf q* $\wedge$ *p hasPropertyExtension e* $\Rightarrow$ *q hasInferredPropertyExtension e*
rdfs7x.II : *p subPropertyOf q* $\wedge$ *p hasObjectExtension e* $\Rightarrow$ *q hasInferredObjectExtension e*
rdfs7x.III : *p subPropertyOf q* $\wedge$ *p hasSubjectExtension e* $\Rightarrow$ *q hasInferredSubjectExtension e*
rdfs8 : *v hasExplicitClassExtension e* $\Rightarrow$ *v subClassOf Resource*
rdfs9 : *v subClassOf w* $\wedge$ *v hasClassExtension e* $\Rightarrow$ *w hasInferredClassExtension e*
rdfs10 : *v hasExplicitClassExtension e* $\Rightarrow$ *v subClassOf v*
rdfs11 : *v subClassOf w* $\wedge$ *w subClassOf u* $\Rightarrow$ *v subClassOf u*

Table 4
The p entailment rules after transformation.

$rdfp1: p\ type\ FunctionalProperty \wedge p\ hasPropertyExtension\ e \Rightarrow p\ hasFunctionalPropertyExtension\ e$
$rdfp2: p\ type\ InverseFunctionalProperty \wedge p\ hasPropertyExtension\ e \Rightarrow p\ hasInverseFunctionalPropertyExtension\ e$
$rdfp3: p\ type\ SymmetricProperty \wedge p\ hasPropertyExtension\ e \Rightarrow p\ hasSymmetricPropertyExtension\ e$
$rdfp4: p\ type\ TransitiveProperty \wedge p\ hasPropertyExtension\ e \Rightarrow p\ hasTransitivePropertyExtension\ e$
$rdfp5a: u\ p\ w \Rightarrow u\ sameAs\ u$ $rdfp5b: u\ p\ w \Rightarrow w\ sameAs\ w$
$rdfp6: v\ sameAs\ w \Rightarrow w\ sameAs\ v$ $rdfp7: u\ sameAs\ v \wedge v\ sameAs\ w \Rightarrow u\ sameAs\ w$
$rdfp8ax.I: p\ inverseOf\ q \wedge p\ hasPropertyExtension\ e \Rightarrow q\ hasInversePropertyExtension\ e$
$rdfp8bx.I: p\ inverseOf\ q \wedge q\ hasPropertyExtension\ e \Rightarrow p\ hasInversePropertyExtension\ e$
$rdfp8ax.II: p\ inverseOf\ q \wedge p\ hasInversePropertyExtension\ e \Rightarrow q\ hasPropertyExtension\ e$
$rdfp8bx.II: p\ inverseOf\ q \wedge q\ hasInversePropertyExtension\ e \Rightarrow p\ hasPropertyExtension\ e$
$rdfp9: v\ hasExplicitSubExtension\ e \wedge v\ sameAs\ w \Rightarrow v\ subClassOf\ w$
$rdfp10: p\ hasExplicitPropertyExtension\ e \wedge p\ sameAs\ q \Rightarrow p\ subPropertyOf\ q$
$rdfp11: u\ p\ v \wedge u\ sameAs\ u' \wedge v\ sameAs\ v' \Rightarrow u'\ p\ v'$
$rdfp12a: u\ equivalentClass\ w \Rightarrow u\ subClassOf\ w$
$rdfp12b: u\ equivalentClass\ w \Rightarrow w\ subClassOf\ u$
$rdfp12c: v\ subClassOf\ w \wedge w\ subClassOf\ v \Rightarrow v\ equivalentClass\ w$
$rdfp13a: v\ equivalentProperty\ w \Rightarrow v\ subPropertyOf\ w$
$rdfp13b: v\ equivalentProperty\ w \Rightarrow w\ subPropertyOf\ v$
$rdfp13c: v\ subPropertyOf\ w \wedge w\ subPropertyOf\ v \Rightarrow v\ equivalentProperty\ w$

- $s_7: \exists c_1 \in lhs_R(obj_{c_1} = owl:TransitiveProperty \vee obj_{c_1} = owl:SymmetricProperty \vee obj_{c_1} = owl:FunctionalProperty \vee obj_{c_1} = owl:InverseFunctionalProperty \vee pred_{c_1} = owl:inverseOf)$Z: there is at least one condition involving property characteristics in the LHS of the rule.

Table 2 shows the pD^* entailment rules matching the specified patterns and identifies when these rules are executed. In this table, [X] represents the rule execution interval. The rules are executed in one or more of the following intervals: during the forward chaining process ($\mathcal{F}_t$ means that the rule is transformed, $\mathcal{F}_n$ means that the rule is applied without any transformation); during expanding the extensions ($\mathcal{E}$); or during the query answering process ($\mathcal{Q}$).

In the remainder of this section, we describe how to transform the original rules of p and D^* languages according to the extension-based knowledge model, and we describe the effects of these transformations. The transformed rules can be found in Tables 3 and 4.

The transformations of the D^* rules *rdfs9* (matches *P3*) and *rdfs7x* (matches *P4*), affect the reasoning process the most. The rule *rdfs9* infers a "type" relation between each individual of a class c and each superclass of c. A high percentage of the inferred "type" relations are derived by this rule. After rule transformation, this rule derives a relation between each extension of a class c and each superclass of c. Let n_I be the number of individuals of c, and let s_c be the number of superclasses of c, then after rule transformation, $n_I \times s_c$ (the number of "type" relations derived by the rule *rdfs9*) is reduced to s_c. Similarly, *rdfs7x* links each individual of a property p to each superproperty of p. After rule transformation, this rule derives a relation between each extension of a property p and each superproperty of p. Let n_t be the number of individuals of p, and let s_p be the number of superproperties of p; then, after rule transformation, $n_t \times s_p$ (the number of relations derived by the rule *rdfs7x*) is reduced to s_p.

The D^* rules *rdfs2* (matches *P7*) and *rdfs3* (matches *P7*) are the rules involving domains and ranges of properties. The rule *rdfs2*/*rdfs3* infers a "type" relation between each subject/object of a property individual and the domain/range class of that property. After rule transformation, this rule derives a relation between each subject/object extension of a property p and each domain/range class of p. Let n_t be the number of individuals of p, and let s_d/s_r be the number of domain/range classes of p. Then, after rule transformation, $n_t \times s_d/n_t \times s_r$ (the number of relations derived by the rule *rdfs2*/*rdfs3*) is reduced to s_d/s_r.

The D^* rules *rdfs4a*(matches *P3*) and *rdfs4b*(matches *P3*) derive that the subject and the object of every triple is an individual of the *Resource* class. In most cases, these rules double or triple the number of triples in the ontology. After rule transformation, these rules derive a relation between each subject and object extension of a property and the *Resource* class. Let n_s be the member count of the set containing the subjects of individuals of property p and let n_o be the member count of the set containing the objects of individuals of property p; then, after rule transformation, $n_s + n_o$ (the maximum number of relations derived from the rules *rdfs4a* and *rdfs4b*) is reduced to *2* (one for relating the subject extension to the *Resource* class and the other for relating the object extension to the *Resource* class).

rdfs6, *rdfs8* and *rdfs10* are the D^* rules matching *P2*. Transforming these rules does not affect the performance of the reasoning, but transformation is necessary to preserve the completeness and the soundness of the reasoning. The rules *rdfs6* and *rdfs10* derive that every concept is a subclass/subproperty of itself. The rule *rdfs8* derives that each class is a subclass of the *Resource* class. After applying the extension-based knowledge model transformation algorithm (see Section 3.2), the concepts differ from the other ontology resources in that each concept has either a class or a property extension. After rule transformation, *rdfs6* and *rdfs10* derive each ontology resource having a class/property extension as a subclass/subproperty of itself. The rule *rdfs8* derives that each ontology resource having a class extension is a subclass of the *Resource* class.

rdfs5 (matches *P1*) and *rdfs11* (matches *P1*) are the D^* rules involving the schema (TBox) of the ontology. These rules have nothing to do with individuals or extensions in the ontology. Therefore, these rules participate in the reasoning process as they are, without any transformation.

Table 5
Additional rules involving concept extensions.

$S_1 := \text{“}p\ has\text{”}(\text{“}Explicit\text{”}\mid\text{“}Inferred\text{”})G_1\ \text{“}Extension\ e \Rightarrow p\ has\text{”}\ G_1\ \text{“}Extension\ e\text{”}$
$G_1 := (\text{“}Class\text{”}\mid\text{“}Subject\text{”}\mid\text{“}Object\text{”}\mid\text{“}Property\text{”})$
$S_2 := \text{“}p\ subPropertyOf\ q \wedge p\ has\text{”}\ G_2\ \text{“}PropertyExtension\ e \Rightarrow q\ has\text{”}\ G_2\ \text{“}PropertyExtension\ e\text{”}$
$G_2 := (\text{“}Symmetric\text{”}\mid\text{“}Transitive\text{”}\mid\text{“}Functional\text{”}\mid\text{“}InverseFunctional\text{”}\mid\text{“}Inverse\text{”})$

Unlike the D^* entailments, the *p* entailment rules are related to the OWL language. The rules *rdfp12a*, *rdfp12b*, *rdfp12c*, *rdfp13a*, *rdfp13b* and *rdfp13c* are the *p* rules matching *P1*. They involve schema (TBox) of the ontology. Therefore, they participate in the reasoning process as they are, without any transformation.

The *p* rules, *rdfp9* (matches *P2*) and *rdfp10* (matches *P2*), derive that if a concept *x* is related to another resource *y* with the "*owl:sameAs*" predicate, then *x* is a sublass/subproperty of *y*. After rule transformation, *rdfp9* and *rdfp10* derive that if an ontology resource *x* has a class/property extension and *x* is related to another resource *y* with "*owl:sameAs*" predicate, then *x* is a sublass/subproperty of *y*. Transforming these rules does not affect the performance of the reasoning, but transformation is necessary to preserve the completeness and the soundness of the reasoning.

The rules *rdfp14bx*, *rdfp5a*, *rdfp5b*, *rdfp6*, *rdfp7* and *rdfp11* are the *p* entailment rules matching *P5* and *P8*. As a result, they are executed in the phase of *expanding the extensions* (see Section 4.2.4). The rules *rdfp14a*, *rdfp15* and *rdfp16* are the *p* entailment rules matching P6, which are executed in the phase of *query answering*.

The rules *rdfp1*, *rdfp2*, *rdfp3*, *rdfp4*, *rdfp8ax*, and *rdfp8bx* are the *p* entailment rules that are executed both in the *forward chaining* and in the *expanding the extensions* phases. These rules each involve OWL property characteristics. In the *forward chaining* process, some auxiliary relations about property characteristics are derived. These auxiliary relations are used in the phase of *expanding extensions* to help the computation of relations relying on property characteristics. The *p* entailment rule *rdfp1* derives that if a property *p* is a *FunctionalProperty*, and the subject denotes a resource that is the subject of two *p* individuals, then the objects of these *p* individuals have the same denotation (are equivalent); in other words, the objects with two different URIs denote one and the same resource. After transformation, this rule derives a *hasFunctionalPropertyExtension* relation between a *FunctionalProperty* and its property extension. The *p* entailment rule *rdfp2* derives that if a property *p* is an *InverseFunctionalProperty* and the object denotes a resource that is the object of two *p* individuals, then the subjects of these *p* individuals have the same denotation. After transformation, this rule derives a *hasInverseFunctionalPropertyExtension* relation between an *InverseFunctionalProperty* and its property extension.

The *p* entailment rule *rdfp3* derives new statements by switching the subject and the object of every *SymmetricProperty* individual. After transformation, this rule derives a *hasSymmetricPropertyExtension* relation between a *SymmetricProperty* and its property extension. The rule *rdfp4* derives that if a property *p* is a *TransitiveProperty*, and the object of an individual (t_1) of *p* is the subject of another individual (t_2) of *p*, then a new individual of property *p* is derived by linking the subject of t_1 and the object of t_2. After transformation, this rule derives a *hasTransitivePropertyExtension* relation between a *TransitiveProperty* and its property extension.

The *p* entailment rules *rdfp8ax* and *rdfp8bx* derive that if a property p_1 is *inverseOf* a property p_2, then a new individual of property p_2 is derived by switching the subject and the object of every individual of p_1, and vice versa. These rules are transformed to *rdfp8ax-I*, *rdfp8ax-II*, *rdfp8bx-I* and *rdfp8bx-II*. This new rule set:

- Links the property extensions of p_1 to property p_2 with *hasInversePropertyExtension*, and vice versa.
- Links the inverse property extensions of p_1 to property p_2 with *hasPropertyExtension*, and vice-versa.

4.1.2. Additional rules

The extension-based knowledge model extends the rules in Tables 3 and 4 with some additional rules (Table 5). After executing the eight rules derived by the grammar S_1 in Table 5, all extensions of a concept are accessible using only one query condition with a *"has(Class-Subject-Object-Property)Extension"* predicate. These rules reduce the number of conditions of the transformed pD^* rules and the rewritten queries in the *query answering process* (see Section 4.3).

The five rules derived by the grammar S_2 in Table 5 are used to ensure that the extensions of properties having characteristics are linked to their superproperties properly. These links are used in the phase of *expanding extensions* (see Section 4.2.4).

Table 6
Computing the final extensions of the example properties.

Property name	Extension definition
p_1	$sym(e_1 + e_2 + e_3 + tran(e_4) + e_5)$
p_2	e_2
p_3	$e_3 + tran(e_4) + e_5$
p_4	$tran(e_4)$
p_5	e_5

Table 7
SQL queries involving connector nodes.

$hasValue(C,p,v) \Rightarrow$	*SELECT T.subject* *FROM* $\delta(\Omega_p)$ *AS T* *WHERE T.object* $= v$
$someValuesFrom(C_X,p,C_Y) \Rightarrow$	*SELECT* T_2*.subject* *FROM* $\gamma(\Omega_{C_Y})$ *AS* T_1, $\delta(\Omega_p)$ *AS T* *WHERE* T_1*.resource* $= T_2$*.object*
$allValuesFrom(C_X,p,C_Y) \Rightarrow$	*SELECT* T_2*.subject* *FROM* $\gamma(\Omega_{C_Y})$ *AS* T_1, $\delta(\Omega_p)$ *AS* T_2 *GROUP BY* T_2*.subject* *HAVING object IN T*
$intersectionOf(C,C_1,\ldots,C_n) \Rightarrow$	$((\gamma(\Omega_{C_1})$ *INTERSECT* $\gamma(\Omega_{C_2}))\cdots)$ *INTERSECT* $\gamma(\Omega_{C_n})$
$unionOf(C,C_1,\ldots,C_n) \Rightarrow$	$((\gamma(\Omega_{C_1})$ *UNION* $\gamma(\Omega_{C_2}))\cdots)$ *UNION* $\gamma(\Omega_{C_n})$

Table 8
Number of class/property individuals in the example ontology.

Class name	Individual count	Property name	Individual count
ResearchGroup	2	*subOrganizationOf*	5
Department	3	*worksFor*	10
University	5	*degreeFrom*	10
Professor	10	*headOf*	3

4.2. Reasoning process

The reasoning process includes the following steps: (a) filtering triples, (b) applying the forward chaining algorithm, (c) processing the extensions, and (d) expanding the extensions. These steps are described in the following subsections.

4.2.1. Filtering triples

The aim of the forward chaining algorithm is to infer the statements about ontology schema and extension-concept relations. The forward chaining process makes the maximum possible inferences about instance data using extensions instead of using the instance data itself. Therefore, the ontology triples about individuals are filtered, and only the triples about ontology schema (and relations between concepts and their extensions) participate in reasoning. As a result, time consumption of the reasoning process remains fixed even if the size of the instance data increases.

4.2.2. Applying the forward-chaining algorithm

In this phase, we apply the forward chaining algorithm on the transformed schema of the ontology, using the transformed pD^* rules (Tables 3 and 4). At the end of the reasoning process, statements about concepts, extensions and the relations between them are inferred.

4.2.3. Processing the extensions

This phase is the prerequisite for the next phase (see Section 4.2.4). In this phase, we create the database tables, which store the property restrictions and property characteristics. These tables are used in the next phase. This phase includes the following steps:

- Step-I: storing data about property restrictions and set operators on class extensions,
- Step-II: storing data about property characteristics.

Table 9
Extensions of anonymous classes.

Extension name	In definition of	Anonymous class of extension
ε_{α_1}	*Chair*	$\alpha_1 = Person \cap \exists headOfDepartment$
ε_{α_2}	*Employee*	$\alpha_2 = Person \cap \exists worksForOrganization$
ε_{α_3}	*GraduateStudent*	$\alpha_3 = Person \cap \exists takesCourseGraduateCourse$
ε_{α_4}	*Student*	$\alpha_4 = Person \cap \exists takesCourseCourse$
ε_{α_5}	*Chair*	$\alpha_5 = \exists headOfDepartment$
ε_{α_6}	*Employee*	$\alpha_6 = \exists worksForOrganization$
ε_{α_7}	*GraduateStudent*	$\alpha_7 = \exists takesCourseGraduateCourse$
ε_{α_8}	*Student*	$\alpha_8 = \exists takesCourseCourse$

Table 10
PropertyRestrictionsOnExtensions table.

extensionUri	*restrictionType*	*onProperty*	*classOrValueUri*
ε_{α_5}	someValuesFrom	headOf	Department
ε_{α_6}	someValuesFrom	worksFor	Organization
ε_{α_7}	someValuesFrom	takesCourse	GraduateCourse
ε_{α_8}	someValuesFrom	takesCourse	Course

Table 11
SetOperatorsOnExtensions table.

extensionUri	*setOperator*	*listOfSetElements*
ε_{α_1}	intersection	Person, α_5
ε_{α_2}	intersection	Person, α_6
ε_{α_3}	intersection	Person, α_7
ε_{α_4}	intersection	Person, α_8

In the first step, we create and fill the *PropertyRestrictionsOnClassExtensions, SetOperatorsOnClassExtensions* and *ConstraintsOnClassExtensions* tables. These tables are defined as follows:

- **PropertyRestrictionsOnExtensions**: stores the extensions of classes, which are defined by property restrictions (*owl:someValuesFrom*, *owl:allValuesFrom* or *owl:hasValue*). This table includes the following fields: the URI of the extension (*extensionUri*), the type of restriction (*restrictionType*), the URI of the restricted property (*onProperty*), and the URI of the class or value, to which the range of property is constrained (*classOrValueUri*).
- **SetOperatorsOnExtensions**: stores the extensions of classes, which are constructed using the set operations (*owl:unionOf* and *owl:intersectionOf*). *SetOperatorsOnExtensions* includes the following fields: the URI of the extension (*extensionUri*), the name of the operator (*setOperator*) and the classes to which the set operator is applied (*listOfClasses)*.
- **ConstraintsOnClassExtensions**: stores all extensions of the classes, which are defined by property restrictions or set operations. *ConstraintsOnClassExtensions* includes the following fields: the URI of the extension (*extensionUri*), and the type of extension (*extensionType*), whose value may be either "*extensionWithPropertyRestriction*" or "*extensionWithSetOperator*."

In the second step, we create and fill the *PropertiesWithCharacteristics* table. The *PropertiesWithCharacteristics* includes the following fields: the URI's of properties with characteristics (*propertyUri*), and four boolean fields (*sym*, *tran*, *func*, and *invfunc*), whose values are determined according to the characteristics of the property.

4.2.4. Expanding the extensions

After the forward chaining process, the closure of the ontology and the instance data are moved to the database. For the sake of query performance, some implicit instance data and data about further types of extensions are computed in the database before the *query answering process*. In this phase, we compute this data in the following three steps:

- Step-I: expanding related extensions with inferred instance data relying on the *owl:hasValue* restriction,
- Step-II: expanding related property extensions with inferred instance data relying on transitive and symmetric property characteristics,
- Step-III: deriving *sameAs* relations relying on *Functional* and *InverseFunctional* properties.

In the first step, we process the inferred property individuals relying on the *owl:hasValue* property restriction. The inferred property individuals, which are derived using the *p* rule *rdfp14bx* (Table 25 in Appendix B) are added to the *statements* table as well as to the subject and object extensions of the related properties.

The second step involves expanding related property extensions with inferred instance data relying on transitive and symmetric property characteristics. A property usually has more than one extensions. To expand these extensions according to the

Table 12
ConstraintsOnClassExtensions table.

extensionUri	ε_{α_1}	ε_{α_2}	ε_{α_3}	ε_{α_4}	ε_{α_5}	ε_{α_6}	ε_{α_7}	ε_{α_8}
extensionType	set	set	set	set	restriction	restriction	restriction	restriction
Property name							Extension definition	
p_1							$sym(e_1+e_2+e_3+tran(e_4)+e_5)$	
p_2							e_2	
p_3							$e_3+tran(e_4)+e_5$	
p_4							$tran(e_4)$	
p_5							e_5	

Table 13
PropertiesWithCharacteristics table.

propertyUri	*transitive*	*symmetric*	*functional*	*inv-functional*
subOrganizationOf	+	–	–	–

Table 14
Data statistics for the LUBM and UOBM benchmarks.

	LUBM	UOBM
Number of classes	43 (22)	51 (41)
Number of Datatype properties	7 (3)	9 (5)
Number of Objecttype properties	25 (14)	34 (24)
Property individuals per university	90,000–110,000	210,000–250,000
Class individuals per university	8000–15,000	10,000–20,000

Table 15
Ontology loading times (ms) for standard and optimized inference engines with subsets of LUBM (1,0).

		Number of triples				
		21,729	41,828	62,062	81,752	100,881
Parsing	Standard	3756	4349	4847	6283	7893
	Optimized	1834	3439	4599	5774	7934
Transformation	Standard	–	–	–	–	–
	Optimized	1127	1776	2464	3267	4083
Reasoning	Standard	46,022	169,344	294,549	499,776	756,506
	Optimized	384	305	305	309	297
Total (Loading)	Standard	49,778	173,693	299,396	506,059	764,399
	Optimized	3345	5520	7368	9350	12,314

property characteristics properly, it is necessary to apply these characteristics on the union of these extensions instead of applying them on each extension separately. Assume that p is a transitive property having extensions $e_1 = \{a\ p\ b, c\ p\ d\}$ and $e_2 = \{d\ p\ e, f\ p\ g\}$. If we apply the transitivity characteristic on each extension separately, no new triple is derived. On the other hand, if we apply the transitivity characteristic on the union of these extensions ($e_1 \cup e_2 = \{a\ p\ b, c\ p\ d, d\ p\ e, f\ p\ g\}$), a new triple ($c\ p\ e$) is derived.

Here, we present an example to describe how to expand extensions using property characteristics. Fig. 5 shows five properties, p_1, p_2, p_3, p_4 and p_5, with extensions e_1, e_2, e_3, e_4 and e_5, respectively. The property p_1 is a symmetric property, and p_4 is a transitive property.

To correctly compute the effects of property characteristics on property extensions properly, the expanding process starts with the extensions of properties, which are at the bottom of the property hierarchy. Table 6 shows the final property extensions after applying the property characteristics. Let S be a triple set, p be a transitive property, t_1 and t_2 be two triples with predicate p, and the object of t_1 be the subject of t_2. Then, for each $t_1 \in S$ and $t_2 \in S$, *tran(S)* adds a new triple to S by linking the subject of t_1 and the object of t_2. Let S be a triple set, p be a symmetric property and t_1 be a triple with predicate p. Then, for each $t_1 \in S$, *sym(S)* adds a new triple to S by switching the subject and the object of t_1. After expanding the extensions process, the newly computed triples are added to the *statements* table, and the subjects and objects of these triples are added to the subject and object extensions of corresponding properties. The *sym(S)* and *tran(S)* characteristics are applied by exploiting triggers offered by DBMS. Thus, the order of applying these characteristics is not important, if the property is both transitive and symmetric.

The last step involves deriving *sameAs* relations relying on *Functional* and *InverseFunctional* properties using the *rdfp1* and *rdfp2* rules (see Table 25 in Appendix B). The inferred triples are stored in the *SameAs* table, which stores the identical individual pairs in its *individual1* and *individual2* fields.

4.3. Query answering process

The query[3] answering process involves building a query tree and creating the corresponding *SQL* query. The query tree has three kinds of nodes:

- ***Root Node***: each query tree has one and only one root node. The root node keeps information about constraints and relations between query conditions. The constraints of a condition involve the constants of the condition. The relations between conditions involve the common variables of these conditions.

[3] The inference engine accepts conjunctive queries that combine its conditions by conjunction. A condition is a triple in which each member of the triple (subject, predicate and object) may be an unbounded (free) or a bounded variable. The unbounded variables are distinguished by the "?" character occurring at the beginning of the variable name.

Table 16
Query execution times (ms) for standard (S) and optimized (O) inference engines with subsets of LUBM (1,0) (21,729 to 100,881 triples).

	21,729		41,828		62,062		81,752		100,881	
	S	O	S	O	S	O	S	O	S	O
Q1	67	1	89	2	81	3	86	3	122	4
Q2	61	3	124	11	162	21	222	37	273	56
Q3	20	1	55	1	64	1	74	1	89	1
Q4	114	193	309	337	379	486	413	721	555	926
Q5	376	102	1000	194	1025	321	1362	406	1651	474
Q6	44	165	100	366	131	512	192	746	267	839
Q7	76	95	183	213	210	323	273	466	379	593
Q8	1032	22	4870	43	8144	60	16,701	84	24,717	99
Q9	1344	13	7270	28	12,561	38	23,109	54	38,456	67
Q10	8	1	20	1	24	1	32	1	40	1
Q11	18	1	74	1	57	2	75	2	94	2
Q12	35	81	92	198	107	261	159	535	194	448
Q13	19	89	43	177	53	256	69	382	83	441
Q14	33	7	53	14	98	21	136	29	194	33

Table 17
Ontology loading times (ms) for standard and optimized inference engines with subsets of UOBM(1,0).

		Number of triples		
		54,605	106,285	157,922
Parsing	Standard	5241	10,452	16,722
	Optimized	4414	8314	14,131
Transformation	Standard	–	–	–
	Optimized	2558	5539	8983
Reasoning	Standard	61,009	244,877	649,904
	Optimized	1497	2210	2809
Total (Loading)	Standard	66,250	255,329	666,626
	Optimized	8469	16,063	25,923

- ***Resource Nodes***: each resource node (r) contains a query component (Ω_r), which contains a table called *memory*. The resource nodes are classified into three node groups according to their query components:
  Class Nodes: an *memory* of a class node query component stores the names of extensions, which belong to a particular class. OWL defines two types of classes: *named classes* and *anonymous classes*. Therefore, an *memory* of a class node stores either the names of *named class extensions* or the names of *anonymous class extensions*. If all of the extensions in the *memory* of a class node are *named class extensions*, then this node is a *basic class node*. Otherwise, this node is a *complex class node*.
  Property Nodes: an *memory* of a property node query component stores two kinds of information: (a) the names of extensions, which belong to a particular property p and (b) the relations (*hasPropertyExtension* or *hasInversePropertyExtension*) between

Table 18
Query execution times (ms) for standard (S) and optimized (O) inference engines with subsets of UOBM(1,0) (54,605 to 157,922 triples).

	54,605		106,285		157,922	
	S	O	S	O	S	O
Q1	124	5	222	15	320	14
Q2	31	128	60	243	90	452
Q3	249	768	520	1498	810	810
Q4	702	337	3568	805	7650	2181
Q5	62	1	110	1	170	2
Q6	156	2039	472	6520	850	12,323
Q7	93	676	454	1517	560	2007
Q8	62	701	512	2715	200	5937
Q9	109	21	233	45	320	114
Q10	31	1	50	1	90	2
Q11	1762	952	9027	1864	24,466	3105
Q12	2730	387	12,160	839	38,782	1218
Q13	62	622	93	2855	160	4890

Table 19
Ontology loading times (ms) for Jena, Pellet and the optimized inference engines with subsets of LUBM (1,0).

		Number of triples				
		21,729	41,828	62,062	81,752	100,881
Parsing	Optimized	1834	3439	4599	5774	7934
	Jena	2584	3261	4249	4608	5209
	Pellet	1922	2819	3347	5166	6128
Transformation	Optimized	1127	1776	2464	3267	4083
	Jena	–	–	–	–	–
	Pellet	–	–	–	–	–
Reasoning	Optimized	384	610	305	309	297
	Jena	37,339	170,569	187,072	223,678	364,500
	Pellet	1669	2018	2262	2967	9773
Total (Loading)	Optimized	3345	5825	7368	9350	12,314
	Jena	39,923	174,818	190,333	228,887	369,108
	Pellet	3591	4837	5609	8133	15,901

Table 20
The query answering performances of *Jena*, *Pellet* and the optimized inference engine.

	Number of triples														
	21,729			41,828			62,062			81,752			100,881		
	O	J	P	O	J	P	O	J	P	O	J	P	O	J	P
Q1	1	23	21	2	32	33	3	26	47	3	22	42	4	24	54
Q2	3	12,953	336,763	11	–	–	21	–	–	37	–	–	56	–	–
Q3	1	62	41	1	50	51	1	58	74	1	79	79	1	96	101
Q4	193	7539	12	337	17,801	13	486	23,615	22	721	33,497	21	926	43,776	28
Q5	102	78	39	194	162	62	321	252	98	406	362	113	474	388	133
Q6	165	5	53	366	10	82	512	15	140	746	14	160	839	18	178
Q7	95	5982	11,390	213	26,665	45,020	323	50,008	98,264	466	88,389	174,516	593	143,870	301,111
Q8	22	310	2967	43	1048	15,096	60	2440	30,899	84	4207	57,300	99	7140	119,173
Q9	13	577,557	1,109,700	28	–	–	38	–	–	54	–	–	67	–	–
Q10	1	22	47	1	36	101	1	48	190	1	67	190	1	83	233
Q11	1	24	7	1	120	13	2	336	20	2	642	21	2	1542	35
Q12	81	3	11	198	4	32	261	5	74	535	6	110	448	8	229
Q13	89	88	19	177	168	42	256	227	65	382	566	87	441	376	121
Q14	7	6	11	14	9	18	21	16	24	29	12	33	33	18	45

each property extension and property *p*. The OWL language does not provide for the use of anonymous properties. Therefore, all extensions in an *memory* of a property node belong to named properties.

Individual/Value Nodes: an *memory* of an individual/value node query component stores either an individual name or a value. If there are other individuals that are related to the individual via an "owl:sameAs" relation, then the names of these individuals are also stored in the *memory*.

- ***Connector Nodes***: for each *anonymous class extension* in a *complex class node* (η), a connector node is added to the children of η. Unlike resource nodes, connector nodes do not contain an *memory*. Connector nodes are classified into seven groups, according to the OWL construct that is used to identify the corresponding anonymous class (ς) in the parent node (η): *intersection nodes*, *union nodes*, *someValuesFrom nodes*, *allValuesFrom nodes*, *cardinality nodes*, *maxCardinality nodes*, *minCardinality nodes* and *hasValue nodes*.

Table 21
The ontology loading performances of optimized inference engine and *OWLIM*.

	1	5	10	20	50
Optimized	11,764	80,564	248,326	468,996	846,407
OWLIM	1000	21,000	66,000	125,000	239,000
Optimized/OWLIM	11.764	3.836381	3.762515	3.751968	3.541452

Table 22
The completeness of *Minerva* and *DLDB2* on the UOBM queries.

	Q1	Q2	Q3–8	Q9	Q10–12	Q13
DLDB2	100%	95%	100%	0%	100%	80%
Minerva	100%	100%	100%	100%	100%	61%

Table 23
Comparison of existing approaches and extension-based inference algorithm.

	SHER	Minerva	OWLDB	DLDB	Jena	Ext-based
Database schema	G	C	G	C	G	G
Summarization	+	−	−	−	−	+
Materialization	P	T	T	T	T	P
Paradigm	DL	DL + RB	RB	DL	RB	RB
Works in	M	DB	DB	M + DB	M	M + DB
Supported language	OWL DL	OWL DL	OWL DL	DAML + OIL	OWL Lite	pD^*

For each class, property, individual or value, which is referred to in the definition of ς, a class, a property or an individual/value node is added to the children of the connector node.

The components of the query tree are ranked in three layers: (a) *the first layer* contains the root node, (b) *the second layer* contains the direct children of the root node, and (c) *the third layer* contains all of the children of the nodes in the second layer. The query tree is constructed in three phases (initial phase, growth phase and final phase), which are described in the following subsections.

Table 24
The D^* entailment rules.

*rdfs*2 : p *domain* $u \wedge v\ p\ w \Rightarrow v$ *type* u *rdfs*3 : p *range* $u \wedge v\ p\ w \Rightarrow w$ *type* u
*rdfs*4*a* : $v\ p\ w \Rightarrow v$ *type* u *rdfs*4*b* : $v\ p\ w \Rightarrow w$ *type* u
*rdfs*5 : v *subPropertyOf* $w \wedge w$ *subPropertyOf* $u \Rightarrow v$ *subPropertyOf* u
*rdfs*6 : v *type Property* $\Rightarrow v$ *subPropertyOf* v *rdfs*7*x* : p *subPropertyOf* $q \wedge v\ p\ w \Rightarrow v\ q\ w$
*rdfs*8 : v *type Class* $\Rightarrow v$ *subClassOf Resource* *rdfs*9 : v *subClassOf* $w \wedge u$ *type* $v \Rightarrow u$ *type* w
*rdfs*10 : v *type Class* $\Rightarrow v$ *subClassOf* v
*rdfs*11 : v *subClassOf* $w \wedge w$ *subClassOf* $u \Rightarrow v$ *subClassOf* u

Table 25
The p entailment rules.

*rdfp*1 : p *type FunctionalProperty* $\wedge\ up\ v \wedge u\ p\ w \Rightarrow v$ *sameAs* w
*rdfp*2 : p *type InverseFunctionalProperty* $\wedge\ u\ p\ w \wedge v\ p\ w \Rightarrow u$ *sameAs* v
*rdfp*3 : p *type SymmetricProperty* $\wedge\ v\ p\ w \Rightarrow w\ p\ v$
*rdfp*4 : p *type TransitiveProperty* $\wedge\ u\ p\ v \wedge v\ p\ w \Rightarrow u\ p\ w$
*rdfp*5*a* : $v\ p\ w \Rightarrow v$ *sameAs* v *rdfp*5*b* : $v\ p\ w \Rightarrow w$ *sameAs* w
*rdfp*6 : v *sameAs* $w \Rightarrow w$ *sameAs* v *rdfp*7 : u *sameAs* $v \wedge v$ *sameAs* $w \Rightarrow u$ *sameAs* w
*rdfp*8*ax* : p *inverseOf* $q \wedge v\ p\ w \Rightarrow w\ q\ w$ *rdfp*8*bx* : p *inverseOf* $q \wedge v\ q\ w \Rightarrow w\ p\ v$
*rdfp*9 : v *type Class* $\wedge v$ *sameAs* $w \Rightarrow v$ *subClassOf* w
*rdfp*10 : p *type Property* $\wedge\ p$ *sameAs* $q \Rightarrow p$ *subPropertyOf* q
*rdfp*11 : $u\ p\ v \wedge u$ *sameAs* $u' \wedge v$ *sameAs* $v' \Rightarrow u'\ p\ v'$
*rdfp*12*a* : u *equivalentClass* $w \Rightarrow u$ *subClassOf* w
*rdfp*12*b* : u*equivalentClass* $w \Rightarrow w$ *subClassOf* u
*rdfp*12*c* : u *subClassOf* $w \wedge w$ *subClassOf* $u \Rightarrow u$ *equivalentClass* w
*rdfp*13*a* : v *equivalentProperty* $w \Rightarrow w$ *subPropertyOf* v
*rdfp*13*b* : v *equivalentProperty* $w \Rightarrow v$ *subPropertyOf* w
*rdfp*13*c* : v *subPropertyOf* $w \wedge w$ *subPropertyOf* $v \Rightarrow v$ *equivalentProperty* w
*rdfp*14*a* : v *hasValue* $w \wedge v$ *onProperty* $p \wedge u\ p\ w \Rightarrow u$ *type* v
*rdfp*14*bx* : v *hasValue* $w \wedge v$ *onProperty* $p \wedge u$ *type* $v \Rightarrow u\ p\ w$
*rdfp*15 : v *someValuesFrom* $w \wedge v$ *onProperty* $p \wedge u\ p\ x \wedge x$ *type* $w \Rightarrow u$ *type* v
*rdfp*16 : v *allValuesFrom* $w \wedge v$ *onProperty* $p \wedge u$ *type* $v \wedge u\ p\ x \Rightarrow x$ *type* w

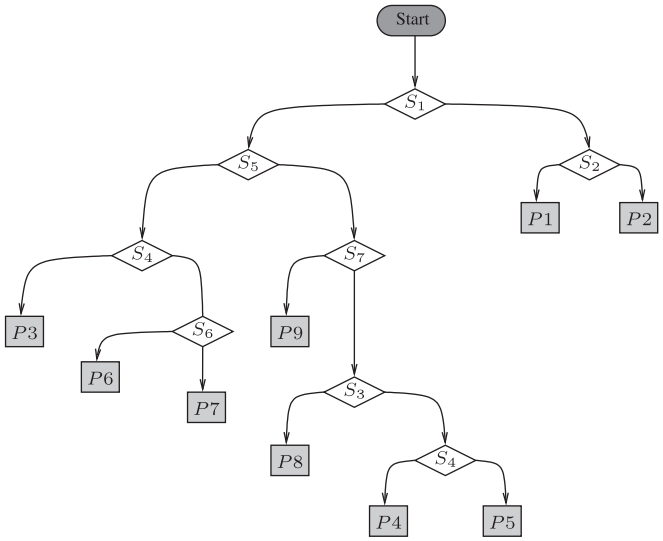

Fig. 4. Executing rules using patterns.

4.3.1. Initial phase

This phase constructs the nodes in the first and second layers. For each condition in the query, a class or a property node is created and added to the children of the root node in the following way:

- For each query condition with a *"type"* predicate (*type*(?*x*,*C*)), a class node is created and the *memory* of this node is filled with the results of the following query: *hasClassExtension*(*C*,?ε).
- For each query condition with a predicate other than "type" (?*p*(?*x*,*C*)), a property node is created and the *memory* of this node is filled with the results of the following queries:
  - *hasPropertyExtension*(*?p,?*ε): the relation between the property and the ?ε values are stored as *hasPropertyExtension* in the *memory*.
  - *hasInversePropertyExtension*(*?p,?*ε): the relation between the property and the ?ε values are stored as *hasInversePropertyExtension* in the *memory*.

4.3.2. Growth phase

The growth phase is about expanding the query tree until all leaf nodes are equal to either an individual/value node, a property node or a basic class node. Expanding a tree node (η) involves expanding anonymous class extensions in the *memory* of the node using *PropertyRestrictionsOnClassExtensions*, *SetOperatorsOnClassExtensions* and *ConstraintsOnClassExtensions* tables. For each anonymous class extension (ε) in the *memory*, a connector node is added to the children of η. The type of the connector node is read from the *restrictionType* field in the *RestrictionComplexExtensions* table or from the *setOperator* field in the *SetComplexExtensions* table.

If the connector node is an *intersection* or *union node*, then for each item in the *listOfSetElements* field in the *SetComplexExtensions* table, a new class node is added to the children of the connector node. For other types of connector nodes, a class node (read from the *classUri* field in the *RestrictionComplexExtensions* table) and a property node (read from the *onProperty* field in the *RestrictionComplexExtensions* table) are added to the children of the connector node. If the newly added child contains complex class nodes, then these children are expanded in the same way. The iterative node expansion algorithm continues until each leaf node is equal to an individual/value node, a property node or a basic class node. If a newly added child node (α) is equal

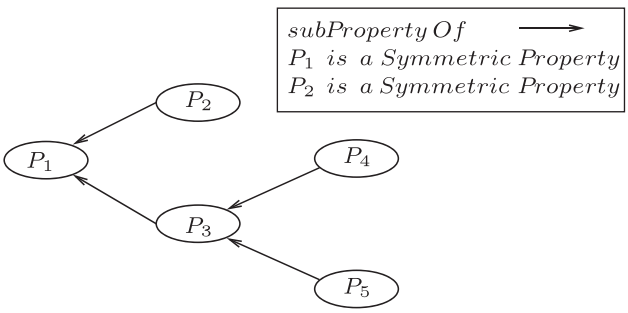

Fig. 5. An example property hierarchy.

to one its ancestors, then the **Termination_Method (α)** is triggered. This method prevents the addition of a child to the newly added α node by removing all anonymous class extensions in the *memory* of this node.

4.3.3. Final phase

This phase builds an *SQL* query using the query tree built in the growth phase. The building process starts with conjoining the computation results of the second layer nodes according to the constraints and relations specified in the root node. Second layer nodes are class or property nodes, which are computed by the following methods:

- If the node is a basic class node (Ω_C), then the $\gamma(\Omega_C)$ function computes basic class extensions in *memory* of the Ω_C node and unifies the results. Each basic class extension (ε_{basic}) is computed with the following *SQL* query:
  - $Q_{\varepsilon_{basic}}$:*SELECT resource FROM contains WHERE extension* $=\varepsilon_{basic}$
- If the node is a property node (Ω_p), then the $\delta(\Omega_p)$ function computes the property extensions in *memory* of the Ω_p node and unifies the results. Each extension, which is related to the property with a *hasPropertyExtension* predicate, is computed via $Q_{\varepsilon_{p+}}$. Each extension, which is related to the property with a *hasInversePropertyExtension* predicate, is computed via $Q_{\varepsilon_{p-}}$.
  $Q_{\varepsilon_{p+}}$:*SELECT subject, object FROM statements WHERE predicate* $=\rho$
  $Q_{\varepsilon_{p-}}$:*SELECT subject AS object, object AS subject FROM statements WHERE predicate* $=\rho$
- If the node is a complex class node, then each child of the node is computed and the results are unified with computations of the basic class extensions in the *memory* (using $Q_{\varepsilon_{basic}}$). Each child node of the complex class node is a connector node, which is computed using the corresponding *SQL* query (S_1) in Table 7. If S_1 requires the computation results of basic class nodes or property nodes, these nodes are computed using $Q_{\varepsilon_{basic}}$, $Q_{\varepsilon_{p+}}$ and $Q_{\varepsilon_{p-}}$. If S_1 requires the computation result of a complex class node, then a corresponding *SQL* query (S_2) in Table 7 is created and nested in query S_1. This nesting process continues iteratively until there is no complex class node to compute.

5. Running example

This section describes the extension-based inference algorithm using an example ontology, which is a subset of the well-known LUBM (Lehigh University Benchmark) [27] ontology schema.

Fig. 6 shows the schema of the example ontology. In addition to the information in the figure, it is also necessary to note that *subOrganizationOf* is a transitive property and *hasAlumnus* is an inverse property of *degreeFrom*. Table 8 shows the numbers of class/property individuals.

The following subsections describe how the extension-based inference algorithm is applied to the example ontology.

5.1. Transforming an example ontology into the extension-based ontology model

After transforming the example ontology into its equivalent in the extension-based knowledge model, we derive the following 16 triples about extension-concept relations: **(*)** four *hasExplicitClassExtension* relations for classes in Table 8 **(*)** four *hasExplicitSubjectExtension* relations, four *hasExplicitObjectExtension* relations and four *hasExplicitPropertyExtension* relations for properties in Table 8. After transformation, a maximum of 76 *contains* relations are derived: **(*)** 20 *contains* relations for 20 $(2+3+5+10)$ class individuals (Table 8); **(*)** a maximum of 28 *contains* relations for relating the subjects of 28 $(5+10+10+3)$ property individuals (Table 8) to the corresponding subject extensions (if there are n individuals of property p having the same subjects, then the number of *contains* relations to be added is reduced by $n-1$); and **(*)** a maximum of 28 *contains* relations for relating the objects

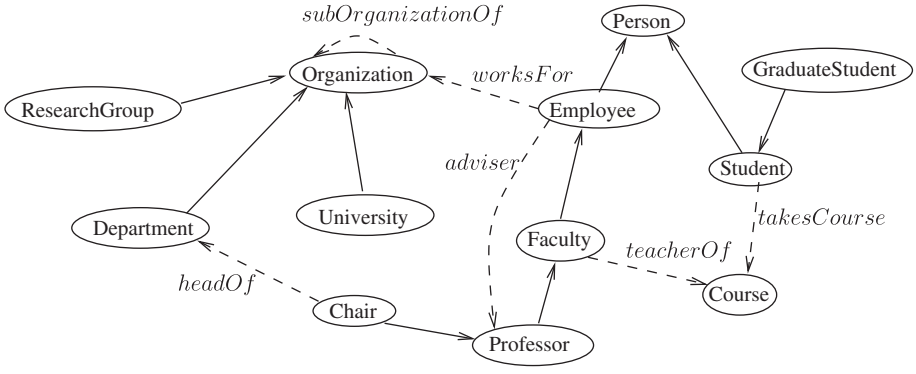

Fig. 6. Example ontology schema.

of 28 property individuals to the corresponding object extensions. If there are n individuals of property p having the same objects, then the number of *contains* relations to be added is reduced by $n-1$.

5.2. Filtering triples of the example ontology and the forward chaining process

Filtering triples of the example ontology prevents instance data from participating in reasoning. Let η_S be the number of ontology schema triples; then, without applying the model, the number of triples participating in reasoning is $\eta_S+20+28$ (20 class individuals, 28 property individuals). After applying the model, only triples of ontology schema (η_S) and extension-concept relations participate in reasoning. Before applying the extension-based knowledge model, 71 triples are inferred with the example ontology. After applying the extension-based knowledge model, this number is reduced to 11. The utility of the model is in direct proportion to the ratio of instance data. Even with a small amount of instance data in the example ontology, the inferred triples are reduced by 84.5%.

5.3. Processing the extensions in the example

In Step-I and Step-II (see Section 4.2.3), data about property restrictions, set operators and property characteristics are stored in the corresponding database tables, as shown in Tables 10–13. Table 9 shows the definitions of anonymous classes, their extensions and the classes, whose definition refers to these anonymous classes.

5.4. Expanding the extensions in the example

Step-I (see Section 4.2.4) makes no change because there is no anonymous class defined using the *owl:hasValue* property restriction in the example ontology. In Step-II (see Section 4.2.4), the following triples are added to the *statements* table {*rg01 subOrganizationOf univ01, rg02 subOrganizationOf univ01*}. The subjects and objects of these triples are also added to the subject/object extensions of the *subOrganizationOf* property, by adding the proper fields to the *contains* table. Step-III (see Section 4.2.4) makes no change because there is no functional or inverse functional property in the example ontology.

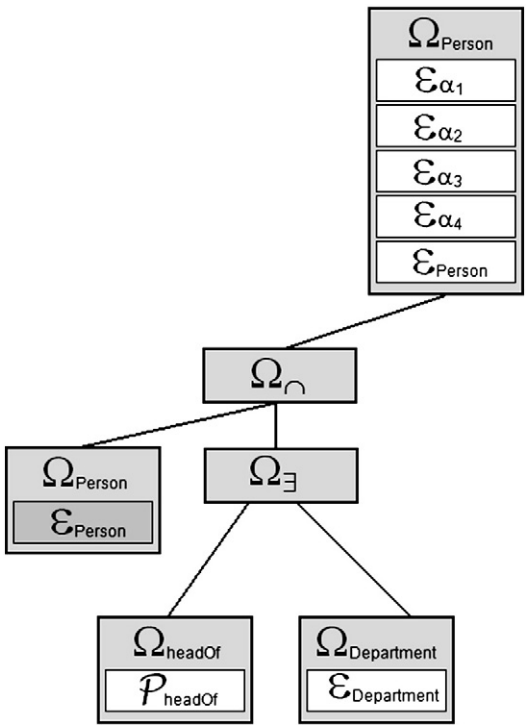

Fig. 7. The query tree after expanding ε_{α_i}.

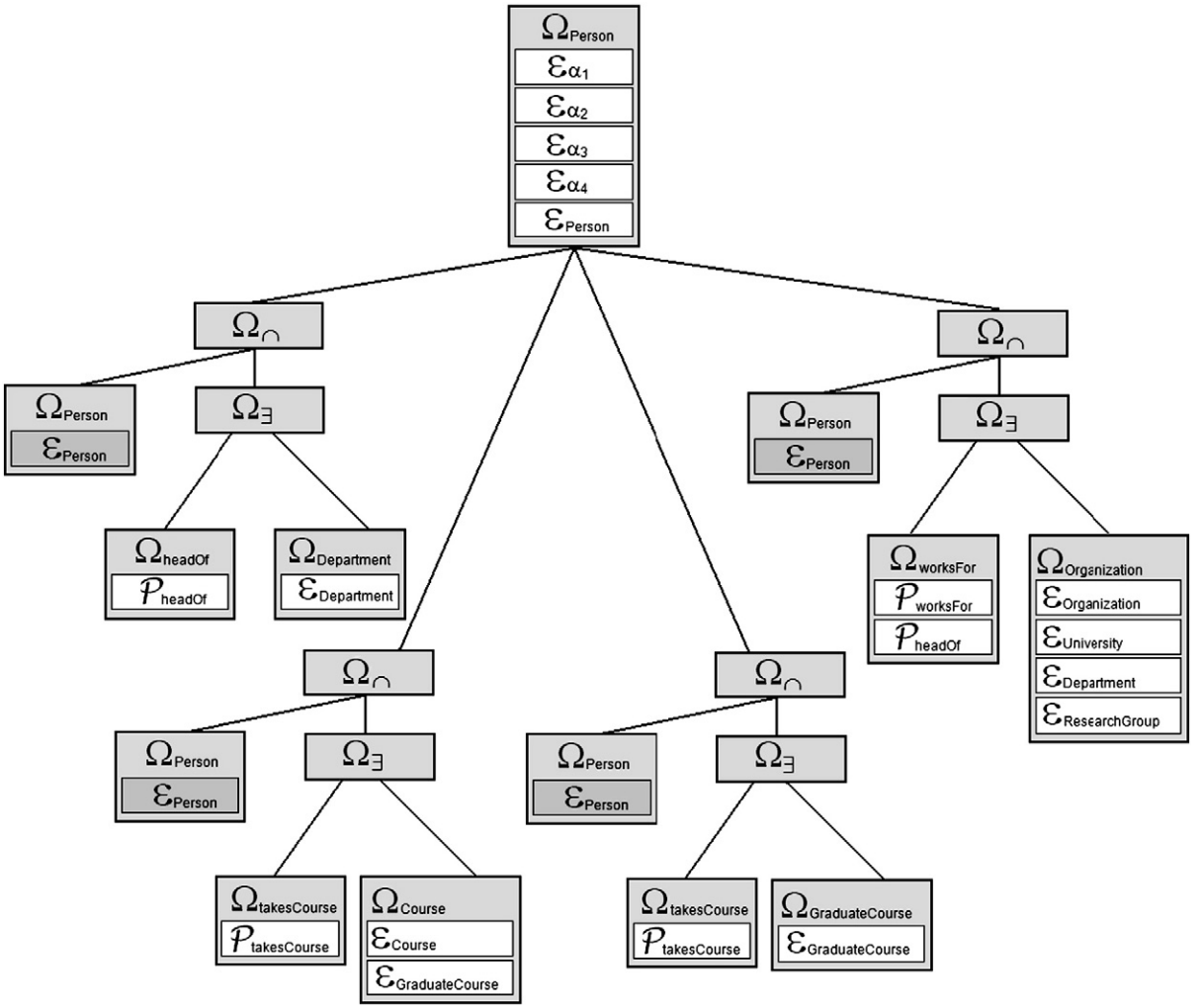

Fig. 8. The final state of the query tree (*Example Query 1*).

5.5. Example queries

To exemplify the query answering process, we use the *Example Query 1:* (*?X type Person*). The query has only one condition with a "type" predicate, and therefore only one class node is added to the children of the root node. There is no information added to the root node about the constraints and relations between query conditions.

The initial phase is completed by filling the *memory* of the class node with the extensions of the *Person* class (ε_{Person}, ε_{α_1}, ε_{α_2}, ε_{α_3}, ε_{α_4}). The extensions are obtained using the following query: *hasClassExtension(Person,?ε)*.

In the growth phase, the children of the anonymous class extensions in the *memory* of the node (ε_{α_1}, ε_{α_2}, ε_{α_3}, ε_{α_4}) are added to the query tree. The first anonymous class extension is ε_{α_1}, which belongs to the anonymous class α_1. The class α_1 is defined as $Person \cap \{\exists headOf\ Department\}$. Fig. 7 shows the query tree after expanding the children of ε_{α_1}.

The children of the connector node $\Omega_{\cap}$ contain the class node Ω_{Person}, which is equal to one of its ancestors. Therefore, the ***Termination Method*** (Ω_{Person}) is triggered, and all anonymous class extensions in the *memory* of the newly added Ω_{Person} node are removed. Expansion of the tree ends when all leaf nodes equal an individual/value node, a property node or a basic class node (Fig. 8).

The example query has only one condition; therefore, the corresponding *SQL* query (*S-Ex1*) unifies the computations of the children of Ω_{Person} with the results of the basic class extensions in the *memory* of Ω_{Person} in the following way:

- $S-Ex1 : Q_{\varepsilon_{Person}}$ *UNION* Q_1 *UNION* Q_2 *UNION* Q_3 *UNION* Q_4

$Q_{\varepsilon_{Person}}$ computes the only basic class extension (ε_{Person}) in the *memory* as follows:

- $Q_{\varepsilon_{Person}}$*Z:SELECT resource FROM contains WHERE extension* $= \varepsilon_{Person.}$

Q_1 (SQL 1), Q_2 (SQL 2), Q_3 (SQL 3), and Q_4 (SQL 4) compute the complex class extensions ε_{α_1}, ε_{α_2}, ε_{α_3}, and ε_{α_4}, respectively. ε_{α_1} involves the computation of a connector node $\Omega_{\cap}$. One of the children (Ω_{Person}) of the connector node is terminated; therefore, the *SQL* query of this node is reduced to $Q_{\varepsilon_{Person}}$. The other child ($\Omega_{\exists}$) is converted to *SQL* using Table 7 (SQL 1). One of the children of the $\Omega_{\exists}$ node is a property node, whose *memory* is filled with the extensions, which are related to the property via a *hasPropertyExtension* or a *hasInversePropertyExtension* predicate.

SQL 1 *SQL* query *Q1*

```
(SELECT resource FROM contains
WHERE extension = explicit class extension of Person)
INTERSECT
(SELECT T2.subject FROM
(SELECT resource FROM contains
WHERE extension = explicit class extension of Department) AS T1,
(SELECT subject,object FROM statements WHERE predicate = headOf) AS T2
WHERE T1.resource = T2.object)
```

SQL 2 *SQL* query *Q2*

```
(SELECT resource FROM contains
WHERE extension = explicit class extension of Person)
INTERSECT
(SELECT T2.subject FROM
(SELECT resource FROM contains
WHERE extension = explicit class extension of Organization
OR extension = explicit class extension of University
OR extension = explicit class extension of Department
OR extension = explicit class extension of ResearchGroup) AS T1,
(SELECT subject,object FROM statements
WHERE predicate = worksFor OR predicate = headOf) AS T2
WHERE T1.resource = T2.object)
```

SQL 3 *SQL* query *Q3*

```
(SELECT resource FROM contains
WHERE extension = explicit class extension of Person)
INTERSECT
(SELECT T2.subject FROM
(SELECT resource FROM contains
WHERE extension = explicit class extension of GraduateCourse) AS T1
(SELECT subject,object FROM statements WHERE predicate = takesCourse) AS T2
WHERE T1.resource = T2.object)
```

SQL 4 *SQL* query *Q4*

```
(SELECT resource FROM contains
WHERE extension = explicit class extension of Person)
INTERSECT
(SELECT T2.subject FROM
(SELECT resource FROM contains
WHERE extension = explicit class extension of Course
OR extension = explicit class extension of GraduateCourse) AS T1
(SELECT subject,object FROM statements WHERE predicate = takesCourse) AS T2
WHERE T1.resource = T2.object)
```

Example Query 2 is as follows:(?*X type Chair*) ∧(?*Y type Department*) ∧(?*X worksFor*?*Y*)∧(?*Y subOrganizationOf* "http://www.University0.edu"). This query has multiple conditions; therefore, the root node keeps information about constraints and relations between query conditions. In this query, the subjects of the first and third conditions, the subjects of the second and fourth conditions, and the object of the third condition are the same. The object of the fourth condition is a constant ("http://www.University0.edu").

The root node has four children nodes, including two class (Ω_{Chair} and $\Omega_{Department}$) and two property nodes ($\Omega_{worksFor}$ and $\Omega_{subOrganizationOf}$). One of these class nodes (Ω_{Chair}) is a complex class node. Fig. 9 shows the final state of the query tree. The expansion of the (Ω_{Person}) node is shown in Fig. 8; thus, it is not repeated here. The corresponding *SQL* query of *Example Query 2* (*S-Ex2*) is given in SQL 5. Query *S-Ex1* is given in the previous example and is also not repeated here.

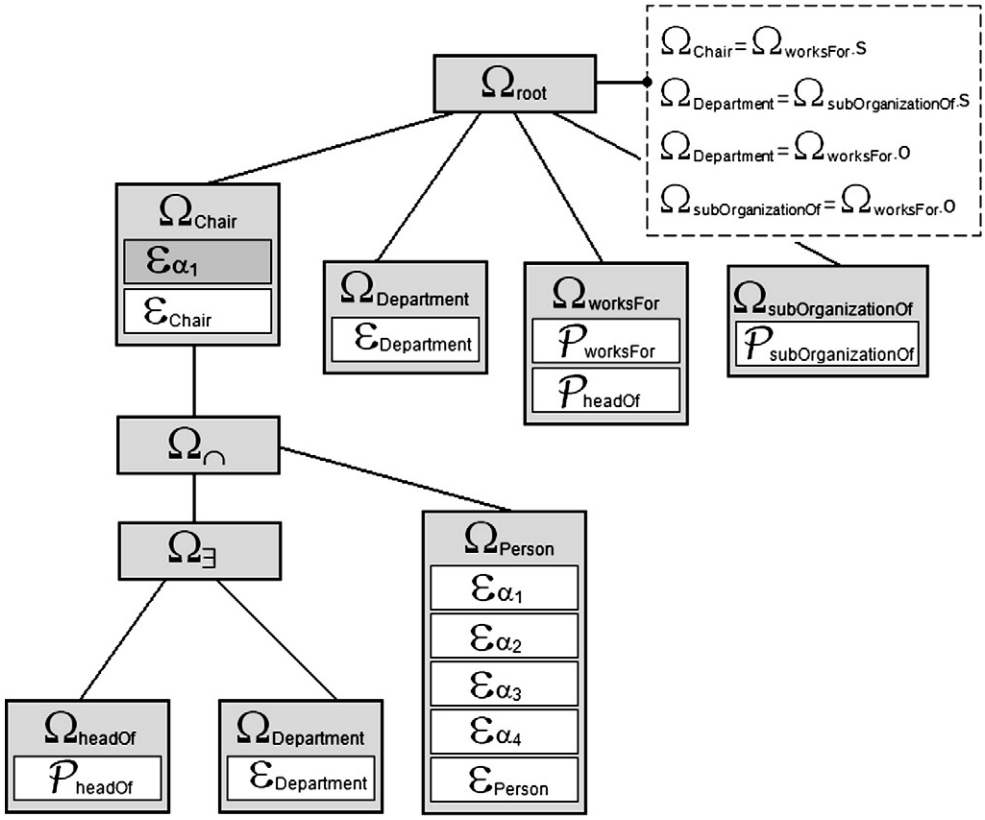

Fig. 9. The final state of the query tree (*Example Query 2*).

SQL 5 *SQL* query *Q5*

```
SELECT T1.subject
FROM
((S-Ex1 INTERSECT (SELECT T1.subject FROM
(SELECT resource FROM contains
WHERE extension = explicit class extension of Department) AS T0,
(SELECT subject,object FROM statements WHERE predicate = headOf) AS T1
WHERE T0.resource = T1.object))
UNION
(SELECT resource FROM contains
WHERE extension = explicit class extension of Chair)) AS T2),
(SELECT resource FROM contains
WHERE extension = explicit class extension of Department) AS T3,
(SELECT subject,object FROM statements
WHERE predicate = worksFor OR predicate = headOf) AS T4,
(SELECTsubject,object FROM statements
WHERE predicate = subOrganizationOf) AS T5
WHERE T2.resource = T4.subject
AND T3.resource = T5.subject AND T3.resource = T4.object
AND T5.subject = T4.object
```

6. Complexity analysis

We use the RETE algorithm to implement our forward chaining inference. RETE takes $O(n_f \times f \times c)$ time per inference iteration where n_f is the number of forward chaining rules, f is the number of facts and c is the average number of conditions of the forward chaining rules [28]. The total time for a forward chaining inference is $O(n_f \times c \times (f_1 \times S + f_2 \times (S-1) + \ldots + f_S))$, where S is the total number of states and f_i is the newly available data items in each state in the composition schema.

The extension-based reasoning algorithm increases the performance of the reasoning process by reducing the number of facts participating in reasoning and the number of facts inferred in each cycle.

Let $\mathcal{O}$ be an ontology, $n(\mathcal{O})$ be the number of triples, $\mathcal{R}_{\mathcal{O}}^{type}$ be the number of *type* relations, and $\mathcal{R}_{\mathcal{O}}^{\mathcal{P}}$ be the number of instances of all of the properties except for the *type* property; then,

$$n(\mathcal{O}) = \mathcal{R}_{\mathcal{O}}^{type} + \mathcal{R}_{\mathcal{O}}^{\mathcal{P}}.$$

After transformation, every *type* relation between a class and its individual is replaced with a *contains* relation between the *classExtension* of the class and the individual. As a result, the number of *contains* relations in the transformed ontology is equal to $\mathcal{R}_{\mathcal{O}}^{type}$. Let $\mathcal{C}_{\mathcal{O}}$ be the number of classes in the ontology. Because a *hasExplicitClassExtension* relation is added to every class, we add $\mathcal{C}_{\mathcal{O}}$ new triples to the transformed ontology. Then, for every instance of a property other than *type* (e.g. $\mathcal{A}(\mathcal{U},\mathcal{B})$), we add two new relations to the ontology (*contains*($\mathcal{S}_{\mathcal{E}\mathcal{A}},\mathcal{U}$), *contains*($\mathcal{O}_{\mathcal{E}}\_\mathcal{A},\mathcal{B}$)). As a result, the number of instances of a property other than *type* ($\mathcal{R}_{\mathcal{O}}^{\mathcal{P}}$) is tripled ($3\times\mathcal{R}_{\mathcal{O}}^{\mathcal{P}}$) in the transformed ontology. Let $\mathcal{P}_{\mathcal{O}}$ be the number of properties in the ontology. Because three relations are added (*hasExplicitSubjectExtension*, *hasExplicitObjectExtension* and *hasExplicitPropertyExtension*) to every property that is not a *type* property, we add $3\times\mathcal{P}_{\mathcal{O}}$ triples to the transformed ontology. The transformation overhead is *O(f)*, where *f*, namely $n(\mathcal{O})$, is the number of facts in the ontology (see Algorithm 1).

Let $\mathcal{O}'$ be the transformed version of ontology $\mathcal{O}$ and let Δ_1 be the increase in the number of triples after transformation, i.e., the difference between $n(\mathcal{O})$ and $n(\mathcal{O}')$; then,

$$\begin{aligned} n(\mathcal{O}') &= \mathcal{R}_{\mathcal{O}}^{type} + \mathcal{C}_{\mathcal{O}} + 3\times\mathcal{R}_{\mathcal{O}}^{\mathcal{P}} + 3\times\mathcal{P}_{\mathcal{O}} \\ \Delta_1 &= \mathcal{C}_{\mathcal{O}} + 2\times\mathcal{R}_{\mathcal{O}}^{\mathcal{P}} + 3\times\mathcal{P}_{\mathcal{O}} \end{aligned}.$$

After transformation, only a small number of triples participates in reasoning. These triples contain the schema knowledge of the ontology and contain the relations between concepts and their extensions ($\mathcal{C}_{\mathcal{O}} + 3\times\mathcal{P}_{\mathcal{O}}$). The number of these triples (f') are computed as follows, where f_{ABox} is the number of triples from individual data, and *f* is the number of facts participating in forward chaining before applying the extension-based inference algorithm:

$$f' = f - f_{ABox} + \mathcal{C}_{\mathcal{O}} + 3\times\mathcal{P}_{\mathcal{O}}.$$

We still make reasoning on the schema but we prevent ABox triples to participate in reasoning. Because the schema level reasoning is not affected by the transformation, the decrease in the number of inferred triples equals the decrease in the number of inferred triples about individuals. Additional facts about *grouping predicates* are derived, but these derivations are small enough to be negligible. The number of these additional facts ($n(f_{groupingPred})$) is computed as follows, where $n(f_{subclassOf})$ is the number of inferred subclass relations, $n(f_{subpropertyOf})$ is the number of inferred subproperty relations, $n(f_{functionalProperties})$ is the number of functional properties, $n(f_{inverseFunctionalProperties})$ is the number of inverse functional properties, $n(f_{symmetricProperties})$ is the number of symmetric properties, $n(f_{transitiveProperties})$ is the number of transitive properties, and $n(f_{inverseProperties})$ is the number of properties that are the subject or object of the "owl:inverseOf" relations:

$$\begin{aligned} n\left(f_{groupingPred}\right) = {} & n\left(f_{subclassOf}\right) + 3\times n\left(f_{subpropertyOf}\right) + n\left(f_{functionalProperties}\right) + n\left(f_{inverseFunctionalProperties}\right) + n\left(f_{symmetricProperties}\right) \\ & + n\left(f_{transitiveProperties}\right) + n\left(f_{inverseProperties}\right). \end{aligned}$$

In a nutshell, the transformation leads to an increase in the triple count. However, only the ontology schema triples participate in reasoning, and, as a result, the memory and time consumption of reasoning decreases dramatically. Moreover, the memory and time consumption of the reasoning and the number of inferred triples remain fixed, even when the size of the instance data increases. The utility of the approach increases with increasing amounts of instance data.

7. Evaluation

We evaluated the performance of the extension-based reasoning algorithm using the LUBM (Lehigh University Benchmark) [27] and UOBM (University Ontology Benchmark) [29] benchmarks. Table 14 shows the data statistics for these benchmarks [29]. The number of classes and properties used to define ABox are denoted in the bracket. Some classes and properties are used only to define class and property hierarchies in TBox and are not used to restrict individuals directly. The experiments were conducted on a laptop computer with 2 GB RAM and a Core 2 Duo T7700@2.4 GHz processor. The software configuration was as follows: Windows Vista Business 32 bit operating system. The .NET runtime environment version 2.0 and Sqlite database management system was used. Our standard inference engine is a Rete [30] based inference engine. The optimized inference engine is a version of the standard inference engine, which applies the extension-based inference algorithm.

We evaluated the inference algorithm using the following metrics: (a) **Building time**: the amount of time required to read an ontology file and to transform the file to the reasoner's data model in memory; (b) **Loading time**: the amount of time required to load an ontology. We define the term "ontology loading" as the total process of parsing, extension-based model transformation, inference and property characteristics materialization. The three factors that affect ontology loading performance are the number of triples participating in the inference (η_{tpi}), the number of inferred triples (η_{it}) and the number of total triples (η_t). The number of triples loaded per second (η_{tps}) is another metric to evaluate the loading performance [18]; (c) **Space consumption**: the space consumption involves the memory (η_m) and the disk sizes (η_{db}) that the inference algorithm uses. The factors that affect the space consumption are the

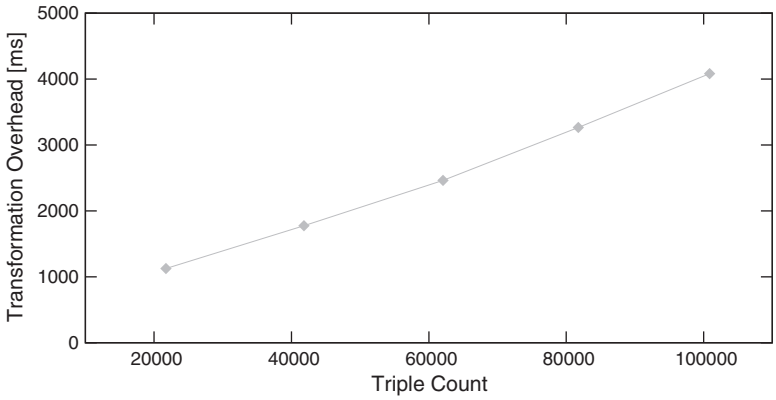

Fig. 10. The linear increase in the transformation overhead with increasing sizes of instance data (up to LUBM(1,0)).

number of triples participating in the inference (η_{ltpi}), the number of inferred triples (η_{it}), and the number of total triples (η_t); and (d) **Query execution time**: a metric to evaluate the query performances is the query execution time per result (q_{eps}).

7.1. Experiment 1

This experiment compares the standard and optimized inference engines using subsets of the LUBM(1,0) data set, which is the largest data set that the standard inference engine can process. The loading times are given in Table 15. The optimized inference engine performs better because both building and reasoning times are reduced after optimization. The building time is reduced because not all of the triples, but only triples that participate in reasoning, are transformed to the data model in memory. The transformation

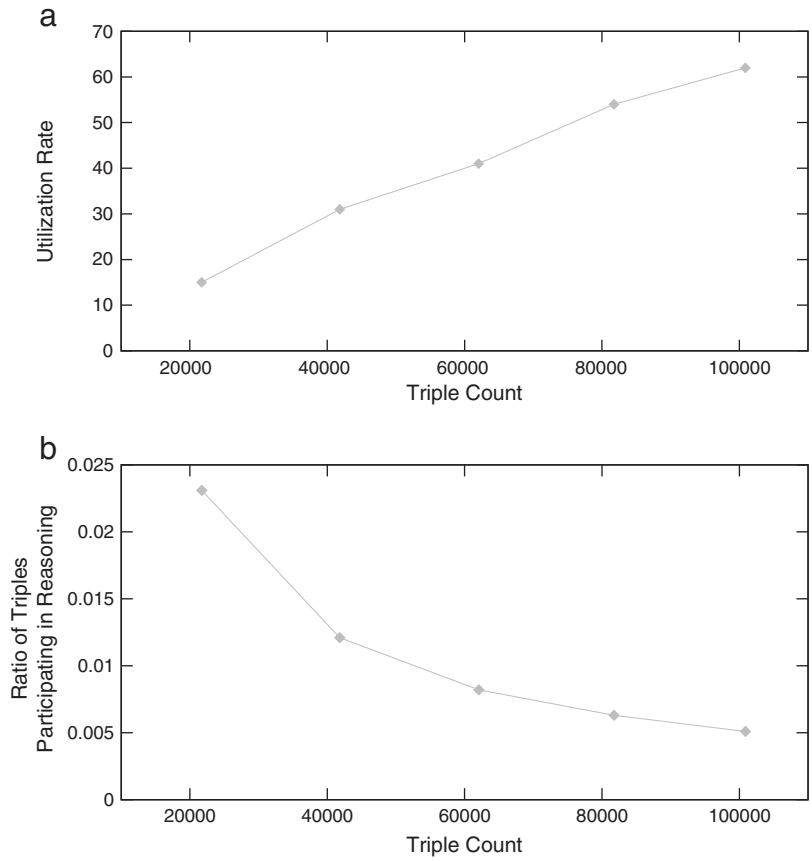

Fig. 11. (a) Utilization rate in ontology loading time with increasing sizes of instance data (up to LUBM(1,0)). (b) The ratio of the number of triples participating in reasoning with increasing sizes of instance data (up to LUBM(1,0)).

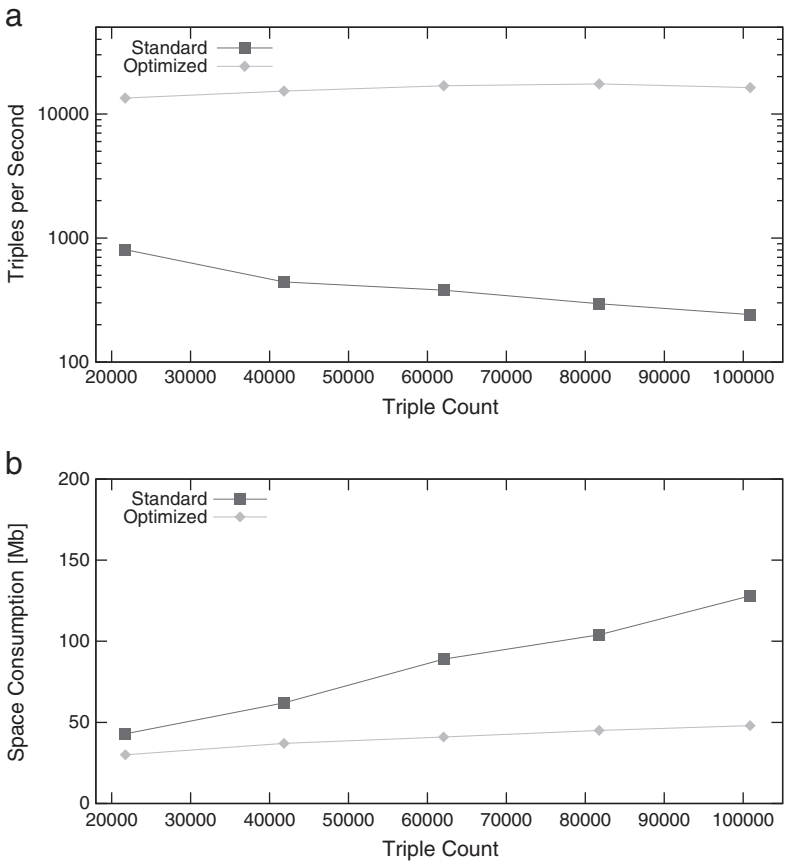

Fig. 12. (a) The number of triples loaded per second for the standard and optimized inference engines with increasing sizes of instance data (up to LUBM(1,0)). (b) Space consumption of standard and optimized inference engines with increasing sizes of instance data (up to LUBM(1,0)).

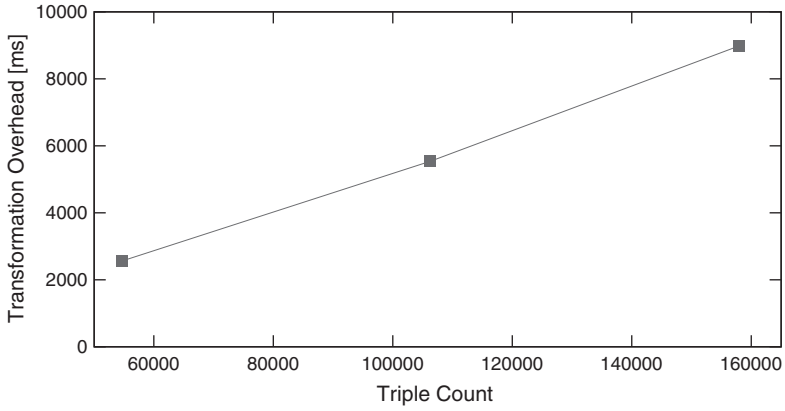

Fig. 13. The linear increase of transformation overhead with increasing sizes of instance data (up to UOBM(1,0)).

overhead is presented in Fig. 10. The triples with ontology individuals do not participate in reasoning; as a result the reasoning time is also reduced. The number of triples participating in reasoning and the time consumption of the reasoning process remain fixed, even if the size of the instance data increases. The utilization rate for the loading time[4] increases with the size of the instance data (Fig. 11(a)). Fig. 11(b) shows the ratio of the number of triples participating in reasoning to the number of total triples.

[4] The utilization rate is the ratio of the loading time of the standard inference engine to the loading time of the optimized inference engine.

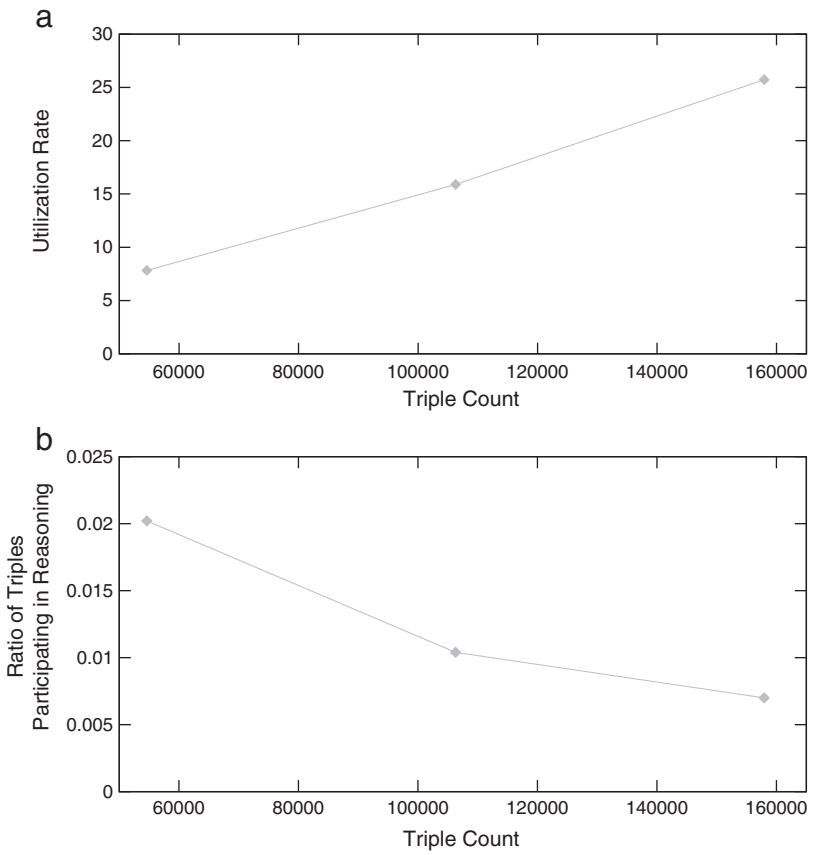

Fig. 14. (a) Utilization rate in ontology loading time with increasing sizes of instance data (up to UOBM(1,0)). (b) The ratio of the number of triples participating in reasoning with increasing sizes of instance data (up to UOBM(1,0)).

Fig. 12(a) shows the number of triples loaded per second (η_{tps}) as a function of the data size. In the standard inference engine, the value of η_{tps} is reduced significantly with increasing amounts of instance data. In contrast, in the optimized inference engine, the value of η_{tps} remains almost fixed. The space consumption is also improved by up to 60% in the optimized inference engine (Fig. 12(b)).

Table 16 shows the effect of the algorithm on the performances of 14 LUBM queries (*Q1–Q14* in Table 26 in *Appendix* C) with subsets of LUBM (1,0) (21,729 triples–100,881 triples). As the table shows, nine queries are positively affected, and five queries are negatively affected. Because we use backward chaining reasoning, a decrease is expected in performances of all queries. Using the extension-based knowledge model and the database reasoning results in a general improvement in query performances. The optimized algorithm performs worse in some cases due to non-optimized *SQL* queries. We think that our query performance can still be improved with some pruning algorithms on the query tree or by using a database system, which has a built-in query optimizer (see Section 9).

7.2. Experiment 2

This experiment compares the standard and optimized inference engines using a more complex ontology schema. As a standard OWL ontology benchmark, the LUBM has two limitations. First, the LUBM does not completely cover either OWL Lite or OWL DL inference. For example, inference on the cardinality and *allValuesFrom* restrictions cannot be tested by the LUBM. In fact, the inference supported by this benchmark is only a subset of OWL Lite. Some real ontologies are more expressive than the LUBM ontology. The UOBM extends the LUBM by adding extra TBox axioms, making use of all of OWL Lite and OWL DL. Second, the extended benchmark generates instance data sets in a more reasonable way. The necessary links between individuals from different universities make the test data form a connected graph rather than multiple isolated graphs. This will guarantee the effectiveness of scalability testing.

In the UOBM test data, every university contains 15 to 25 departments, each described by a separate OWL file. The generated instance data may form multiple relatively isolated graphs and may lack necessary links between the graphs. More precisely, the benchmark generates individuals (such as departments, students and courses) at a university as a basic unit. Individuals from a university do not have relations with individuals from other universities.

We use subsets of the UOBM(1,0) data set, which are the largest data sets that the standard inference engine can process. The UOBM benchmark includes two distinct datasets for OWL-Lite and OWL-DL. The complexity of pD^* is between that of RDFS and OWL Lite; therefore, we use the UOBM dataset for OWL-Lite in this experiment. Both standard and optimized inference engines gave *100%* complete answers to 13 UOBM queries in Table 26 in Appendix C. The performance results of this experiment are

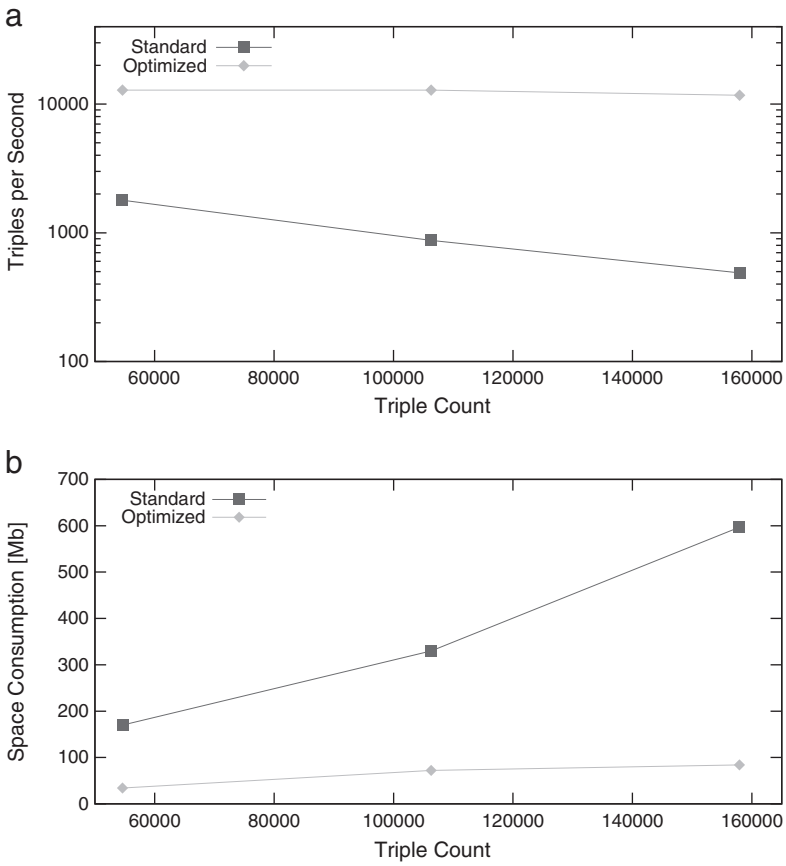

Fig. 15. (a) The number of triples loaded per second for the standard and optimized inference engines with increasing sizes of instance data (up to UOBM(1,0)). (b) Space consumption of standard and optimized inference engines with increasing sizes of instance data (up to UOBM(1,0)).

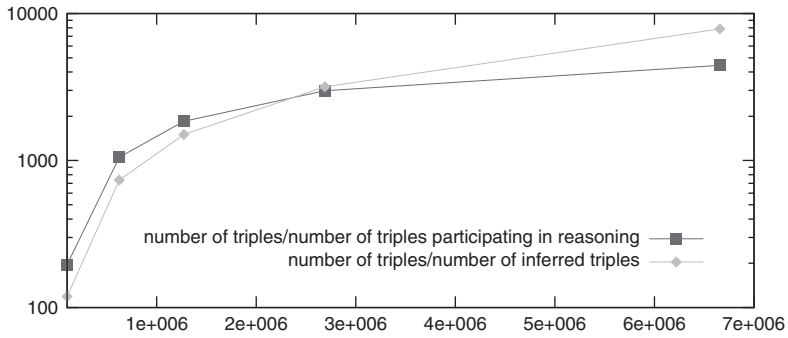

Fig. 16. The utility of the extension-based inference algorithm with increasing sizes of instance data (up to LUBM(50,0)).

parallel with the results of *Experiment 1* (Tables 17, 18, Figs. 13–15). The parsing and transformation processes are not affected by the extra TBox axioms. However, the number of TBox triples participating in reasoning is increased; thus, the reasoning process is longer in comparison to a LUBM dataset of the same size.

7.3. Experiment 3

This experiment evaluates the optimized inference engine with larger data sets (LUBM(1,0) to LUBM(50,0)). The utility of the extension-based inference algorithm is proven with a decrease in the number of triples participating in the inference and the number of inferred triples.

Fig. 16 shows the ratio of the number of total triples to the number of triples participating in reasoning and to the number of inferred triples. These ratios, and consequently, the utility of the algorithm, increase logarithmically with the size of the data set. For example, increasing the data set by 60 times results in an increase in the utility by 1000 times.

7.4. Experiment 4

In this experiment, we compare the performances of the optimized inference engine with the existing reasoners, including *Jena*,[5] *Pellet*,[6] and *OWLIM*[7] [31]. The optimized inference engine outperformed *Jena* with all datasets (Table 19). On the other hand, *Pellet* performed slightly better with smaller datasets, but when the dataset was large enough (in this case, 100,881 triples), the optimized inference engine scaled better (Table 19). The query answering performance is evaluated using 14 queries of the LUBM benchmark. Table 20 shows the results of LUBM queries. The optimized inference engine outperformed *Jena* and *Pellet* in 9 queries. Jena and Pellet outperformed the optimized inference engine in 5 queries, which is a reasonable result especially when you consider the performance penalties caused by backward chaining. Besides, the query response time is planned to be reduced with further optimizations (see Section 9).

OWLIM is the most scalable semantic database. Table 21 compares the ontology loading performances of the optimized inference engine with *OWLIM* using varying sizes of the LUBM data set. *OWLIM* outperforms the optimized inference engine, but the optimized inference engine reduces the difference with increasing sizes of instance data. In other words, the optimized inference engine is slower but scales better for larger data sets.

8. Comparison with other approaches

In this section, we briefly present some of the research literature addressing the scalability issue with respect to the development and deployment of knowledge base systems on the Semantic Web. The rest of this section lists the existing systems and compares the extension-based inference algorithm with them in terms of reasoning, querying and storage strategies.

The extension-based inference algorithm uses a partial materialization strategy. The forward chaining process makes the maximum possible inference about the instance data using extensions instead of using instance data. The remaining inference about instance data is completed via database reasoning and backward chaining. Reducing the load in the backward chaining process, results in a decrease in the complexity of the rewritten queries. For example, [32] describes a schema-closure reasoning approach, which computes schema statements (schema closure) with forward reasoning, and uses a query rewriting technique to handle inferences on instance data. However, because the extension-based inference algorithm infers all of the facts about the instance data by backward reasoning, the query rewriting process results in more complex queries compared to our approach. The following example shows the difference between two approaches with an instance retrieval query:

$$
\begin{aligned}
&schema{-}closure:\\
&type(?i,C)\Rightarrow(subClassOf(?C_s,C)\wedge type(?i,C))\vee\\
&(subClassOf(?C_s,C)\wedge subPropertyOf(?p_s,p)\wedge domain(p,C_s)\wedge ?p_s(?i,?x))\vee\\
&(subClassOf(?C_s,C)\wedge subPropertyOf(?p_s,p)\wedge range(p,C_s)\wedge ?p_s(?x,?i))\\
&extension{-}based:\\
&type(?i,C)\Rightarrow hasClassExtension(C,?\varepsilon)\wedge contains(?\varepsilon,?i).
\end{aligned}
$$

Our query answering algorithm, creates an initial query tree using the closure of the schema, and then sprouts this tree according to our node expansion algorithm. Finally, the algorithm transforms the query tree into its SQL equivalent and executes this SQL query. The use of extension-based knowledge model and database technology results in a general improvement in query performances.

SHER ([33,34]) is a scalable DL reasoner developed at IBM. *SHER*'s reasoning technique relies on a novel combination of indexing the instances of the database from the perspective of reasoning. This indexing technique summarizes the instance data into a very compact representation that is used for reasoning. *SHER* uses this representation to efficiently filter instance data that is irrelevant for answering a certain query, and selectively decompresses portions of the summarized representation relevant for the query, in a process called refinement. The combination of summarization and refinement is the key to *SHER*'s scalability. *SHER* performs membership query answering as well as conjunctive query answering using a set of optimization techniques, which leverage summarization in the context of conjunctive querying, and also incorporate faster incomplete reasoning techniques into query answering. *SHER* therefore can be used to get fast, incomplete answers to queries. This faster algorithm can help retrieve large result sets for most queries within a minute or two [35]. However, the instance retrieval queries used in [35] are simpler than the UOBM queries (*Q1–Q13* in Appendix C). *SHER* uses the approach of query answering through inconsistency checking, which is an expensive approach for an application on conjunctive queries. According to [18], this concern could be the explanation for why [35] provides no results from evaluation with the original UOBM queries.

Our extension-based knowledge model resembles *SHER* in that the reasoning is performed on the summary of ontologies, but the two approaches use very different summarization mechanisms. The extension-based summarization mechanism is based on the TBox of the ontology. In contrast, "Summary Abox" is based on instance data; thus, whenever the ABox data changes, the summarization process starts from scratch. When you consider that the ABox is not only expected to be the largest part of an ontology but is also subject to frequent changes [5], then a summarization mechanism on the Tbox of an ontology is much more reasonable

[5] We used Jena API version 2.6.4 in Java version 1.6.0. We chose OWL_MEM_MICRO_RULE_INF from the OntModelSpec, which creates an ontology object in memory and makes Jena use the micro OWL rules inference engine.
[6] We used Pellet version 2.2.2 with Jena.
[7] Version 2.9.1.

than the one on the ABox of an ontology. Another disadvantage of using *SHER* is that the queries that *SHER* can answer are limited and the query answers are incomplete.

Minerva [24] is a storage and inference system for large-scale OWL ontologies on top of relational databases. Minerva aims to meet scalability requirements of real applications and provides practical reasoning capability as well as high query performance. The method combines Description Logic reasoners for the TBox inference with logic rules for the ABox inference. Furthermore, it customizes the database schema based on inference requirements. User queries are answered by directly retrieving materialized results from the back-end database. A drawback of Minerva in comparison to the extension-based inference algorithm is that Minerva can only answer a fraction even of the OWL Lite queries completely. For example, Minerva is incomplete on *Q13* of the UOBM queries (Table 22).

OWLDB [36] is a lightweight and extensible approach for the integration of relational databases and description logic based ontologies. The inference rules are translated into queries on a relational DBMS instance, and the query results are added back to this database. The OWLDB differs from the extension-based inference algorithm in that it uses a total materialization technique, which requires considerable additional disk space. Besides, the extension-based inference algorithm prevents the inference of type relations. Therefore, the space consumption of ABox reasoning is reduced dramatically.

DLDB2 [37] is a knowledge base system that combines a relational database management system with additional capabilities for partial OWL reasoning. *DLDB2* [37] delegates TBox reasoning to a DL reasoner and pre-computes the subsumption hierarchy. The system uses table schema, database views, and algorithms that achieve essential ABox reasoning over an RDBMS. *DLDB2* has very fast load times because the inferred closure of the database is not calculated at load time, but its querying is slow [38]. An advantage of the system is that because the closure is only calculated when queries are posed on the system, updates and deletes can be performed on the system.

Our extension-based inference algorithm resembles *DLDB2* in that it uses RDBMS to achieve scalable ABox reasoning. The extension-based inference algorithm is complete on all LUBM and UOBM queries. *DLDB2* is also complete on all LUBM queries but is incomplete on 3 out of 13 UOBM queries. The incomplete UOBM queries (*Q2*, *Q9*, and *Q13*) are because *DLDB2* is not able to perform inference based on universal restrictions and cardinalities (Table 22) [37].

Jena [22] uses a hybrid reasoner, which employs both of the forward and backward rule engines together. Ontologies can be stored either in memory or in a database. The database schema is a generic RDF store, which is based on a relational statement table of three columns (Subject, Property, and Object) to store all triples. However, unlike other generic RDF stores, *Jena* uses multiple statement tables. *Jena* uses total materialization and in-memory reasoning. Total materialization requires relatively much more space; therefore, the extension-based inference algorithm outperforms *Jena* by a considerable margin in time and space performance tests using LUBM and UOBM benchmarks.

Table 23 shows the results of comparison between the extension-based inference algorithm and related approaches. The first criterion is the type of database schema. *SHER*, *OWLDB*, *Jena* and extension-based inference algorithm use generic RDF stores (G). In contrast, *Minerva* and *DLDB2* customize the database schema (C) based on inference requirements. The second criterion is summarization. *SHER* and the extension-based knowledge model differ in the way that they use novel summarization mechanisms on ontological data. The third criterion is materialization, which is the process of computing all implicit assertions in the KB and is frequently employed by semantic query and reasoning engines to improve query performance. *Minerva*, *OWLDB*, *DLDB2* and *Jena* apply total materialization (T) on ontological data. On the other side, in partial materialization (P), specific reasoning is not performed at the materialization time but rather at the query time. *SHER* and extension-based inference algorithm use partial materialization. The fourth criterion is the reasoning approach. *SHER* and *DLDB* are description logic reasoners (DL). *OWLDB*, *Jena* and the extension-based inference algorithm are rule-based reasoners (RB). *Minerva* is a hybrid reasoner that combines Description Logic reasoners for the TBox inference with logic rules for the ABox inference. The last criterion concerns where the reasoning takes place. *SHER* and *Jena* are in-memory reasoners (M). *Minerva* and *OWLDB* are in-database reasoners (DB). *DLBD2* and extension-based inference algorithm are both in-memory and in-database reasoners.

9. Conclusions and future work

This paper presents a rule-based hybrid reasoning approach for the pD^* ontology language. This approach is based on the extension-based knowledge model, which works like a summarization mechanism on ontology individuals and provides a foundation for both backward and forward chaining. The model transformation overhead becomes negligible compared to the decrease in time and space consumption of forward reasoning. Another effect on forward reasoning process is the decrease in the number of inferred triples. Besides the ratio of the decrease in the inferred triples is proportional with the size of individual data. The extension-based knowledge model reduces the complexity (number of constraints) of the queries.

Although the technique shows a significant amount of improvement, there is still room for optimizations, which can be grouped into two categories. The first group contains optimizations on the database schema. Using a customized database schema (such as Minerva) would improve the inference, storage and query performances of the database. Another interesting piece of future work would be to integrate “vertical partitioning” [39] into our approach for improved inference and query performance. According to this technique, our *statements* table would be split based on the predicate column. These newly created tables, which contain two columns (*subject*, *object*), would be indexed by their subject and optionally by their objects. We expect this partitioning to improve the query performances significantly.

The second group contains the optimizations on query rewriting. The query rewriting algorithm mainly involves collecting members of extensions. We plan to optimize the query rewriting algorithm by preventing attempts to collect the members of

empty extensions. Finally, we plan to eliminate node repetition in the query tree through the use of a node sharing approach (similar to the "node sharing" of the Rete algorithm [30]).

In addition to these optimizations, using a database system (other than Sqlite) that has a built-in query optimizer for improving query performance further will be an interesting piece of future work.

Acknowledgments

This work is based on the PhD dissertation of Övünç Öztürk (2009). The author is sincerely grateful to Professor Oğuz Dikenelli, Asst. Prof. Adil Alpkoçak, Asst. Prof. R. Cenk Erdur and Asst. Prof. Murat Komesli for their valuable contributions to the thesis studies. The inference engine is a part of a project, which is being carried out with financial support from TUBITAK (The Scientific and Technical Research Council of Turkey) under grant reference 3070489.

Appendix A. OWL constructs

Property characteristics

- *TransitiveProperty*: if a property P is specified as transitive, then $P(x,y)$ and $P(y,z)$ implies $P(x,z)$.
- *SymmetricProperty*: if a property P is tagged as symmetric, then $P(x,y)$ implies $P(y,x)$.
- *FunctionalProperty*: if a property P is tagged as functional, then $P(x,y)$ and $P(x,z)$ implies $y=z$.
- *inverseOf*: if a property P_1 is tagged as the "owl:inverseOf" P_2, then $P_1(x,y)$ implies $P_2(y,x)$.
- *InverseFunctionalProperty*: if a property P is tagged as *InverseFunctional*, then $P(y,x)$ and $P(z,x)$ implies $y=z$.

Property restrictions

- *allValuesFrom, someValuesFrom*: the "owl:allValuesFrom" restriction requires that for every instance of the class that has instances of the specified property, the values of the property are all members of the class indicated by the "owl:allValuesFrom" clause. For example, the maker of a *Wine* must be a *Winery*. The *allValuesFrom* restriction is applied on the *hasMaker* property of this *Wine* class only. Makers of *Cheese* are not constrained by this local restriction. The case of "owl:someValuesFrom" is similar. If we replaced "owl:allValuesFrom" with "owl:someValuesFrom" in the example above, it would mean that at least one of the *hasMaker* properties of a *Wine* must point to an individual that is a *Winery*.
- *Cardinality*: the cardinality constraints specify the exact, minimum, or maximum number of elements in a relation.
- *hasValue: hasValue* allows us to specify classes based on the existence of specific property values. Hence, an individual will be a member of such a class whenever at least one of its property values is equal to the *hasValue* resource.

Complex classes

- Set operators
  - *intersectionOf*: the intersection of two classes A and B is the class that contains all elements of A that also belong to B (or equivalently, all elements of B that also belong to A), but no other elements.
  - *unionOf*: the union of two classes A and B is the class that contains all distinct elements of class A and class B.
  - *complementOf*: the *complementOf* construct selects all individuals from the domain of discourse that do not belong to a certain class.
- *Enumerated Classes*: OWL provides the means to specify a class via a direct enumeration of its members. This is done using the *oneOf* construct.
- *Disjoint Classes*: the disjointness of a set of classes can be expressed using the "owl:disjointWith" constructor. This constructor guarantees that an individual that is a member of one class cannot simultaneously be an instance of a specified other class.

Appendix B. *pD** Language

The *pD** language weakens the standard iff-semantics of OWL and extends RDFS entailment. The *pD** entailment is largely defined by means of "if conditions", and extends RDFS with datatypes and a property-related fragment of OWL. The *pD** semantics is intended to represent a reasonable interpretation that is useful for drawing conclusions about instances in the presence of an ontology and that leads to simple entailment rules and a relatively low computational complexity. Reference [8] shows the completeness of a set of simple entailment rules for *pD** entailment. *pD** Entailment is decidable, NP-complete, and in P, if the target graph has no blank nodes. Just as for RDFS entailment, a partial closure that is sufficient for deciding entailment can be computed in polynomial time.

The expressivity or complexity of *pD** is between RDFS and OWL Lite. *pD** Extends RDFS in two steps [18]:

- The D^* (see Table 24) adds entailment support for literal data-types
- The p (see Table 25) adds rules, which provide partial support for the following OWL primitives: *FunctionalProperty*, *InverseFunctionalProperty*, *SymmetricProperty*, *TransitiveProperty*, *sameAs*, *inverseOf*, *equivalentClass*, *equivalentProperty*, *onProperty*,

hasValue, *someValuesFrom*, *allValuesFrom*, *differentFrom* and *disjointWith*. The last two primitives are supported through inconsistency rules. The OWL entailments related to *someValuesFrom* and *allValuesFrom* are supported only in one of the directions (i.e., there is no full support for the iff-semantics of these OWL primitives).

Appendix C. Test queries

Table 26
LUBM and UOBM queries.

LUBM queries	
Q1	(?x type GraduateStudent) (?x takesCourse http://www.Department0.University0.edu/GraduateCourse0)
Q2	(?x type GraduateStudent) (?y type University) (?z type Department) (?x memberOf ?z) (?z subOrganizationOf ?y) (?x undergraduateDegreeFrom ?y)
Q3	(?x type Publication) (?x publicationAuthor http://www.Department0.University0.edu/AssistantProfessor0)
Q4	(?x type Professor) (?x worksFor http://www.Department0.University0.edu) (?x name ?y1) (?x emailAddress ?y2) (?x telephone ?y3)
Q5	(?x type Person ?X) (?x memberOf ?X http://www.Department0.University0.edu)
Q6	(?x type Student ?X)
Q7	(?x type Student ?X) (?y type Course ?Y) (http://www.Department0.University0.edu/AssociateProfessor0 teacherOf ?Y) (?x takesCourse ?y)
Q8	(?x type Student) (?y type Department) (?x memberOf ?y) (?y subOrganizationOf http://www.University0.edu) (?x emailAddress ?z)
Q9	(?x type Student) (?y type Faculty) (?z type Course) (?x advisor ?y) (?x takesCourse ?z) (?y teacherOf?z)
Q10	(?x type Student) (?x takesCourse http://www.Department0.University0.edu/GraduateCourse0)
Q11	(?x type ResearchGroup) (?x subOrganizationOf http://www.University0.edu)
Q12	(?x type Chair) (?y type Department) (?x worksFor ?y) (?y subOrganizationOf http://www.University0.edu)
Q13	(?x type Person) (?x hasAlumnus http://www.University0.edu)
Q14	(?x type UndergraduateStudent)
UOBM queries	
Q1	(?x type UndergraduateStudent) (?x takesCourse http://www.Department0.University0.edu/Course0)
Q2	(?x type Employee)
Q3	(?x type Student) (?x isMemberOf http://www.Department0.University0.edu)
Q4	(?x type Publication) (?x publicationAuthor ?y) (?y type Faculty) (?y isMemberOf http://www.Department0.University0.edu)
Q5	(?x type ResearchGroup) (?x subOrganizationOf http://www.University0.edu)
Q6	(?x type Person) (http://www.University0.edu hasAlumnus ?x)
Q7	(?x type Person) (?x hasSameHomeTownWith http://www.Department0.University0.edu/FullProfessor0)
Q8	(?x type SportsLover) (http://www.Department0.University0.edu hasMember ?x}
Q9	(?x type GraduateCourse) (?x isTaughtBy ?y) (?y isMemberOf ?z) (?z subOrganizationOf http://www.University0.edu)
Q10	(?x isFriendOf http://www.Department0.University0.edu/FullProfessor0)
Q11	(?x type Person) (?x like ?y) (?z type Chair) (?z isHeadOf http://www.Department0.University0.edu) (?z like ?y)
Q12	(?x type Student) (?x takesCourse ?y) (?y isTaughtBy http://www.Department0.University0.edu/FullProfessor0)
Q13	(?x type PeopleWithHobby) (?x isMemberOf http://www.Department0.University0.edu)

References

[1] T. Berners-Lee, J. Hendler, O. Lassila, The Semantic Web, Scientific American, , 2001.
[2] J.Z. Pan, A flexible ontology reasoning architecture for the Semantic Web, IEEE Transactions on Knowledge and Data Engineering 19 (2) (2007) 246–260.
[3] T.R. Gruber, Toward principles for the design of ontologies used for knowledge sharing, International Journal of Human Computer Studies 43 (5–6) (1995) 907–928.
[4] K. Srinivas, OWL Reasoning in the Real World: Searching for Godot, in: B.C. Grau, I. Horrocks, B. Motik, U. Sattler (Eds.), Description Logics, Vol. 477 of CEUR Workshop Proceedings, CEUR-WS.org, 2009.
[5] T. Weithöner, T. Liebig, M. Luther, S. Böhm, F. Henke, O. Noppens, Real-World Reasoning with OWL, Proceedings of the 4th European Conference on The Semantic Web (ESWC2007), Springer-Verlag, Berlin, Heidelberg, 2007, pp. 296–310.
[6] R. B'Far, Scalable Reasoning Techniques for Semantic Enterprise Data, in: D. Wood (Ed.), Linking Enterprise Data, Springer, US, 2010, pp. 127–147.
[7] T. Özacar, Ö. Öztürk, M.O. Ünalir, A Pragmatic Approach for RDFS Reasoning over Large Scale Instance Data, OTM Workshops, 2, 2007, pp. 1155–1164.
[8] H.J. ter Horst, Completeness, decidability and complexity of entailment for RDF schema and a Semantic extension involving the OWL vocabulary, Journal of Web Semantics 3 (2–3) (2005) 79–115.
[9] J. Heflin, OWL Web Ontology Language Use Cases and Requirements, Tech. rep., W3C, , 2004.
[10] G.D. Giacomo, M. Lenzerini, TBox and ABox Reasoning in Expressive Description Logics, Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR1996, Morgan Kaufmann, 1996, pp. 316–327.
[11] N.B.G. Meditskos, Handbook of Research on Emerging Rule Based Languages and Technologies: Open Solutions and Approaches, IGI Global, Ch. Rule based OWL Reasoning Systems: Implementations, Strengths and Weaknesses, 2009, pp. 124–148.
[12] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003.
[13] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, Y. Katz, Pellet: a practical OWL-DL reasoner, Web Semantics: Science, Services and Agents on the World Wide Web 5 (2) (2007) 51–53.
[14] V. Haarslev, K. Hidde, R. Moller, M. Wessel, The RacerPro knowledge representation and reasoning system, Semantic Web Journal 2 (2011).
[15] D. Tsarkov, I. Horrocks, FaCT++ Description Logic Reasoner: System Description, Proceedings of the International Joint Conference on Automated Reasoning (IJCAR2006), Springer, 2006, pp. 292–297.
[16] G. Meditskos, N. Bassiliades, CLIPS–OWL: a framework for providing object-oriented extensional ontology queries in a production rule engine, Data & Knowledge Engineering 70 (7) (2011) 661–681.
[17] G. Meditskos, N. Bassiliades, Combining a DL Reasoner and a Rule Engine for Improving Entailment Based OWL Reasoning, Proceedings of the 7th International Conference on The Semantic Web (ISWC2008), 2008, pp. 277–292.
[18] A. Kiryakov, Measurable Targets for Scalable Reasoning, Tech. Rep, The Large Knowledge Collider: A Platform for Large Scale Integrated Reasoning and Web-Search, 2008.

[19] Z. Pan, J. Heflin, DLDB: Extending Relational Databases to Support Semantic Web Queries, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, 2003, pp. 109–113.
[20] M.d.M. Roldan-Garcia, J.F. Aldana-Montes, DBOWL: Towards a Scalable and Persistent OWL Reasoner, Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services (ICIW2008), IEEE Computer Society, Washington, DC, USA, 2008, pp. 174–179.
[21] S. Heymans, L. Ma, D. Anicic, Z. Ma, N. Steinmetz, Y. Pan, J. Mei, A. Fokoue, A. Kalyanpur, A. Kershenbaum, E. Schonberg, K. Srinivas, C. Feier, G. Hench, B. Wetzstein, U. Keller, Ontology Reasoning with Large Data Repositories, in: R. Jain, A. Sheth, M. Hepp, P. Leenheer, A. Moor, Y. Sure (Eds.), Ontology Management, Vol. 7 of Semantic Web and Beyond, Springer, US, 2008, pp. 89–128.
[22] J.J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, K. Wilkinson, Jena: Implementing the Semantic Web Recommendations, WWW Alt.'04: Proceedings of the 13th international World Wide Web Conference on Alternate Track Papers and Posters, ACM, New York, NY, USA, 2004, pp. 74–83.
[23] S. Das, E.I. Chong, Z. Wu, M. Annamalai, J. Srinivasan, A Scalable Scheme for Bulk Loading Large RDF Graphs into Oracle, Proceedings of the IEEE 24th International Conference on Data Engineering (ICDE2008), 2008, pp. 1297–1306.
[24] J. Zhou, L. Ma, Q. Liu, L. Zhang, Y. Yu, Y. Pan, Minerva: A Scalable OWL Ontology Storage and Inference System, Proceedings of the 1st Annual Asian Semantic Web Conference (ASWC2006), 2006, pp. 429–443.
[25] L. Al-Jadir, C. Parent, S. Spaccapietra, Reasoning with large ontologies stored in relational databases: the OntoMinD approach, Data & Knowledge Engineering 69 (11) (2010) 1158–1180 Special issue on contribution of ontologies in designing advanced information systems.
[26] H. Stuckenschmidt, M. Klein, Reasoning and change management in modular ontologies, Data & Knowledge Engineering 63 (2) (2007).
[27] Y. Guo, Z. Pan, J. Heflin, An Evaluation of Knowledge Base Systems for Large OWL Datasets, Proceedings of the Third International Semantic Web Conference (ISWC2004), 2004, pp. 274–288.
[28] L. Zeng, A.H.H. Ngu, B. Benatallah, R.M. Podorozhny, H. Lei, Dynamic composition and optimization of web services, Distributed and Parallel Databases 24 (1–3) (2008) 45–72.
[29] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, Towards A Complete OWL Ontology Benchmark, Proceedings of the 3rd European Semantic Web Conference (ESWC2006), 2006.
[30] C. Forgy, Rete: a fast algorithm for the many patterns many objects match problem, Artificial Intelligence 19 (1) (1982) 17–37.
[31] A. Kiryakov, D. Ognyanov, D. Manov, OWLIM — A Pragmatic Semantic Repository for OWL, Proceedings of the WISE Workshops (WISE2005), 2005, pp. 182–192.
[32] H. Stuckenschmidt, J. Broekstra, Time — Space Trade-Offs in Scaling up RDF Schema Reasoning, Proceedings of the WISE Workshops, 2005, pp. 172–181.
[33] A. Fokoue, A. Kershenbaum, L. Ma, E. Schonberg, K. Srinivas, The Summary Abox: Cutting Ontologies Down to Size, In Proceedings of the 5th International Semantic Web Conference (ISWC2006), 2006, pp. 343–356.
[34] J. Dolby, A. Fokoue, A. Kalyanpur, E. Schonberg, K. Srinivas, Scalable highly expressive reasoner (SHER), Journal of Web Semantics 7 (4) (2009) 357–361.
[35] J. Dolby, A. Fokoue, A. Kalyanpur, A. Kershenbaum, E. Schonberg, K. Srinivas, L. Ma, Scalable Semantic Retrieval through Summarization and Refinement, Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI2007), 2007, pp. 299–304.
[36] S. Auer, Z. Ives, Integrating Ontologies and Relational Data, Tech. Rep. MS-CIS-07-24, Computer and Information Sciences Department, School of Engineering and Applied Science, University of Pennsylvania, 2007.
[37] Z. Pan, X. Zhang, J. Heflin, DLDB2: A Scalable Multi-perspective Semantic Web Repository, IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, 1, 2008, pp. 489–495.
[38] P.J. McBrien, N. Rizopoulos, A.C. Smith, SQOWL: Type Inference in an RDBMS, Proceedings of the 29th International Conference on Conceptual Modeling (ER2010), Springer-Verlag, Berlin, Heidelberg, 2010, pp. 362–376.
[39] D.J. Abadi, A. Marcus, S.R. Madden, K. Hollenbach, Scalable Semantic Web Data Management using Vertical Partitioning, Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB2007), 2007, pp. 411–422.

Övünç Öztürk (PhD/MS, Ege University, Department of Computer Engineering, Turkey, 2009/2005; BS, Middle East Technical University, Department of Computer Engineering, Turkey, 2001) is currently an assistant professor at Celal Bayar University, Department of Computer Engineering. His research interests include scalable reasoning, database systems and logic programming.

Tuğba Özacar (PhD/MS, Ege University, Department of Computer Engineering, Turkey, 2009/2005; BS, Dokuz Eylül University, Department of Computer Engineering, Turkey, 2001) is currently an assistant professor at Celal Bayar University, Department of Computer Engineering. Her research interests include ontological engineering, and expert systems.

Murat Osman Ünalır (PhD/MS/BS, Ege University, Department of Computer Engineering, Turkey, 1993/1995/2001) is currently an assistant professor at Ege University, Department of Computer Engineering. His research interests include semantic web, and database systems.