Incrementally maintaining materializations of ontologies stored in logic databases

Raphael Volz(1,2) and Steffen Staab (1) and Boris Motik (2)

[1] Institute AIFB, University of Karlsruhe, Germany, mail: lastname@aifb.uni-karlsruhe.de
[2] FZI, University of Karlsruhe, Germany, volz@fzi.de

Abstract. This article presents a technique to incrementally maintain materializations of ontological entailments. Materialization consists in precomputing and storing a set of implicit entailments, such that frequent and/or crucial queries to the ontology can be solved more efficiently. The central problem that arises with materialization is its maintenance when axioms change, viz. the process of propagating changes in explicit axioms to the stored implicit entailments.
When considering rule-enabled ontology languages that are operationalized in logic databases, we can distinguish two types of changes. Changes to the ontology will typically manifest themselves in changes to the rules of the logic program, whereas changes to facts will typically lead to changes in the extensions of logical predicates. The incremental maintenance of the latter type of changes has been studied extensively in the deductive database context and we apply the technique proposed in [31] for our purpose. The former type of changes has, however, not been tackled before.
In this article we elaborate on our previous papers [34,35], which extend the approach of [31] to deal with changes in the logic program. Our approach is not limited to a particular ontology language but can be generally applied to arbitrary ontology languages that can be translated to Datalog programs, i.e. such as O-Telos, F-Logic [17] RDF(S), or Description Logic Programs [36].

1 Introduction

Germane to the idea of the Semantic Web are the capabilities to assert facts and to derive new facts from the asserted facts using the semantics specified by an ontology. Both current building blocks of the Semantic Web, RDF [14] and OWL [22], define how to assert facts and specify how new facts should be derived from stated facts.

The necessary derivation of entailed information from asserted information is usually achieved at the time clients issue queries to inference engines such as logic databases. Situations where queries are frequent or the procedure to derive entailed information is time consuming and complex typically lead to low performance. Materialization can be used to increase the performance at

query time by making entailed information explicit upfront. Thereby, the recomputation of entailed information for every single query is avoided.

Materialization has been applied successfully in many applications where reading access to data is predominant. For example, data warehouses usually apply materialization techniques to make *online* analytical processing possible. Similarly, most Web portals maintain cached web pages to offer fast access to dynamically generated web pages.

We conjecture that reading access to ontologies is predominant in the Semantic Web and other ontology-based applications, hence materialization seems to be a promising technique for fast processing of queries on ontologies.

Materialization is particularly promising for the currently predominant approach of aggregating distributed information into a central knowledge base (cf. [8,15,32,21]). For example, the OntoWeb[1] Semantic portal [29] employs a *syndicator* (cf. Figure 1), which regularly visits sites specified by community members and transfers the detected updates into a central knowledge base in a batch process. Hence, the knowledge base remains unchanged between updates for longer periods of time.

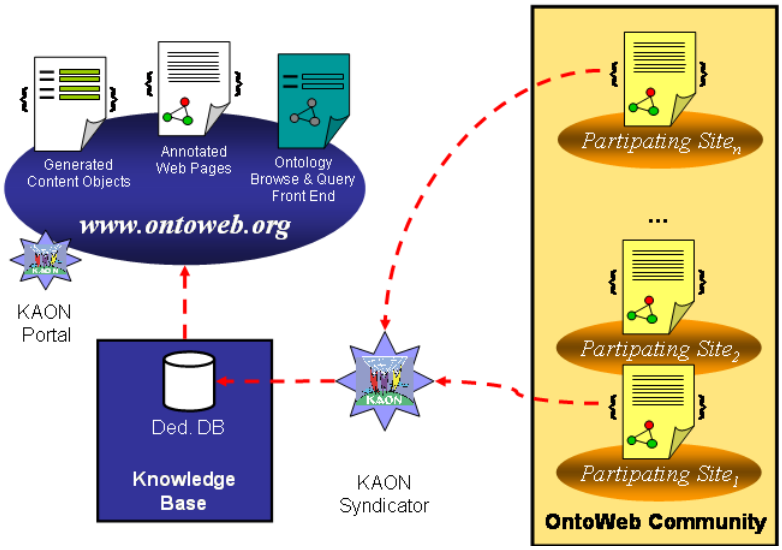

Fig. 1. OntoWeb Architecture

[1] `http://www.ontoweb.org/`.

The OntoWeb portal, however, provides answers to queries issued on the knowledge base whenever visitors browse the portal content. This is due to the fact that most queries are hard-coded into the definition of dynamic Web pages, which are generated for every request. In applications such as OntoWeb, materialization turns out to be a sine qua non.[2]

Central to materialization approaches is the issue of maintaining a materialization when changes occur. This issue can be handled by simply recomputing the whole materialization. However, as the computation of the materialization is often complex and time consuming, it is desirable to apply more efficient techniques in practice, i.e. to *incrementally* maintain a materialization.

1.1 Contribution

We present a technique for the incremental maintenance of materialized ontologies. Our technique can be applied to a wide range of ontology languages, namely those that can be axiomatized by a set of rules[3].

The challenge that has not been tackled before comes from the fact that updates of ontology definitions are equivalent to the updates and new definitions of rules, whereas existing maintenance techniques only address the update of ground facts.

To cope with changing rules, our solution extends a declarative algorithm for the incremental maintenance of views [31] that was developed in the deductive database context. We show the feasibility of our solution in a performance evaluation.

1.2 Organization

The remainder of the article is organized as follows: Section 2 reviews how current Web ontology languages such as RDF(S) and OWL interplay with rules. Section 3 presents the underlying principles which are applied to achieve incremental maintenance of a materialization. Section 4 recapitulates the incremental maintenance algorithm presented in [31] , presents a novel modular rewriting algorithm based on generator functions and shows how this algorithm deals with changes to facts. Section 5 extends this algorithm to deal with changing rules as they result from changes in the ontology. Section 6 sketches how the developed techniques can be applied in implementations of RDF rule languages. Section 7 describes our prototypical implementation. Section 8 performs a performance analysis and shows the benefits of our approach. Section 10 summarizes our contribution and discusses further uses.

[2] Even though in OntoWeb, due to the unavailability of the solution developed in this article, the problem was approached by caching the Web pages through a proxy server.

[3] The underlying rule language used for our approach is Datalog with stratified negation.

2 Web ontology languages and logic databases

In the brief history of the Semantic Web, most applications, e.g. [7], have implemented the logical entailment supported by ontology languages either directly using Logic Programming techniques, e.g. [4,26], or by relying on (available) logic databases[4] [23,28]. Furthermore, a large expressive fragment of the recently standardized Web Ontology Language (OWL) can be implemented in logic databases [36].

2.1 Axiomatization of ontology languages

Systems like SilRi [7], CWM[5], Euler [26], JTP[6] or Triple [28] and ConceptBase [16] implement the semantics of a particular ontology language via a static axiomatization, i.e. a set of rules. For example, Figure 2 presents the Datalog axiomatization of the RDF vocabulary description language (RDFS) [5]. This axiomatization implements the semantics of RDF specified by the RDF model theory [14] (without datatype entailments and support for stronger iff semantics of domain and ranges). The ontology and associated data is stored in a single ternary predicate t, i.e. the extension of t stores all triples that constitute a particular RDF graph.

t(P,a,rdf:Property)	:- t(S,P,O).	*rdf1*
t(S,a,C)	:- t(P,domain,C), t(S,P,O).	*rdfs2*
t(O,a,C)	:- t(P,range,C), t(S,P,O).	*rdfs3*
t(S,a,Resource)	:- t(S,P,O).	*rdfs4a*
t(O,a,Resource)	:- t(S,P,O).	*rdfs4b*
t(P,subPropertyOf,R)	:- t(Q,subPropertyOf,R), t(P,subPropertyOf,Q).	*rdfs5a*
t(S,R,0)	:- t(P,subPropertyOf,R), t(S,P,O).	*rdfs6*
t(C,a,Class)	:- t(C,subClassOf,Resource).	*rdfs7*
t(A,subClassOf,C)	:- t(B,subClassOf,C), t(A,subClassOf,B).	*rdfs8*
t(S,a;B)	:- t(S,a,A), t(A,subClassOf,B).	*rdfs9*
t(X,subPropertyOf,member)	:- t(X,a,ContainerMembershipProperty).	*rdfs10*
t(X,subClassOf,Literal)	:- t(X,a,Datatype).	*rdfs11*
t(Resource,subClassOf,Y)	:- t(X,domain,Y), t(rdf:type,subPropertyOf,X).	*rdfs12*

Fig. 2. Static Datalog rules for implementing RDF(S)

[4] We use the term logic database over the older term deductive databases since the later is very closely associated with Datalog, a particular Logic Programming language that is frequently used in logic databases. Modern logic databases such as XSB [27] and CORAL [25] support more expressive Logic Programming languages that include function symbols and nested expressions. Furthermore, several lectures, e.g. `http://user.it.uu.se/~voronkorov/ddb.htm` nowadays use this term.

[5] `http://www.w3.org/2000/10/swap/doc/cwm`

[6] `http://ksl.stanford.edu/software/jtp/`

2.2 Dynamic rule sets

The set of rules is typically not immutable. With the advent of higher layers of the Semantic Web stack, i.e. the rule layer, users can create their own rules. Hence, we are facing a scenario where not only base facts can change but also the set of rules. This requires the ability to maintain a materialization in this situation.

Besides support for a rule layer, the ability to maintain a materialization under changing rule sets is also required for approaches where the semantics of the ontology language is not captured via a static set of rules but instead compiled into a set of rules. Such an approach is for example required by Description Logic Programs (DLP) [36], where OWL ontologies are translated to logic programs.

2.2.1 Semantic Web Rule layer We now briefly present some languages for the specification of Semantic Web rules that may be compiled into the paradigm we use. The Rule Markup Initiative[7] aims to develop a canonical Web language for rules called RuleML. RuleML covers the entire rule spectrum and spans from derivation rules to transformation rules to reaction rules. It has a well-defined Datalog subset, which can be enforced using XML schemas, and for which we can employ the materialization techniques developed within this paper. The reader may note, that materialization is not an issue for many other aspects found in RuleML, e.g. transformation rules or reaction rules.

In parallel to the RuleML iniative, Notation3 (N3)[8] has emerged as a human-readable language for RDF/XML. Its aim is to optimize expression of data and logic in the same language and has become a serious alternative since many systems that support inference on RDF data, e.g. cwm[9], Euler (cf. `http://www.agfa.com/w3c/euler/`), Jena2 (cf. `http://www.hpl.hp.com/semweb/jena.htm`), support it. The rule language supported by N3 is an extension of Datalog with existential quantifiers in rule heads. Hence, the materialization techniques developed within this paper can be applied to the subset of all N3 programs which do not make use of existential quantification in the head.

2.2.2 Description Logic Programs (DLP) Both of the above mentioned approaches allow the definition of rules but are not integrated with the ontology layer in the Semantic Web architecture. Description Logic Programs [36] aim to integrate rules with the ontology layer by compiling ontology definitions into a logic program which can later be extended with additional rules. This approach can deal with an expressive subset of the standardized Web ontology language OWL (i.e. OWL without existential quantification, negation and disjunction in rule heads).

OWL classes are represented in the logic database as unary predicates and OWL properties is represented as binary predicates. Classes may be constructed

[7] cf. `http://www.ruleml.org/`
[8] cf. `http://www.w3.org/DesignIssues/Notation3.html`
[9] cf. `http://www.w3.org/2000/10/swap/doc/cwm`

OWL Abstract Syntax	Logic Database
`Class` (A partial $D_1 \dots D_n$)	$\bigcup_{i\in[1,n]} \varphi_{\mathcal{LP}}(A \sqsubseteq D_i)$
`Class` (A complete $D_1 \dots D_n$)	$\bigcup_{i\in[1,n]} \varphi_{\mathcal{LP}}(A \equiv D_i)$
`EquivalentClasses` ($D_1 \dots D_n$)	$\bigcup_{i\in[1,n]} \varphi_{\mathcal{LP}}(D_1 \equiv D_i)$
`SubClassOf` ($D_1 D_2$)	$\varphi_{\mathcal{LP}}(D_1 \sqsubseteq D_2)$
$\varphi_{\mathcal{LP}}(C \equiv D)$	$\begin{cases} \varphi_{\mathcal{LP}}(C \sqsubseteq D) \\ \varphi_{\mathcal{LP}}(D \sqsubseteq C) \end{cases}$
$\varphi_{\mathcal{LP}}(C \sqsubseteq D)$	$\varphi^R_{\mathcal{LP}}(D,x)\text{:-}\varphi^L_{\mathcal{LP}}(C,x).$
$\varphi^R_{\mathcal{LP}}(A,x)\text{:-}B.$	$A(x)\text{:-}B.$
$\varphi^R_{\mathcal{LP}}(\exists R.\{i\},x)\text{:-}B.$	$R(x,i)\text{:-}B.$
$\varphi^R_{\mathcal{LP}}(C \sqcap D,x)\text{:-}B.$	$\begin{cases} \varphi^R_{\mathcal{LP}}(C,x)\text{:-}B. \\ \varphi^R_{\mathcal{LP}}(D,x)\text{:-}B. \end{cases}$
$\varphi^R_{\mathcal{LP}}(\forall R.C,x)\text{:-}B.$	$\varphi^R_{\mathcal{LP}}(C,y_i)\text{:-}R(x,y_i),B.$
$H\text{:-}\varphi^L_{\mathcal{LP}}(\exists R.\{i\},x),B.$	$H\text{:-}R(x,i),B.$
$H\text{:-}\varphi^L_{\mathcal{LP}}(A,x),B.$	$H\text{:-}A(x),B.$
$H\text{:-}\varphi^L_{\mathcal{LP}}((\exists R.C),x),B.$	$H\text{:-}R(x,y_i),C(y_i),B.$
$H\text{:-}\varphi^L_{\mathcal{LP}}((C \sqcap D),x),B.$	$H\text{:-}\varphi^L_{\mathcal{LP}}(C,x),\varphi^L_{\mathcal{LP}}(C,x),B.$
$H\text{:-}\varphi^L_{\mathcal{LP}}((C \sqcup D),x),B.$	$\begin{cases} H\text{:-}\varphi^L_{\mathcal{LP}}(C,x),B. \\ H\text{:-}\varphi^L_{\mathcal{LP}}(D,x),B. \end{cases}$

Table 1. DLP representation of OWL classes in logic databases

using various constructors (cf. Table 1). The OWL T-Box may consist of class inclusion axioms and class equivalence axioms, which are mapped to logical implications, i.e. rules[10]. Similarly, the T-Box may also consist of inclusion and equivalence axioms between properties. Properties may have inverses and may be defined to be symmetric and transitive (cf. Table 2).

Example 1. The following example OWL fragment declares Wine to be potable liquids who are made by Wineries:

$$\text{Wine} \sqsubseteq \text{PotableLiquid} \sqcap \forall \text{hasMaker.Winery}$$

This will be translated to the following set of rules:

$$\begin{aligned} \text{PotableLiquid}(X) &\text{ :- } \text{Wine}(X). \\ \text{Winery}(Y) &\text{ :- } \text{Wine}(X), \text{hasMaker}(X,Y). \end{aligned}$$

We can easily see that a change to the class and property structure of an ontology will result in a change of the compiled rules. Again, it is necessary to be able to maintain a materialization in case of such a change.

[10] Equivalence is decomposed into two inverse inclusion axioms

OWL Abstract Syntax	Logic Database
`ObjectProperty` (P	
super(Q_1) ... super(Q_n)	$\bigcup_{i\in[1,n]}\{Q_i(x,y)\text{:-}P(x,y).\}$
domain(C_1)	$C_1(x)\text{:-}P(x,y).$
...	...
domain(C_n)	$C_n(x)\text{:-}P(x,y).$
range(R_1)	$R_1(y)\text{:-}P(x,y).$
...	...
range(R_n)	$R_n(y)\text{:-}P(x,y).$
inverseOf(Q)	$P(x,y)\text{:-}Q(y,x).$
	$Q(x,y)\text{:-}P(y,x).$
Symmetric	$P(x,y)\text{:-}P(y,x).$
Transitive	$P(x,z)\text{:-}P(x,y),P(y,z).$
)	
`EquivalentProperties` ($P_1 \dots P_n$)	$\bigcup_{i\in[1,n]}\{P_1(x,y)\text{:-}P_i(x,y).P_i(x,y)\text{:-}P_1(x,y).\}$
`SubPropertyOf` (P Q)	$Q(x,y)\text{:-}P(x,y).$

Table 2. DLP Representation of OWL properties in logic databases

An OWL A-Box, i.e. individuals and property fillers, are represented as facts in the logic database, where individuals i instantiating a class C and fillers (a, b) of a property P are simple facts of the form $C(i)$ and $P(a, b)$.

2.3 Differentiating Between Asserted and Entailed Information

The fundamental requirement for our approach to maintenance is the ability to distinguish entailed information from asserted information. This ability is required in order to propagate changes. The requirement also commonly arises in many ontology-based applications [2], which often need to differentiate between asserted information and information that has been derived by making use of TBox axioms, e.g. one could prevent users from updating entailed information [3].

To achieve this differentiation, all TBox axioms are translated into rules between purely intensional predicates C_{idb}, P_{idb}. ABox assertions, however, are stored in dedicated extensional predicates C_{edb}, P_{edb}. The connection between the intensional and the extensional database is made using simple rules that derive the initial (asserted) extension of the intensional predicates:

$$C_{idb}(x)\text{:-}C_{edb}(x).$$
$$P_{idb}(x,y)\text{:-}P_{edb}(x,y).$$

3 Maintenance Principles

This section discusses how the two main kinds of updates that have been mentioned in the introduction of the chapter, viz. updates to facts and rules, affect

the materialization of an example knowledge base. Based on this discussion, we identify the main assumptions that underly the approaches for incremental maintenance presented in the subsequent sections.

As an example, we use the genealogical relationships between the different (more prominent) members of the Bach family. The relevant subset of the ABox is presented in Figure 3.

3.1 Updates to Facts

Since we expect that the historic data about the Bach family members in our knowledge base is unlikely to change, we choose to materialize the closure of the transitive ancestorOf property to speed up query processing. Figure 3 depicts an excerpt of the family tree of the Bach family, where the left-hand side of Figure 3 depicts the asserted property fillers. The right-hand side of the Figure depicts the transitive closure of the ancestorOf graph. We will now consider how updates, which we consider as deletions and insertions only, will affect the materialization (cf.lower part of Figure 3).

3.1.1 Deletions Let us assume that we have to revoke an asserted property filler, since a historian finds out that *Johann Sebastian* was not the father of *Wilhelm Friedemann*. Clearly, this has consequences to our materialization. For example, *Johann Ambrosius* is no longer an ancestor of *Wilhelm Friedemann*. However, *Johannes* is still an ancestor of *Wilhelm Friedemann*, since not only *Johann Sebastian* but also his cousin and first wife *Maria Barbara* are descendants of *Johannes*.

If we maintain the materialization of the graph depicted in Figure 3 ourselves, a natural and straightforward approach proceeds in two steps. In order to delete links that do not hold any longer, we first mark all links that could have potentially been derived using the link leading from *Wilhelm Friedemann* to the nodes in the graph that possibly interact with the deleted link, viz. are also connected with *Johann Sebastian*. As the second step, we check whether the deletion mark is correct by reconsidering whether the respective link could be derived on some other way by combining the links supported by the updated source graph. If a mark is determined to be correct, we can delete the appropriate link in our source graph.

This two staged principle for deletion is common to most approaches for the incremental maintenance of materializations [12,31,13] and is applied by the approach presented in Section 4.

3.1.2 Insertions Now assume that we assert that *Johann Sebastian* is the ancestor of another *Johann Christian*. Clearly, we can manually derive in the example graph that *Johann Christian* must be linked with the nodes that can possibly interact with the new link, viz. are also connected with *Johann Sebastian* in the updated source graph. All new links discovered in this way have to be added to the materialization.

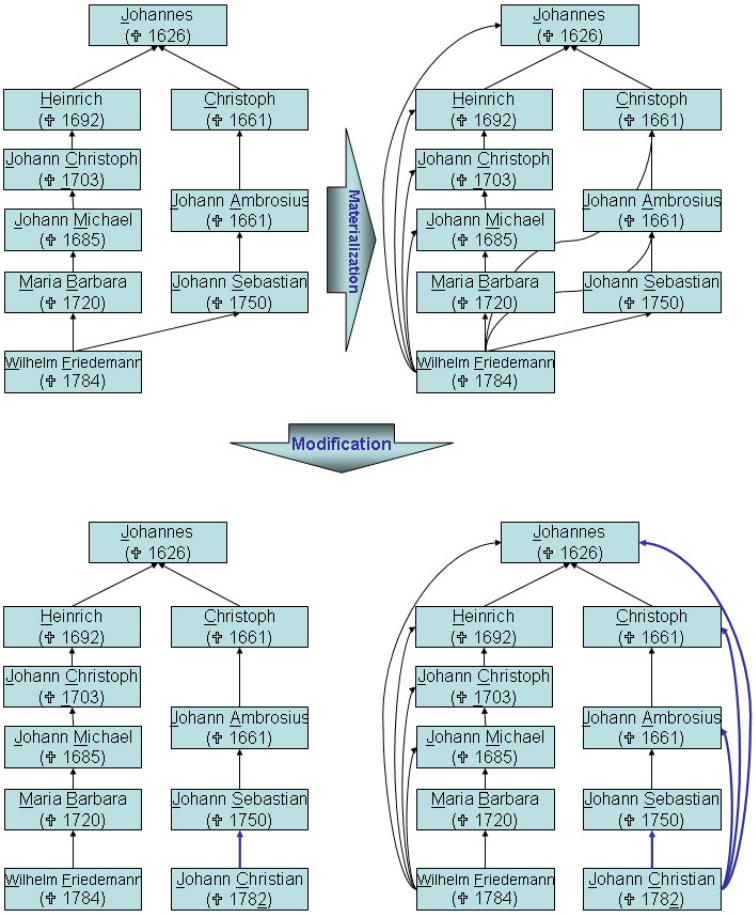

Fig. 3. Bach Family Tree Excerpt

3.2 Updates to Rules

A typical source of updates in Web ontologies is the change of TBox axioms, since ontologies have to evolve with their applications and react to changing application requirements [20,21]. Similarly, the advent of a rule layer in the Semantic Web will lead to changing rules. In the case of DLP, both situations, changing TBox axioms and changing rules, are actually equivalent since they manifest both themselves as changes to the logic program $\mathcal{LP}$.

Let's assume that our TBox states that the transitive ancestorOf property is a specialization of the inDynasty property (which is not necessarily transitive), i.e. $\mathcal{T} = \{T_0, T_1\}$ (cf. Table 3). Let's additionally assume that our ABox only contains property fillers for ANCESTOROF, e.g. the tuples $\{(h, jc1), (j, h), (j, c) \ldots\}$, where each constant is the abbreviation for the name of an individual in the

Axiom	*OWL*	*DLP*
T_0	`SubPropertyOf` (ancestorOf inDynasty)	inDynasty(x, y) :- ancestorOf(x, y)
T_1	`ObjectProperty` (ancestorOf `Transitive`)	ancestorOf(x, y) :- ancestorOf(x, z), ancestorOf(z, y).

Table 3. Example TBox $\mathcal{T}$

Bach family tree (cf. Figure 3). Clearly, the extension of both ancestorOf and inDynasty are equivalent in this particular knowledge base, since inDynasty has no own property fillers.

Manipulating the TBox $\mathcal{T}$ by deleting axiom T_0 leads to the situation that the extension of inDynasty is empty, since the derivations supported by the respective axiom are no longer supported.

Now assume that we add a new axiom T_2 to the old $\mathcal{T}$, i.e. $\mathcal{T} = \mathcal{T} \cup \{T_2\}$, that states that the property inDynasty is symmetric:

Axiom	*OWL*	*DLP*
T_2	`ObjectProperty` (inDynasty `Symmetric`)	inDynasty(x, y):-inDynasty(y, x).

Apparently, the new extension of inDynasty will now contain the tuple $(jc1, c)$ (among others), which is derived by the combination of the existing axioms and the new axiom.

Unlike the change of facts, we do not only have an interaction of particular (inserted or deleted) facts with existing facts, but also the interaction of (inserted or deleted) rules with all other rules. In particular, we can observe that we need to consider all rules defining a predicate to determine the extension of the predicate.

The approach to maintenance presented in Section 5 will therefore recompute the extensions of all predicates, which are redefined, i.e. are the head of changed rules. We will, however, reuse the mechanisms of propagating the resulting fact changes to other predicates (and possibly back to the predicate in case of cycles) from the maintenance procedure for facts (cf. next section).

4 Maintaining Changing Facts

This section presents the maintenance of a materialization when facts change, viz. new tuples are added or removed from the extension of a predicate.

4.1 Approach

We reuse the declarative variant [31] of the delete and re-derive (DRed) algorithm proposed in [12]. DRed takes the three steps illustrated in Section 3.1 to incrementally maintain the materialization of an intensional database predicate:

1. *Overestimation of deletion:* Overestimates deletions by computing all direct consequences of a deletion.
2. *Rederivation:* Prunes those estimated deletions for which alternative derivations (via some other facts in the program) exist.
3. *Insertion:* Adds the new derivations that are consequences of insertions to extensional predicates.

The declarative version[11] of DRed maintains the materialization of a given predicate by means of a maintenance program. The maintenance program is rewritten from the original program using several rewriting patterns.

The goal of the rewriting is the provision of a pair of maintenance predicates P^+ and P^- for every materialized predicate P, such that the extensions of P^+ and P^- contain the changes that are needed to maintain P after the maintenance program is evaluated on a given set of extensional insertions P^{Ins} and deletions P^{Del}.

The maintenance process is carried out as follows: First, we *setup* maintenance, i.e. the maintenance program is created for a given source program and the initial materialization of intensional predicates is computed.

Whenever extensional changes occur, the actual maintenance is carried out. In this step, we first put insertions (deletions) to an extensional predicate P_{edb} into the extension of the predicate P^{Ins}_{edb} (P^{Del}_{edb}). We then evaluate the maintenance program. For every intensional predicate P_{idb}, the required incremental changes, viz. insertions and deletions, can be found in the extension of P^+_{idb} and P^-_{idb}. We use these changes to update the materialization of the intensional predicate P and update P_{edb} with the explicit changes P^{Ins}_{edb} and P^{Del}_{edb}, while the extensions of the later predicates are deleted.

4.2 Maintenance Rewritings

The maintenance of an intensional predicate P is achieved via seven maintenance predicates :

1. P itself contains the (old) materialization.
2. P^{Del} receives so-called deletion candidates, which are the aforementioned overestimation of facts that ought to be deleted from the materialization. For extensional predicates, P^{Del} contains explicitly what should be removed from the materialization.
3. P^{Ins} contains the facts that ought to be inserted into the materialization. For extensional predicates, P^{Ins} contains the explicit insertions that were asserted by the user.

[11] The benefit of reusing the declarative version of DRed with respect to the original (procedural) version is that it allows us to reuse generic logic databases for the evaluation of the maintenance program. This also motivates why we did not use the optimized version provided in [31], since the optimization requires logic databases to evaluate the maintenance program using the supplementary magic set technique, which is not used in all logic databases (e.g. XSB [27]).

4. P^{Red} receives those facts that are marked for deletion but have alternative derivations.
5. P^{New} describes the new state of the materialization after updates.
6. P^{+} receives the net insertions required to maintain the materialization of P.
7. P^{-} receives the net deletions required to maintain the materialization of P.

New Materialization For every intensional predicate P, P^{New} captures the new materialization, which is constituted of all old data that has not been deleted (N_1). Additionally, it contains re-derived data (N_2) and inserted data (N_3):

(N_1) $\quad P^{New} \text{:-} P, \mathbf{not} P^{Del}.$
(N_2) $\quad P^{New} \text{:-} P^{Red}.$
(N_3) $\quad P^{New} \text{:-} P^{Ins}.$

For every extensional database predicate P, we only instantiate the rules (N_1 and N_3) to define an auxiliary predicate P^{New}. P^{New} is used in the rewritings for insertions and re-derivation of dependent intensional predicates.

Differentials The following differentials P^{+} and P^{-} compute positive and negative deltas, i.e. the changes that are necessary to incrementally maintain the stored materialization of an intensional predicate P:

$P^{+} \text{:-} P^{Ins}, \mathbf{not} P.$
$P^{-} \text{:-} P^{Del}, \mathbf{not} P^{Ins}, \mathbf{not} P^{Red}.$

Deletion Candidates The deletion candidates P^{Del} are constituted by all possible combinations between deleted facts of a given body predicate and the remaining body predicates. Therefore, n deletion rules are created for every rule with n conjuncts in the body:

(D_i): $\quad P^{Del} \text{:-} R_1, \ldots, R_{i-1}, R_i^{Del}, R_{i+1}, \ldots, R_n.$

If R_i is an extensional predicate, R_i^{Del} contains those facts that are explicitly deleted from R_i. Otherwise, R_i^{Del} contains the aforementioned overestimation.

Re-derivations The re-derivations P^{Red} are computed by joining the new states of all body predicates with the deletion candidates:

(R): $\quad P^{Red} \text{:-} P^{Del}, R_1^{New}, \ldots, R_n^{New}.$

Insertions Insertions P^{Ins} for intensional predicates P are calculated by ordinary semi-naive rewriting, i.e. by constructing rules (I_i) that join the insertions into a body predicate with the new materializations of all other body predicates:

(I_i): $P^{Ins} \text{ :- } R_1^{New}, \ldots, R_{i-1}^{New}, R_i^{Ins}, R_{i+1}^{New}, \ldots, R_n^{New}.$

If R_i is an extensional predicate, R_i^{Ins} contains those facts that are explicitly inserted into R_i.

4.3 Maintenance Programs

A logic program $\mathcal{LP}$ consists of a finite set of rules of the form $H \text{ :- } B_1, \ldots, B_n.$, where $H, B_i \in \mathbf{P}$. We call $\mathbf{P}$ the set of predicates used in $\mathcal{LP}$. Without loss of generality we assume that $\mathbf{P}$ can be partioned into two disjoint sets of intensional and extensional predicates, i.e. $\mathbf{P} = \mathbf{P}_{idb} \cup \mathbf{P}_{edb}$ and $\mathbf{P}_{idb} \cap \mathbf{P}_{edb} = \emptyset$. A maintenance program is generated from a program $\mathcal{LP}$ through the application of generator functions (cf. Table 4).

Generator	Parameter	Rewriting Result
	Predicate	
θ_{idb}	$P \in \mathbf{P}_{idb}$	$\theta^{New}_{idb}(P) \cup \theta^{Ins}_{idb}(P) \cup \theta^{Del}_{idb}(P) \cup \theta^{Red}_{idb}(P)$
θ^{New}_{idb}	$P \in \mathbf{P}_{idb}$	$\{\theta^{New}_1(P)\} \cup \{\theta^{New}_2(P)\} \cup \{\theta^{New}_3(P)\}$
θ^{New}_{edb}	$P \in \mathbf{P}_{edb}$	$\{\theta^{New}_1(P)\} \cup \{\theta^{New}_3(P)\}$
θ^{New}_1	$P \in \mathbf{P}$	$P^{New} \text{ :- } P, \mathbf{not}\, P^{Del}.$
θ^{New}_2	$P \in \mathbf{P}_{idb}$	$P^{New} \text{ :- } P^{Red}.$
θ^{New}_3	$P \in \mathbf{P}$	$P^{New} \text{ :- } P^{Ins}.$
θ^{+}_{idb}	$P \in \mathbf{P}_{idb}$	$P^{+} \text{ :- } P^{Ins}, \mathbf{not} P.$
θ^{-}_{idb}	$P \in \mathbf{P}_{idb}$	$P^{-} \text{ :- } P^{Del}, \mathbf{not} P^{Ins}, \mathbf{not} P^{Red}.$
θ^{Ins}_{idb}	$P \in \mathbf{P}_{idb}$	$\{\cup \theta^{Ins}(r) \mid \forall r \in rules(P)\}$
θ^{Del}_{idb}	$P \in \mathbf{P}_{idb}$	$\{\cup \theta^{Del}(r) \mid \forall r \in rules(P)\}$
θ^{Red}_{idb}	$P \in \mathbf{P}_{idb}$	$\{\theta^{Red}(r) \mid \forall r \in rules(P)\}$
	Rule	
θ	$H \text{ :- } B_1, \ldots, B_n.$	$\{\theta^{Red}\} \cup \theta^{Del} \cup \theta^{Ins}$
θ^{Red}	$H \text{ :- } B_1, \ldots, B_n.$	$H^{Red} \text{ :- } H^{Del}, B^{New}_1, \ldots, B^{New}_n.$
θ^{Del}	$H \text{ :- } B_1, \ldots, B_n.$	$\{H^{Del} \text{ :- } B_1, \ldots, B_{i-1}, B^{Del}_i, B_{i+1}, \ldots, B_n.\}$
θ^{Ins}	$H \text{ :- } B_1, \ldots, B_n.$	$\{H^{Ins} \text{ :- } B^{New}_1, \ldots, B^{New}_{i-1}, B^{Ins}_i, B^{New}_{i+1}, \ldots, B^{New}_n.\}$

Table 4. Rewriting Functions (derived from [31])

Definition 1 (Maintenance Program). *A maintenance program $\mathcal{LP}_M$ of a logic program $\mathcal{LP}$ is a set of maintenance rules such that:*

1. $\forall P \in \mathbf{P}_{idb} : \theta_{idb}(P) \in \mathcal{LP}_M$
2. $\forall P \in \mathbf{P}_{edb} : \theta^{New}_{edb}(P) \in \mathcal{LP}_M$

The θ_{idb} and θ^{New}_{edb} rewriting functions themselves call other rewriting functions presented in Table 4. For example, the function $\theta : R \rightarrow \mathbf{MR}$ rewrites a rule $R \in \mathcal{LP}$ into a set of maintenance rules $\mathbf{MR}$ by instantiating rewriting patterns for deletion θ^{Del}, insertion θ^{Ins} and rederivation θ^{Red}. By definition, θ maps every rule with n body literals into $2 * n + 1$ maintenance rules.

The reader may note that the rewriting makes use of two auxiliary functions:

- $head : \mathcal{LP} \rightarrow \mathbf{P}_{idb}$ maps a rule to its rule head.

– $rules : \mathbf{P}_{idb} \rightarrow \mathbf{R}$ maps rule heads to a set of rules $\mathbf{R}$, such that:

$$\forall P \in \mathbf{P}_{idb} : rules(P) = \{R \in \mathbf{R} | head(R) = P\}$$

Example 2 (Maintenance Rewritings). Let us return to the Bach family tree example established in Section 3.1 and consider all edges between the different individuals depicted in Figure 3 as fillers of the transitive property ANCESTOROF.

Let us consider the following logic program $\mathcal{LP}$, which generated from a simple ontology containing one single (transitive) property called ANCESTOROF.The second rule implements the differentiation between asserted and entailed information, that was described in Section 2.3:

$$(R_1)\ \text{ANCESTOROF}(x,z) \text{:-} \text{ANCESTOROF}(x,y), \text{ANCESTOROF}(y,z).$$
$$(R_2)\ \text{ANCESTOROF}(x,y) \text{:-} \text{ANCESTOROF}_{edb}(x,y).$$

In the following we will use the abbreviation A for ANCESTOROF.

Since $\mathcal{LP}$ includes one intensional predicate A and one extensional predicate A_{edb}, the generation of the maintenance program $\mathcal{LP}_M$ only involves to apply θ_{idb} to A and θ_{edb}^{New} to A_{edb}:

$$\begin{array}{lll}
\theta_{edb}^{New}(A_{edb}) = & \{A_{edb}^{New}(x,y) \text{:-} A_{edb}(x,y), \mathbf{not} A_{edb}^{Del}(x,y). & (\theta_1^{New}(A_{edb})) \\
& A_{edb}^{New}(x,y) \text{:-} A_{edb}^{Ins}(x,y).\} & (\theta_3^{New}(A_{edb})) \\
\theta_{idb}(A) = & \{A^{Del}(x,y) \text{:-} A_{edb}^{Del}(x,y). & (\theta^{Del}(R_2)) \\
& A^{Red}(x,y) \text{:-} A^{Del}(x,y), A_{edb}^{New}(x,y). & (\theta^{Red}(R_2)) \\
& A^{Ins}(x,y) \text{:-} A_{edb}^{Ins}(x,y). & (\theta^{Ins}(R_2)) \\
& A^{New}(x,y) \text{:-} A(x,y), \mathbf{not} A^{Del}(x,y). & (\theta_1^{New}(A)) \\
& A^{New}(x,y) \text{:-} A^{Red}(x,y). & (\theta_2^{New}(A)) \\
& A^{New}(x,y) \text{:-} A^{Ins}(x,y). & (\theta_3^{New}(A)) \\
& A^{Del}(x,z) \text{:-} A^{Del}(x,y), A(y,z). & (\theta^{Del}(R_1)) \\
& A^{Del}(x,z) \text{:-} A(x,y), A^{Del}(y,z). & (\theta^{Del}(R_1)) \\
& A^{Red}(x,z) \text{:-} A^{Del}(x,z), A^{New}(x,y), A^{New}(y,z). & (\theta^{Red}(R_1)) \\
& A^{Ins}(x,z) \text{:-} A^{Ins}(x,y), A^{New}(y,z). & (\theta^{Ins}(R_1)) \\
& A^{Ins}(x,z) \text{:-} A^{New}(x,y), A^{Ins}(y,z).\} & (\theta^{Ins}(R_1)) \\
\mathcal{LP}_M = & \theta_{idb}(A) \cup \theta_{edb}^{New}(A_{edb}) &
\end{array}$$

The invocation of the θ_{edb}^{New} generator on A_{edb} initiates the invocation of the $(\theta_1^{New}(A_{edb}))$ and $(\theta_3^{New}(A_{edb}))$ generators and collects their results. Similarly, the invocation of the θ_{idb} generator on A leads to the invocation of $rules$ on A to retrieve the rules R_1, R_2 and the invocation of $\theta_1^{New}, \ldots, \theta^{Ins}(R_1)$.

4.4 Size of Maintenance Programs

As we can see from example 2 the size of the maintenance program $\mathcal{LP}_M$ is substantially larger than the original program $\mathcal{LP}$.

4.4.1 OWL (DLP) Maintenance Programs The structure of the rules that are generated by translating OWL axioms into a logic program allows us to observe that the rewriting of each OWL inclusion axiom creates the following number of maintenance rules:

$$
\begin{aligned}
|\theta(\phi_{\mathcal{LP}}(C \sqsubseteq D))| &= 3 \\
|\theta(\phi_{\mathcal{LP}}(C_1 \sqcap \ldots \sqcap C_n \sqsubseteq D))| &= 2 * n + 1 \\
|\theta(\phi_{\mathcal{LP}}(C \sqsubseteq D_1 \sqcap \ldots \sqcap D_n))| &= n * |\theta(\phi_{\mathcal{LP}}(C \sqsubseteq D_i))| \\
|\theta(\phi_{\mathcal{LP}}(D_1 \sqcup \ldots \sqcup D_n \sqsubseteq E))| &= n * |\theta(\phi_{\mathcal{LP}}(D_i \sqsubseteq E))| \\
|\theta(\phi_{\mathcal{LP}}(C \sqsubseteq \forall R.D))| &= |\theta(\phi_{\mathcal{LP}}(\exists R.C \sqsubseteq D))| = 5
\end{aligned}
$$

OWL object property transitivity is translated to five maintenance rules by applying θ to $\phi_{\mathcal{LP}}$. All other DL property axioms are translated to three maintenance rules by applying θ to $\phi_{\mathcal{LP}}$. θ is applied to all atomic classes and properties in $\mathcal{KB}_0^{DLP}$ as well as the auxiliary classes that are created by the structural transformation which is carried out during the preprocessing step.

4.4.2 RDF(S) Maintenance Programs Since the 12 static Datalog rules for the single predicate-based axiomatization of RDF(S) (cf. Table 2) contain 19 body predicates, the application of θ leads to the generation of 60 rules, namely 19 insertion rules, 19 deletion rules, 12 re-derivation rules, 5 maintenance rules for t^{New}, t^{+} and t^{-}, as well as 5 further rules to differentiate between entailments and assertions.

4.5 Evaluating Maintenance Programs

[30] show that the evaluation of the maintenance rules is a sound and complete procedure for computing the differentials between two database states when extensional update operations occur.

During the evaluation it is necessary to access the old state of a predicate. Bottom-up approaches to evaluation therefore require that all intensional relations involved in the computation are completely materialized.

The maintenance rules for capturing the new database state contain negated predicates to express the algebraic set difference operation. Hence, even though the original rules are pure Datalog (without negation), a program with negation is generated. The rewriting transformation keeps the property of stratifiability, since newly introduced predicates do not occur in cycles with other negations. Hence, it is guaranteed that predicates can be partitioned into strata such that no two predicates in one stratum depend negatively on each other, i.e. predicates only occur negatively in rules that define predicates of a higher stratum. The evaluation can then proceed, as usual, stratum-by-stratum starting with the extensional predicates themselves.

Example 3 (Evaluating Maintenance Programs). The direct links between members of the Bach family in Figure 3 constitute the extension of A_{edb}, where we abbreviate the names of each individual by the first letters of their forenames:

$$A_{edb} = \{(j,h),(j,c),(h,jc1),(jc1,jm),(jm,mb),(mb,wf),(js,wf),(ja,js),(c,ja)\}$$

Using the maintenance rewriting the materialization of A changes to A^{New} as follows, if $A^{Ins} = (js, jc2)$ is inserted and $A^{Del} = (js, wf)$ is deleted:

$$
\begin{aligned}
A^{Ins}_{edb} &= \{(jc, jc2)\} \\
A^{Del}_{edb} &= \{(js, wf)\} \\
A^{New}_{edb} &= A_{edb} \cup A^{Ins}_{edb} \setminus A^{Del}_{edb} \\
A^{Ins} &= \{(js, jc2), (ja, jc2), (c, jc2), (j, jc2)\} \\
A^{Del} &= \{(js, wf), (ja, wf), (c, wf), (j, wf)\} \\
A^{Red} &= \{(j, wf)\} \\
A^{New} &= (A \setminus A^{Del} \cup A^{Ins} \cup A^{Red}) \\
&= A \cup \{(js, jc2), (ja, jc2), (c, jc2), (j, jc2)\} \setminus \{(js, wf), (ja, wf), (c, wf)\} \\
A^{-} &= \{(js, wf), (ja, wf), (c, wf)\} \\
A^{+} &= \{(js, jc2), (ja, jc2), (c, jc2), (j, jc2)\}
\end{aligned}
$$

Since all maintenance rules of a given predicate have to be evaluated, an axiomatization of RDF(S) based on a single ternary predicate leads to complete re-computation in case of updates. We sketch an optimization for this case in Section 6 which should result in more efficient evaluation for the single predicate axiomatization.

5 Maintaining Changing Rules

This section presents the maintenance of a materialization if the definition of rules changes, i.e. rules that define a predicate are added or removed in the source program. We introduce two simple extensions to the rewriting-based approach presented in the previous section. Firstly, the materialization of predicates has to be maintained in the case of changes. Secondly, the maintenance programs have to be maintained such that additional rewritings are introduced for new rules and irrelevant rewritings are removed for deleted rules.

5.1 Approach

We illustrated in Section 3.2 that every change in the rule set might cause changes in the extension of an intensional predicate P, with the consequence that the materialization of intensional predicates has to be updated. However, unlike in the case of changing extensions, both auxiliary predicates which capture the differences which are used to update the materialization of some predicate $P \in \mathbf{P}_{idb}$, i.e. P^+ and P^- have empty extensions since no actual facts change.

Obviously, we can categorize the intensional predicates that are affected by a change in rules into two sets: *(I)* predicates that are directly affected, i.e. occur in the head of changed rules and *(II)* predicates that are indirectly affected, i.e. by depending on directly affected predicates through the rules in the program.

Our solution uses the existing maintenance rewriting for facts to propagate updates to the *indirectly affected* predicates. To achieve this, the maintenance computation for *directly affected* predicates is integrated into the mainte-

nance program by redefining the auxiliary predicates that are used to propagate changes between predicates, i.e. P^{New}, P^{Ins} and P^{Del}.

5.2 Maintenance Rewriting

Let $\delta^+(\delta^-)$ be the set of rules which are inserted (deleted) from the logic program $\mathcal{LP}$. The reader may recall from the previous section that the function $head : \mathcal{LP} \rightarrow P_{idb}$ maps a rule to its rule head, and the function $rules : P_{idb} \rightarrow \mathcal{LP}$ maps rule heads to rules.

Definition 2 (Directly affected predicate). *An intensional predicate* $p \in P_{idb}$ *is a* directly affected *predicate, if* $p \in \{head(r) | r \in \delta^+ \cup \delta^-\}$.

Generator	Parameter	Rewriting Result
	Predicate	
ϑ	$P \in \mathbf{P}_{idb}$	$\{\vartheta^{Ins}_{idb}(P)\} \cup \{\vartheta^{Del}_{idb}(P)\} \cup \{\vartheta^{New}_{idb}(P)\}$
ϑ^{Ins}_{idb}	$P \in \mathbf{P}_{idb}$	P^{Ins} :- P^{New}.
ϑ^{Del}_{idb}	$P \in \mathbf{P}_{idb}$	P^{Del} :- P.
ϑ^{New}_{idb}	$P \in \mathbf{P}_{idb}$	$\{\vartheta^{New}(r) \| \forall r \in rules(P)\}$
	Rule	
ϑ^{New}	H :- $B_1, \ldots, B_n$.	H^{New} :- $B_1^{New}, \ldots, B_n^{New}$.

Table 5. Rewriting Functions ϑ

The rule structure of the maintenance program $\mathcal{LP}_M$ is modified for all directly affected predicates. For these predicates all maintenance rules are substituted by maintenance rules generated using the ϑ rewriting function (cf. Table 5). ϑ is instantiated for every directly affected predicate P and the following is done:

1. The existing rules defining P^{New} in the maintenance program $\mathcal{LP}_M$ are deleted;
2. New rules axiomatize P^{New} using the (new) rule set that defines P in the updated original program. These rules are slightly adapted, such that references to any predicate P are altered to P^{New}, by instantiating the following rewriting pattern for all rules $R \in rules(P)$:

$$P^{New} \text{ :- } R_1^{New}, \ldots, R_n^{New}.$$

The rewrite pattern simply states that the new state of the predicate P follows directly from the combination of the new states of the predicates R_i in the body of of all rules defining P in the changed source program.

3. All maintenance rules for calculating the insertions and deletions to P have to be removed from the maintenance program and are replaced by the following two static rules.

$$P^{Ins} \text{:-} P^{New}.$$
$$P^{Del} \text{:-} P.$$

The role of $P^{Ins}, P^{Del}, P^{New}$ is exactly the same as in the rewriting for facts, i.e. they propagate changes to dependent predicates. While P^{Ins} propagates the new state of a predicate as an insertion to all dependent predicates, P^{Del} propagates the old state of a predicate as a deletion to all dependent predicates. Figure 4 shows how the information flow in the maintenance program changes with respect to the rewriting of a rule $H(x)\text{:-}B(x)$. from the maintenance rewriting for fact changes *(a)* to the maintenance for rule changes *(b)*. The arrows to (from) nodes depict that the respective predicate possibly uses (is used by) some other predicate in the maintenance program.

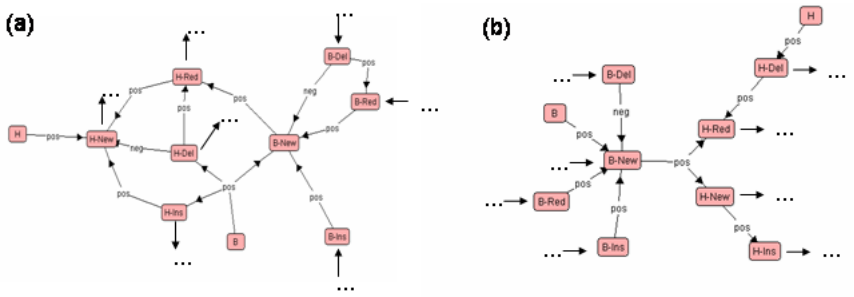

Fig. 4. Information Flow in Maintenance Programs: (a) Maintenance for Facts; (b) Maintenance for Rules

5.3 Evaluating Maintenance Programs

The evaluation of maintenance programs is now carried out in three steps:

1. Update the maintenance rewriting $\mathcal{LP}_M$ of $\mathcal{LP}$ to incorporate the set of rules that are added (δ^+) or removed (δ^-).
2. Evaluate the maintenance program $\mathcal{LP}_M$ and incrementally maintain all materialized predicates.
3. Maintain the maintenance rewriting $\mathcal{LP}_M$ by changing rewritings back to the rewriting for facts.

Require:
δ^+ Set of inserted rules
δ^- Set of deleted rules
$\mathcal{LP}_M$ Maintenance program
Ensure:
Updated maintenance program $\mathcal{LP}_M$

removeMR $= \emptyset$ *// Collects maintenance rules to be removed*
addMR $= \emptyset$ *// Collects maintenance rules to be added*
affectedPred $= \emptyset$ *// Collects all affected predicates*
// Add new rewriting rules for added rules
for all $r \in (\delta^+ \setminus \delta^-)$ **do**
  addMR $= \{\theta^{Red}(r)\} \cup$ addMR
  $p =$ head(r)
  affectedPred $= \{p\} \cup$ affectedPred
  // First rule defining a predicate ?
  if $p \notin \{head(r) \mid r \in \mathcal{LP}\}$ **then**
    addMR $= \theta^+_{idb}(p) \cup \theta^-_{idb}(p) \cup$ addMR *// Need new auxiliary predicates*
  end if
  $\mathcal{LP} = \mathcal{LP} \cup r$
end for
// Add new rewriting rules for deleted rules
for all $r \in (\delta^- \setminus \delta^+)$ **do**
  $p =$ head(r)
  affectedPred $= p \cup$ affectedPred
  // Last rule defining a predicate ?
  if $rules(p) \setminus \{r\} = \emptyset$ **then**
    removeMR $= \theta^+_{idb}(p) \cup \theta^-_{idb}(p) \cup \theta^{Red}_{idb}(p) \cup$ removeMR
  end if
  $\mathcal{LP} = \mathcal{LP} \setminus r$
end for
// Replace rewriting rules for affected predicates
for all $p \in$ affectedPred **do**
  addMR $= \vartheta^{New}_{idb}(p) \cup \vartheta^{Ins}_{idb}(p) \cup \vartheta^{Del}_{idb}(p) \cup$ addMR
  removeMR $= \theta^{New}_{idb}(p) \cup \theta^{Ins}_{idb}(p) \cup \theta^{Del}_{idb}(p) \cup$ removeMR
end for
$LP_M = (LP_M \cup$ addMR $) \setminus$ removeMR

Algorithm 5.1: Updating Rules (Pre-Evaluation Algorithm)

Step 1. Step 1 is implemented by Algorithm 5.1. This algorithm has three functions. Firstly, it replaces all maintenance rewritings for directly affected predicates with the new maintenance rewritings. Secondly, it alters the source program $\mathcal{LP}$ such that the set of updated rules is incorporated into $\mathcal{LP}$. Thirdly, it maintains auxiliary rewriting rules, viz. generates rules for previously unknown intensional predicates and removes those rules if an intensional predicate no longer occurs in the source program.

Example 4 (Maintenance Rewritings for New Rule). Let us return to the maintenance program $\mathcal{LP}_M$ established in Example 2 and consider that rule R_3 is inserted into $\mathcal{LP} = \{R_1, R_2\}$, i.e. the new $\mathcal{LP}$ consists of the following rules after the application of Algorithm 5.1:

(R_1) ancestorOf(x, z) :- ancestorOf(x, y), ancestorOf(y, z).
(R_2) ancestorOf(x, z) :- ancestorOf$_{edb}(x, y)$.
(R_3) inDynasty(x, y) :- ancestorOf(x, y).

Since $\delta^+ = R_3$ and $\delta^- = \emptyset$ and none of the previously existing intensional axioms is directly affected, the algorithm does not remove any rewriting rules from the maintenance program $\mathcal{LP}_M$ in this example. We have, however, to add the new maintenance rules for the directly affected predicate inDynasty, which we will abbreviate as I in the following. The algorithm augments $\mathcal{LP}_M$ with the rules generated by the following calls to rewriting generators (in this order):

$\theta^{Red}(R_3) = I^{Red}(x, y)$:- $I^{Del}(x, y)$, ancestorOf$^{New}(x, y)$.
$\theta^{+}_{idb}(I) = \quad I^{+}(x, y)$:- $I^{Ins}(x, y)$, **not** $I(x, y)$.
$\theta^{-}_{idb}(I) = \quad I^{-}(x, y)$:- $I^{Del}(x, y)$, **not** $I^{Ins}(x, y)$, **not** $I^{Red}(x, y)$.
$\vartheta^{New}(I) = I^{New}(x, y)$:- ancestorOf$^{New}(x, y)$.
$\vartheta^{Ins}(I) = \quad I^{Ins}(x, y)$:- $I^{New}(x, y)$.
$\vartheta^{Del}(I) = \quad I^{Del}(x, y)$:- $I(x, y)$.

The new state of I is now directly derived from the new state of A, which is calculated as part of the maintenance program. Hence, we can obtain the first materialization I just by evaluating the maintenance program.

Step 2. Step 2 evaluates the maintenance program as presented in Section 4.

Step 3 Step 3 is implemented by Algorithm 5.2. It essentially only undoes our special maintenance rewriting, i.e. it replaces the maintenance rewritings that have been generated by Algorithm 5.1 for directly affected predicates with the normal maintenance rewritings for facts.

Example 5 (Maintenance Rewritings for New Rule). Algorithm 5.2 would remove the following maintenance rules from the maintenance program $\mathcal{LP}_M$:

$\vartheta^{New}(I) = I^{New}(x, y)$:- ancestorOf$^{New}(x, y)$.
$\vartheta^{Ins}(I) = \quad I^{Ins}(x, y)$:- $I^{New}(x, y)$.
$\vartheta^{Del}(I) = \quad I^{Del}(x, y)$:- $I(x, y)$.

In parallel, the maintenance program would be extended with the rewritings generated by the rewriting generators that create the maintenance rewriting for facts ($\theta^{New}_{idb}(I), \theta^{Ins}_{idb}(I)$ and $\theta^{Del}_{idb}(I)$).

Since all maintenance rules for dealing with changes in rules are removed by Algorithm 5.2, we obtain the same maintenance program as if we would have completely regenerated the maintenance program for facts from the changed source program.

Require:
δ^+ Set of inserted rules
δ^- Set of deleted rules
$\mathcal{LP}$ Original logic program
$\mathcal{LP}_M$ Maintenance program
Ensure:
Updated logic program $\mathcal{LP}$
Updated maintenance program $\mathcal{LP}_M$

removeMR $= \emptyset$
addMR $= \emptyset$
affectedPred $= \emptyset$
for all $r \in (\delta^+ \setminus \delta^-)$ **do**
affectedPred $= head(r) \cup$ affectedPred
end for
for all $r \in (\delta^- \setminus \delta^+)$ **do**
$p =$ head(r)
if $rules(p) \neq \emptyset$ **then**
affectedPred $= p \cup$ affectedPred
end if
end for
for all $p \in$ affectedPred **do**
removeMR $= \vartheta_{idb}^{New}(p) \cup \vartheta_{idb}^{Ins}(p) \cup \vartheta_{idb}^{Del}(p) \cup$ removeMR
addMR $= \theta_{idb}^{New}(p) \cup \theta_{idb}^{Ins}(p) \cup \theta_{idb}^{Del}(p) \cup$ addMR
end for
$LP_M = LP_M \cup$ addMR $\setminus$ removeMR

Algorithm 5.2: Updating Rules (Post-Evaluation Algorithm)

6 Materializing RDF Rules

An alternative to using OWL TBox axioms to state that INDYNASTY is a symmetric property, is the usage of either one of the RDF-based rule languages (cf. Section 2), e.g. Notation 3.

Notation 3	`{ ?x :inDynasty ?y. }` `log:implies` `{ ?y :inDynasty ?x. }.`
Datalog	$T(x, \texttt{inDynasty}, y)\texttt{:-}T(y, \texttt{inDynasty}, x).$

Table 6. Datalog Translation of RDF Rule Languages

If an RDF rule system internally uses one single predicate within the rules, however, our technique for incrementally maintaining the materialization in case of changes is useless. The evaluation of the maintenance program then corresponds to a total recomputation, since all rules defining this predicate have to be evaluated.

In order to use our approach to materialization, more optimized data structures to represent an RDF graph have to be chosen, such that the part of the knowledge base which takes part in the evaluation can be limited.

6.1 Selection-based Optimization

We will briefly sketch a possible optimization, which we called *selection-based optimization* [35]. The optimization is based on the idea to split the extension of the RDF graph according to *split points*, which are given by constants that occur at a certain argument position of a predicate. Useful split points can be derived from the vocabulary of an ontology or an ontology language such as RDF Schema. In case of arbitrary graph data, a useful split point can be frequently occurring constants, which can be easily determined using counting. The choice of a good split point, however, clearly depends on the application of the RDF rule base.

We can transform a Datalog program into an equivalent program that incorporates split points, if all references to a predicate P (in queries, facts and rules) where a split point occurs are replaced by appropriate split predicates.

In the following, we will assume that a split point is constituted by a constant c that is used as the i-th argument in the predicate P. To generate split predicates, we then split the extension of a predicate P_{edb} into several edb predicates of the form $P_{edb}^{c_i}(Var_1, Var_2, \ldots, Var_{i-1}, c, Var_{i+1}, Var_n)$ to store tuples based on equal constant values c in their i-th component.

Hence, instead of using a single extensional predicate P_{edb} for representing direct RDF assertions, the extensional database is split into several $P_{edb}^{c_i}$. Again, we can differentiate between asserted and derived information by introducing intensional predicates (views) for each component of the extension (i.e. rules

of the form P^{c_i} :- $P^{c_i}_{edb}$).The complete predicate P can still be represented by means of an intensional predicate, which is axiomatized by a collection of rules that unify the individual split predicates: P :- P^{c_i}.

Example 6. Returning to the triple based axiomatization (cf. Figure 2) of the N3 example, we can transform the program by introducing a split point $T^{inDynasty_2}$ for the inDynasty constant (when used as second argument in the ternary predicate T):

- We use two extensional predicates: $T^{Rest}_{edb}, T^{inDynasty_2}_{edb}$ to store the extension in two disjoint sets.
- We capture the intensional predicates and integrate the splits into a complete extension of T and rewrite the example such that split predicates are used instead of the full predicate:

$$
\begin{array}{ll}
T^{Rest}(X,Y,Z) & \text{:- } T^{Rest}_{edb}(X,Y,Z). \\
T^{inDynasty_2}(X,Y,Z) & \text{:- } T^{inDynasty_2}_{edb}(X,Y,Z). \\
T(X,Y,Z) & \text{:- } T^{Rest}(X,Y,Z). \\
T(X,Y,Z) & \text{:- } T^{inDynasty_2}(X,Y,Z). \\
T^{inDynasty_2}(X,\text{inDynasty},Y) & \text{:- } T^{inDynasty_2}(Y,\text{inDynasty},X).
\end{array}
$$

Any other rule that is inserted into the RDF rule base can be transformed into a set of rules, which use the available split predicates.

However, the maintenance of a materialized predicate $T^{inDynasty_2}$ can now be carried out by ignoring all non-relevant rules for T. Hence, the whole extension of T can be updated via the insert and delete maintenance rules that were presented in the previous sections, i.e. without using the complete database.

7 Implementation

The incremental maintenance of materializations is implemented in the KAON Datalog engine[12], which handles materialization on a per predicate, i.e. per class or property, level. In case of the materialization of a predicate all changes to facts relevant for the predicate and the rule set defining a predicate are monitored.

The maintenance process is carried out as follows. When a program is designated for materialization, all maintenance rules are generated, the maintenance program itself is evaluated and the extension of all predicates P designated for materialization is stored explicitly. The maintenance program is then used for evaluation instead of the original program which is kept as auxiliary information to track changes to rules. All rules of the original program which define non-materialized predicates are added to the maintenance program.

[12] The engine is part of the open source KAON tool suite, which can be freely downloaded from `http://kaon.semanticweb.org/`.

Updates to facts are handled in a transactional manner. All individual changes are put into the appropriate p_{edb}^{Ins} and p_{edb}^{Del} predicates. Committing the transaction automatically triggers the evaluation of the maintenance rules. After this evaluation, the extensions of all materialized predicates P are updated by adding the extension of P_{idb}^{+} and removing the extension of P_{idb}^{-}. Similarly, the extension of all extensional predicates P_{edb} is updated by adding P^{Ins} and removing P^{Del}. As a last step, the extension of P^{Ins} and all other auxiliary predicates are cleared for a new evaluation.

Changes in rules are carried out in the three phase process described in Section 5: First, the new maintenance rules of rule change are generated. Then, the maintenance program is evaluated and the extensions of materialized predicates are updated as described for the change of facts. As the last step, the maintenance rules for rule change are replaced with maintenance rules for fact changes.

8 Evaluation

This section reports on the evaluation of our approach which was carried out with various synthetically generated OWL ontologies that are expressible in the DLP fragment.

8.1 Evaluation Setting

Test Assumptions. The evaluation has been carried out with changing OWL ontologies that are operationalized in logic databases using the DLP approach. It is assumed that all predicates are materialized. We assume that an inference engine builds its knowledge base by aggregating data from several web sources. Therefore bulk updates will be predominant.

Test Procedure. Each test is characterized by a certain ontology structure and a class whose extension is read. The ontology structure has been generated for different input parameters, resulting in ontologies of different sizes. The average of five such invocations has been taken as the performance measure for each test.

We obtain six measures: *(a)* the time of query processing without materialization, *(b)* the time required to set up the materialization and the maintenance program, *(c)* the time required to perform maintenance when rules are added, *(d)* rules are removed, *(e)* facts are added, and *(f)* facts are removed. Finally, *(g)* assesses the time of query processing with materialization.

Test Platform. We performed the tests on a laptop with Pentium IV Mobile processor running at 2 GHz, 512 MB of RAM using the Windows XP operating system. The implementation itself is written in Java and executed using Sun's JDK version 1.4.1_01.

8.2 Evaluation Scenarios

First we give an overview of the types of tests we conducted. In the following we use D to denote the depth of the class hierarchy, NS to denote the number of sub classes at each level in the hierarchy, NI to denote the number of instances per class and P to denote the number of properties.

To test changes in facts, we add and remove a random percentage *Change* of the facts. For rules, we add and remove a random rule. This is due to the limitation of the underlying engine, which currently does not allow to alter rules in a bulk manner. The test was performed for different depths of the taxonomy $D = 3, 4, 5$ while the number of sub classes and the number of instances was not altered ($NS = 5$; $NI = 5$). Test 2 and 3 made use of properties. Here, every class had five properties, which are instantiated for every third instance of the class ($NI = 5$). We carried out each test using varying *Change* ratios of 10% and 15% of the facts.

Test 1: Taxonomy Extended taxonomies, e.g. WordNet, currently constitute a large portion of the ontologies that are in use. Our goal with this test is to see how the very basic task of taking the taxonomy into account when retrieving the extension of a class is improved. The taxonomy is constituted by a symmetric tree of classes. We did not make use of properties, hence $P = 0$. The test query involved computing the extension of one of the concepts on the first level of the class hierarchy. This is a realistic query in systems where taxonomies are used for navigation in document collections. Here, navigation typically starts with top-level classes and the set of documents is displayed as the class extension.

Test 2: Database-like The goal of this test was to see how ontologies with larger number of properties are handled. Our goal was to answer a simple conjunctive query on top of this ontology. The DL-like query is $\mathrm{c1} \sqcap \exists \mathrm{p0.c12}$.

Test 3: DL-like This test shows how materialization performs in DL-like ontologies, which contain simple class definitions. Each class in the class tree is defined using the following axiom: $\mathrm{c}_i \sqcup \exists \mathrm{p}_k.\mathrm{c}_{i-1} \sqsubseteq \mathrm{c}$ (where c_i denotes i-th child of concept c). The query retrieves the extension of some random class in the first-level of the taxonomy.

8.3 Results

Figure 5 depicts the average time[13] for querying an ontology without using materialization, setting up the materialization and cost of maintenance for different types of changes (adding and removing rules and facts). Finally, the time for answering the same query using the materialization is depicted. The exact results of the evaluation can be found in the appendix.

As we can see in the appendix, maintenance costs do not vary significantly with the quantity of updates, therefore Figure 5 only shows the results for 10%

[13] in milliseconds on a logarithmic scale

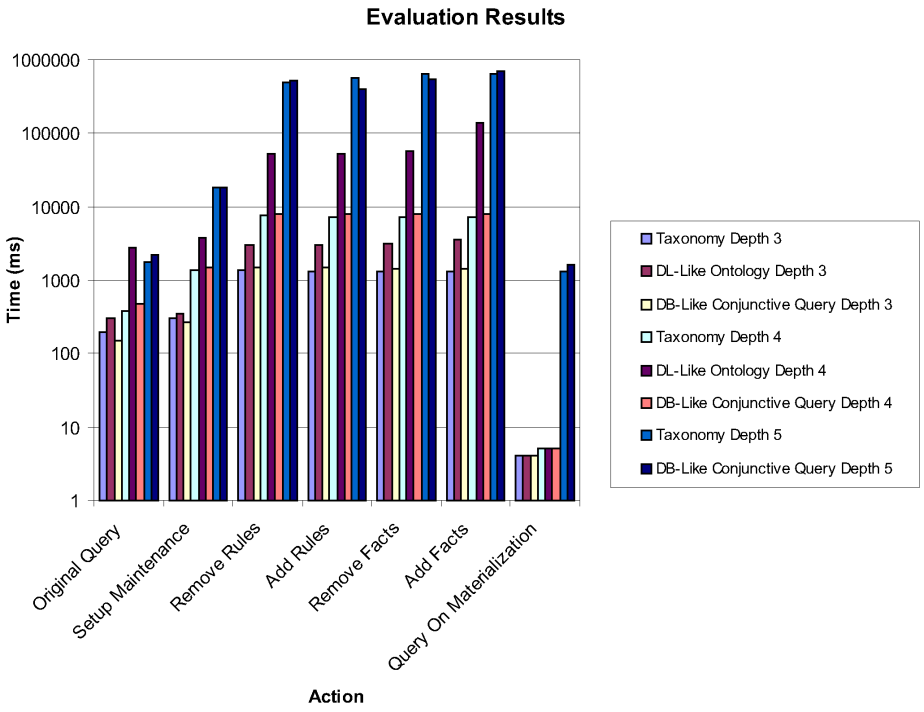

Fig. 5. Evaluation Results (Average Values for 10% change)

change. All costs are directly related to the size of the ontologies. The performance behavior between the taxonomy and DB-like ontologies do also not alter significantly. However, more complex rules as they are constituted by DL-like ontologies are always more expensive to evaluate, therefore setup costs and the cost of evaluating the maintenance rules is also higher.

We want to stress that we measured the performance of concrete tools. Although algorithms implemented by a system are certainly important, the overall performance of a system is influenced by many other factors as well, such the quality of the implementation or the language. It is virtually impossible to exclude these factors from the performance measurement. For example, our Datalog engine ran out of memory with the DL-like ontology where the taxonomic depth was five, viz. the set of rules was generated from 3950 class and 19750 property definitions, while the underlying knowledge base contained 19750 class instantiations and 32915 property instantiations. The other ontologies of taxonomic depth 5 were still handled by the engine, but due to inefficient memory management, most of the time was not actually used for query processing but for memory management (swapping), such that the query on the materialization in this only showed little improvement.

8.4 Discussion

The different costs of each step in the maintenance procedure are always higher than the costs of evaluating a single query. The question whether or not to materialize is therefore determined by the application and the issue whether the system can handle its typical workload, e.g. can it handle the intended number of users if answering a single query takes almost 3 seconds ?

With materialization the cost of accessing the materialized predicates can be neglected. However, the time for the evaluation of the maintenance rules can be a significant bottleneck for a system especially for large knowledge-bases. For example, in one of our test runs it took almost 16 minutes to recompute the materialization after fact changes for the DB-like test with taxonomic depth 5. Fortunately, materialization can be carried out in parallel to answering queries on top of the existing materialization.

In consequence, users will have to operate on stale copies of data. Staleness of data cannot be avoided in distributed scenarios like the Web in the first place, and existing experiences, e.g. with outdated page ranks of a web pages in Google, show that the quality of query answering is still good enough, if data is updated occasionally.

9 Related Work

We can find related work in two areas: Firstly, incremental maintenance of materialized views in deductive databases. Secondly, truth maintenance systems in the Artificial Intelligence context.

9.1 Incremental Maintenance of Materialized Views

Several algorithms have been devised for the incremental maintenance of materialized views in deductive databases. All of these approaches do not consider changes in the set of rules and differ in the techniques used to cope with changes in facts.

In order to cope with changing facts, [1,18] effectively compute the Herbrand model of a stratified database after a database update. The proposed solution of [1] uses sets of positive and negative dependencies that are maintained for all derived facts. This leads to low space efficiency and high cost for maintaining the dependencies. [18] derives rules (so-called meta-programs) to compute the difference between consecutive database states for a stratified Datalog program. Some of the generated rules are not safe, making it impossible to implement the rules in Datalog engines. Additionally, duplicate derivations are not discarded in the algorithm.

[12] presents the Delete and Re-Derive (DRed) algorithm, which is a procedural approach to view maintenance in Datalog with stratified negation. We will follow their principal approach for the computation of changes, in fact their procedural algorithm has been altered to a declarative algorithm [31] which we will extend.

The Propagation Filtration algorithm of [13] is similar to the DRed algorithm, except that changes are propagated on a 'predicate by predicate' basis. Hence, it computes changes in one intensional predicate due to changes in one extensional predicate, and loops over all derived and extensional predicates to complete the maintenance procedure. In each step of the loop, the delete, rederive and insert steps are executed. The algorithm ends up fragmenting computation and rederiving changed and deleted facts over and over again, i.e. it is less efficient than the DRed algorithm.

9.2 Truth Maintenance Systems (TMS)

Truth maintenance [14] is an area of AI concerned with revising sets of beliefs and maintaining the truth in a reasoning system when new information alters existing information. A representation of beliefs and their dependencies is used to achieve the retraction of beliefs and to identify contradictions. For example, justification-based TMS [9] uses a graph data structure where nodes are augmented with two fields indicating their belief status and supporting justification. When the belief status is changed, dependencies are propagated through the graph.

Making TMSs more efficient was a cottage industry in the late 1980s, with most of the attention focused on the Assumption-based TMS (ATMS) [6]. The primary advantage of the ATMS is its ability to rapidly switch among many different contexts, which allows a simpler propagation of fact withdrawals, but comes at the cost of an exponential node-label updating process when facts are added. The main disadvantage of TMS is that the set of justifications (and nodes) grows monotonically as it is not allowed to retract a justification, but only disable information. The fact that the set of assumptions is always in flux introduces most of the complexity in the TMS algorithms. More recent work (e.g. [24]) primarily tried to reduce the cost for incremental updates.

10 Conclusion

10.1 Contribution

We presented a technique for the incremental maintenance of materialized Datalog programs. Our technique can therefore be applied for ontology languages which can be axiomatized in Datalog, i.e. RDF Schema and OWL DLP[15] as well as the Datalog-fragments of Semantic Web rule languages.

We contributed a novel solution to the challenge of updating a materialization incrementally when the rules of a Datalog program change, which has, to our best knowledge, not been addressed in the deductive database context[16].

[14] also called *belief revision* or *reason maintenance*.

[15] We cannot maintain function symbols other than constants, therefore our approach can not be used for $\mathcal{L}_3$.

[16] [11] address the maintenance of views after redefinition for the relational data model.

In order to cope with changing rules, we applied a declarative, rewriting-based algorithm for the incremental maintenance of views [31] and introduced two novel techniques: Firstly, we extended the rewriting to deal with changing rules. Secondly, we introduced two algorithms for the maintenance of the rewritten rules when the underlying source rules change.

Our solution has been completely implemented and evaluated. We reported on our prototypical implementation and presented the results of our empirical analysis of the costs of incremental maintenance, which shows the feasibility of our solution.

The techniques proposed in this article are not specific to any ontology language, but can generally be used for the incremental maintenance of materialized Datalog programs. Due to this generic solution, future developments, e.g. for the rule layer of the Semantic Web, are likely to benefit from our technique as well.

Materialization is certainly not a panacea to all tractability problems. For example, one drawback is that it trades off required inferencing time against storage space and access time. In spite of this restriction, which remains to be assessed by more practical experience and cost models that are derived from those experiences, we conjecture that materialization as explained in this article will help to progress the Semantic Web and to build the large Semantic Web engines of tomorrow.

10.2 Further Uses

We can reuse our approach for incremental maintenance of a materialization in several other contexts:

- *Integrity Constraint Checking*: Incremental maintenance can also be used as a fundamental technique in an implementation of integrity constraints on Semantic Web data, i.e. we can incrementally check the validity of a constraint by maintaining an empty view.
- *Continuous Queries*: [19] The auxiliary maintenance predicates P^+ and P^- can be used as a basis for implementing continuous queries or publish/subscribe systems, which are used to monitor a flow of data. This monitoring can use the extensions of P^+ and P^- as a basis for notification messages that are sent to the subscribers.
- *Interoperability with systems of limited inferencing capabilities*: We can use materialization to explicate data for clients that cannot entail information on their own. In particular, we can store materializations in relational databases which are agnostic about the semantics of the data but may be used for fast query answering.

References

1. K. Apt and J.-M. Pugin. Maintenance of stratified databases viewed as belief revision system. In *Proc. of the 6th Symposium on Principles of Database Systems (PODS)*, pages 136–145, San Diego, CA, USA, March 1987.

2. S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. OilEd: A reasonable ontology editor for the Semantic Web. In *Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence*, volume 2174 of *LNAI*, pages 396–408. Springer, 2001.
3. S. Bechhofer, R. Volz, and P. Lord. Cooking the Semantic Web with the OWL API. In *[10]*, pages 659–675, 2003.
4. T. Berners-Lee. CWM - closed world machine. Internet: `http://www.w3.org/2000/10/swap/doc/cwm.html`, 2000.
5. D. Brickley and R. V. Guha. RDF vocabulary description language 1.0: RDF Schema. W3C Working Draft, 10 October 2003, October 2003. Internet: `http://www.w3.org/TR/2003/WD-rdf-schema-20031010/`.
6. J. de Kleer. An assumption-based truth maintenance system. *Artificial Intelligence*, 28:127–162, 1986.
7. S. Decker, D. Brickley, J. Saarela, and J. Angele. A query and inference service for RDF. In *QL98 - Query Languages Workshop*, December 1998.
8. S. Decker, M. Erdmann, D. Fensel, and R. Studer. Ontobroker: Ontology based Access to Distributed and Semi-structured Information. In *Database Semantics: Semantic Issues in Multimedia Systems.*, pages 351–369. Kluwer Academic, 1999.
9. J. Doyle. A truth maintenance system. In B. Webber and N. J. Nilsson, editors, *Readings in Artifcial Intelligence*, pages 496–516. Morgan Kaufmann, Los Altos, California, 1981.
10. D. Fensel, K. Sycara, and J. Mylopoulos, editors. *The Semantic Web - ISWC 2003, Second International Semantic Web Conference, Sanibel Island, USA, October 9-12, 2002, Proceedings*, volume 2870 of *Lecture Notes in Computer Science*. Springer, 2003.
11. A. Gupta, I. S. Mumick, and K. A. Ross. Adapting materialized views after redefinitions. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 211–222. ACM Press, 1995.
12. Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 157–166. ACM Press, 1993.
13. J. Harrison and S. Dietrich. Maintenance of materialized views in a deductive database: An update propagation approach. In *Workshop on Deductive Databases held in conjunction with the Joint International Conference and Symposium on Logic Programming (JICSLP)*, pages 56–65, Washington, D.C., November 1992.
14. P. Hayes. RDF Semantics. W3C Working Draft, 10 October 2003, October 2003. Internet: `http://www.w3.org/TR/rdf-mt/`.
15. J. Heflin, J. Hendler, and S. Luke. SHOE: A knowledge representation language for internet applications. Technical Report CS-TR-4078, Institute for Advanced Computer Studies, University of Maryland, 1999.
16. M. Jarke, R. Gallersdoerfer, M. A. Jeusfeld, and M. Staudt. ConceptBase - A Deductive Object Base for Meta Data Management. *JIIS*, 4(2):167–192, 1995.
17. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42:741–843, 1995.
18. V. Kuchenhoff. On the efficient computation of the difference betwen consecutive database states. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proc. of 2nd Int. Conf. on Deductive and Object-Oriented Databases*, volume 566 of *Lecture Notes in Computer Science (LNCS)*, pages 478–502, Munich, Germany, December 1991. Springer.

19. L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE TKDE*, 11(4), 1999.
20. A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz. Managing multiple ontologies and ontology evolution in ontologging. In *Proc. of IIP-2002*, Montreal, Canada, 08 2002.
21. A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz. An infrastructure for searching, reusing and evolving distributed ontologies. In *Proc. of WWW-2003*, Budapest, Hungary, 05 2003.
22. D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. Technical report, World Wide Web Consortium (W3C), August 2003. Internet: `http://www.w3.org/TR/owl-features/`.
23. B. Motik, A. Maedche, and R. Volz. A conceptual modeling approach for semantics-driven enterprise applications. In *Proc. 1st Int'l Conf. on Ontologies, Databases and Application of Semantics (ODBASE-2002)*, October 2002.
24. P. P. Nayak and B. C. Williams. Fast context switching in real-time propositional reasoning. In T. Senator and B. Buchanan, editors, *Proceedings of the Fourteenth National Conference on Artificial Intelligence and the Ninth Innovative Applications of Artificial Intelligence Conference*, pages 50–56, Menlo Park, California, 1998. AAAI Press.
25. R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL Deductive System. *VLDB Journal: Very Large Data Bases*, 3(2):161–210, 1994.
26. J. De Roo. Euler proof mechanism. Internet: `http://www.agfa.com/w3c/euler/`, 2002.
27. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In R. T. Snodgrass and M. Winslett, editors, *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94)*, pages 442–453, 1994.
28. M. Sintek and S. Decker. TRIPLE - an RDF query, inference and transformation language. In *Deductive Databases and Knowledge Management (DDLP)*, 2001.
29. P. Spyns, D. Oberle, R. Volz, J. Zheng, M. Jarrar, Y. Sure, R. Studer, and R. Meersman. OntoWeb - A Semantic Web community portal. In *Proc. ofProc. Fourth International Conference on Practical Aspects of Knowledge Management (PAKM)*, pages 189–200, Vienna, Austria, 2002.
30. M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. Technical Report AIB-95-13, RWTH Aachen, 1995.
31. M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 75–86. Morgan Kaufmann, 1996.
32. R. Studer, Y. Sure, and R. Volz. Managing user focused access to distributed knowledge. *Journal of Universal Computer Science (J.UCS)*, 8(6):662–672, 2002.
33. R. Volz, S. Decker, and I. Cruz, editors. *Proc. of first Int. Workshop on Practical and Scalable Semantic Systems*, volume 89, Sanibel Island, Florida, USA, October 2003. CEUR Workshop Proceedings. Internet: `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-89/`.
34. R. Volz, S. Staab, and B. Motik. Incremental maintenance of dynamic datalog programs. In *[33]*, 2003.
35. R. Volz, S. Staab, and B. Motik. Incremental Maintenance of Materialized Ontologies. In R. Meersman, Z. Tari, D. C. Schmidt, B. Kraemer, M. van Steen, S. Vinoski, R. King, M. Orlowska, R. Studer, E. Bertino, and D. McLeod, editors, *Proc. of CoopIS/DOA/ODBASE 2003*, volume 2888 of *LNCS*, pages 707–724, Sicily, 2003.

36. Raphael Volz. *Web Ontology Reasoning in Logic Databases*. PhD thesis, Universitaet Fridericiana zu Karlsruhe (TH), `http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=2004%2Fwiwi%2F2`, February 2004.

A Appendix

The reader may note that OoM is an acronym for "Out Of Memory", i.e. the prototypical implementation could not deal with the problem size.

Original Query	D	NS	NI	P	Change	Orig	Average	Minimum	Maximum
Taxonomy	3	5	5	0	10		197	80	491
Taxonomy	4	5	5	0	10		373	290	571
Taxonomy	5	5	5	0	10		1767	1482	2463
Taxonomy	3	5	5	0	15		147	60	311
Taxonomy	4	5	5	0	15		378	280	581
Taxonomy	5	5	5	0	15		1765	1373	2464
DL-Like Ontology	3	5	5	5	10		310	170	640
DL-Like Ontology	4	5	5	5	10		2764	2523	3475
DL-Like Ontology	5	5	5	5	10		OoM	OoM	OoM
DL-Like Ontology	3	5	5	5	15		263	150	511
DL-Like Ontology	4	5	5	5	15		2774	2523	3515
DL-Like Ontology	5	5	5	5	15		OoM	OoM	OoM
DB-Like Conjunctive Query	3	5	5	5	10		152	70	341
DB-Like Conjunctive Query	4	5	5	5	10		482	310	701
DB-Like Conjunctive Query	5	5	5	5	10		2165	19963	2403
DB-Like Conjunctive Query	3	5	5	5	15		172	70	430
DB-Like Conjunctive Query	4	5	5	5	15		425	301	701
DB-Like Conjunctive Query	5	5	5	5	15		2078	1722	2374

Setup Maintenance	D	NS	NI	P	Change	Average	Minimum	Maximum
Taxonomy	3	5	5	0	10	305	200	441
Taxonomy	4	5	5	0	10	1347	1212	1622
Taxonomy	5	5	5	0	10	18391	16694	19318
Taxonomy	3	5	5	0	15	245	141	251
Taxonomy	4	5	5	0	15	1382	1232	1683
Taxonomy	5	5	5	0	15	18293	16714	19017
DL-Like Ontology	3	5	5	5	10	355	230	531
DL-Like Ontology	4	5	5	5	10	3715	2894	4747
DL-Like Ontology	5	5	5	5	10	OoM	OoM	OoM
DL-Like Ontology	3	5	5	5	15	368	241	571
DL-Like Ontology	4	5	5	5	15	3720	2894	4757
DL-Like Ontology	5	5	5	5	15	OoM	OoM	OoM
DB-Like Conjunctive Query	3	5	5	5	10	265	151	431
DB-Like Conjunctive Query	4	5	5	5	10	1464	1322	1663
DB-Like Conjunctive Query	5	5	5	5	10	18536	16935	19999
DB-Like Conjunctive Query	3	5	5	5	15	272	160	440
DB-Like Conjunctive Query	4	5	5	5	15	1467	1352	1652
DB-Like Conjunctive Query	5	5	5	5	15	18536	16905	20019

Removing Rules	D	NS	NI	P	Change	Average	Minimum	Maximum
Taxonomy	3	5	5	0	10	1386	1292	1592
Taxonomy	4	5	5	0	10	7581	7291	8352
Taxonomy	5	5	5	0	10	494726	227747	717452
Taxonomy	3	5	5	0	15	1452	1292	1772
Taxonomy	4	5	5	0	15	7615	7330	8372
Taxonomy	5	5	5	0	15	273874	189933	386005
DL-Like Ontology	3	5	5	5	10	2979	2864	3195
DL-Like Ontology	4	5	5	5	10	52613	47128	65214
DL-Like Ontology	5	5	5	5	10	OoM	OoM	OoM
DL-Like Ontology	3	5	5	5	15	33128	3055	3555
DL-Like Ontology	4	5	5	5	15	61979	50944	66395
DL-Like Ontology	5	5	5	5	15	OoM	OoM	OoM
DB-Like Conjunctive Query	3	5	5	5	10	1492	1382	1722
DB-Like Conjunctive Query	4	5	5	5	10	8011	7281	8732
DB-Like Conjunctive Query	5	5	5	5	10	517994	284389	723009
DB-Like Conjunctive Query	3	5	5	5	15	1557	1422	1783
DB-Like Conjunctive Query	4	5	5	5	15	8112	7822	8723
DB-Like Conjunctive Query	5	5	5	5	15	507760	132901	709009

Removing Facts	D	NS	NI	P	Change	Average	Minimum	Maximum
Taxonomy	3	5	5	0	10	1302	1282	1332
Taxonomy	4	5	5	0	10	7328	7281	7361
Taxonomy	5	5	5	0	10	631956	487551	759261
Taxonomy	3	5	5	0	15	1301	1291	1312
Taxonomy	4	5	5	0	15	7350	7340	7371
Taxonomy	5	5	5	0	15	542294	381628	650265
DL-Like Ontology	3	5	5	5	10	3071	2974	3184
DL-Like Ontology	4	5	5	5	10	56754	56371	57002
DL-Like Ontology	5	5	5	5	10	OoM	OoM	OoM
DL-Like Ontology	3	5	5	5	15	3242	3125	3355
DL-Like Ontology	4	5	5	5	15	58339	58104	58655
DL-Like Ontology	5	5	5	5	15	OoM	OoM	OoM
DB-Like Conjunctive Query	3	5	5	5	10	1402	1392	1412
DB-Like Conjunctive Query	4	5	5	5	10	7991	7952	8022
DB-Like Conjunctive Query	5	5	5	5	10	537931	299260	787843
DB-Like Conjunctive Query	3	5	5	5	15	1409	1382	1422
DB-Like Conjunctive Query	4	5	5	5	15	7876	7841	7901
DB-Like Conjunctive Query	5	5	5	5	15	424565	292671	482925

Adding Rules	D	NS	NI	P	Change	Average	Minimum	Maximum
Taxonomy	3	5	5	0	10	1317	1252	1463
Taxonomy	4	5	5	0	10	7265	7240	7290
Taxonomy	5	5	5	0	10	559407	393666	706696
Taxonomy	3	5	5	0	15	1286	1251	1332
Taxonomy	4	5	5	0	15	7308	7291	7331
Taxonomy	5	5	5	0	15	464588	247826	611980
DL-Like Ontology	3	5	5	5	10	3009	2834	3345
DL-Like Ontology	4	5	5	5	10	51864	47047	65444
DL-Like Ontology	5	5	5	5	10	OoM	OoM	OoM
DL-Like Ontology	3	5	5	5	15	3307	2884	3565
DL-Like Ontology	4	5	5	5	15	61283	47528	67037
DL-Like Ontology	5	5	5	5	15	OoM	OoM	OoM
DB-Like Conjunctive Query	3	5	5	5	10	1469	1392	1662
DB-Like Conjunctive Query	4	5	5	5	10	8051	7801	8523
DB-Like Conjunctive Query	5	5	5	5	10	400638	150226	541619
DB-Like Conjunctive Query	3	5	5	5	15	1462	1422	1552
DB-Like Conjunctive Query	4	5	5	5	15	7936	7902	7981
DB-Like Conjunctive Query	5	5	5	5	15	484394	141163	691164

Adding Facts	D	NS	NI	P	Change	Average	Minimum	Maximum
Taxonomy	3	5	5	0	10	1284	1262	1312
Taxonomy	4	5	5	0	10	7310	7270	7380
Taxonomy	5	5	5	0	10	649123	522761	781173
Taxonomy	3	5	5	0	15	1367	1282	1612
Taxonomy	4	5	5	0	15	7310	7271	7350
Taxonomy	5	5	5	0	15	648495	576319	756978
DL-Like Ontology	3	5	5	5	10	3620	3565	3685
DL-Like Ontology	4	5	5	5	10	136128	134463	137928
DL-Like Ontology	5	5	5	5	10	OoM	OoM	OoM
DL-Like Ontology	3	5	5	5	15	3790	3725	3895
DL-Like Ontology	4	5	5	5	15	90277	89940	90910
DL-Like Ontology	5	5	5	5	15	OoM	OoM	OoM
DB-Like Conjunctive Query	3	5	5	5	10	1399	1392	1402
DB-Like Conjunctive Query	4	5	5	5	10	7931	7761	8012
DB-Like Conjunctive Query	5	5	5	5	10	714216	460882	878814
DB-Like Conjunctive Query	3	5	5	5	15	1434	1412	1452
DB-Like Conjunctive Query	4	5	5	5	15	8011	7891	8262
DB-Like Conjunctive Query	5	5	5	5	15	763873	482964	955724

Query on Materialization	D	NS	NI	P	Change	
Taxonomy	3	5	5	0	10	0
Taxonomy	4	5	5	0	10	0
Taxonomy	5	5	5	0	10	1331
Taxonomy	3	5	5	0	15	0
Taxonomy	4	5	5	0	15	0
Taxonomy	5	5	5	0	15	1252
DL-Like Ontology	3	5	5	5	10	10
DL-Like Ontology	4	5	5	5	10	0
DL-Like Ontology	5	5	5	5	10	OoM
DL-Like Ontology	3	5	5	5	15	0
DL-Like Ontology	4	5	5	5	15	0
DL-Like Ontology	5	5	5	5	15	OoM
DB-Like Conjunctive Query	3	5	5	5	10	0
DB-Like Conjunctive Query	4	5	5	5	10	0
DB-Like Conjunctive Query	5	5	5	5	10	1633
DB-Like Conjunctive Query	3	5	5	5	15	0
DB-Like Conjunctive Query	4	5	5	5	15	0
DB-Like Conjunctive Query	5	5	5	5	15	1282