Updating OWL2 ontologies using pruned rulesets

Sana Al-Azwari John N. Wilson
Dept. of Computer & Information Sciences, University of Strathclyde, Glasgow UK.
Sana.AlAzwari@strath.ac.uk, John.N.Wilson@strath.ac.uk

ABSTRACT

Evolution in Semantic Web content produces difference files (deltas) that track changes between RDF versions. These changes may represent ontology modifications and be expressed in OWL. The deltas can be used to reduce the storage and bandwidth overhead involved in disseminating ontology updates. Minimising the delta size can be achieved by reasoning over the underlying knowledge base. OWL 2 is a development of the OWL 1 standard that incorporates new features to aid application development. Among the sub languages of OWL 2, OWL 2 RL/RDF provides an enriched rule set that extends the semantic capability of the OWL environment. This additional semantic content can be exploited in change detection approaches that strive to minimise the alterations to be made when ontologies are updated. The presence of blank nodes (i.e. nodes that are neither a URI nor a literal) in RDF collections provides a further challenge to ontology change detection because of the practical problems they introduce when comparing data structures before and after update. In the light of OWL 2 RL/RDF, this paper examines the potential for reducing the delta size by pruning the application of unnecessary rules from the reasoning process and using an approach to delta generation that produces the smallest number of updates. It also assesses the impact of alternative approaches to handling blank nodes during the change detection process in ontology structures. The results indicate that pruning the rule set is a potentially expensive process but has the benefit of reducing the joins when carrying out the subsequent inferencing.

Categories and Subject Descriptors

I.2.4 [**Semantic networks**]: Miscellaneous

General Terms

Performance

Keywords

Ontology updates, OWL 2 RL-RDF, rule pruning

1 Introduction

The continuing growth of data repositories, both in size and quantity, in association with expanding resources available over the World Wide Web is leading to major challenges in the area of data science. These must be addressed so as to provide a way of extracting the knowledge content from such sources and consequently the potential benefits that they offer. Semantic Web technologies provide a framework for integrating these data collections but there are still many problems to be overcome. A further contribution to these difficulties is the increasingly distributed nature of data in a rapidly changing world. In this context, repositories that contain useful data content typically need to be updated and these updates need to be propagated to data replicas stored in a variety of locations. This propagation may need to be carried out over data connections that are relatively slow and unreliable given the size of the data sets to be maintained. The practice of generating the difference (delta) between successive versions of an ontology helps to reduce data transfer. The delta can be copied to a remote site and used there to produce a local version of the updated ontology.

OWL 2 provides a rich ontology language that describes data structures and the way that data elements (triples) are related to one another. The OWL 2 RL/RDF variant also offers a rule set that presents opportunities for reasoning over these data collections. Given the size of such collections, for performance purposes it is necessary to consider the most effective way of selecting and applying these rules to generate deltas that can then be distributed and applied.

This paper assesses the impact of pruning rules in the OWL 2 ruleset in the context of their use in reducing the size of updates needed to transform ontologies between versions. It addresses the problem of generating a minimal update set that could be propagated to remote ontology replicas in order to maintain their consistency. As well as mimimising this set for the purposes of conserving Internet bandwidth, it is also important to generate the set with minimum expenditure of resources given that a remote site may be blocked until the update arrives.

Blank nodes provide a widely used method of representing n-ary facts in ontologies. They present a challenge in the context of comparing ontology versions since it is difficult to be certain of their eqivalence. The work reported here describes an approach to handling blank nodes in OWL 2 ontologies in the context of updating these structures. The paper reviews similar ontology update methods and describes the technical elements that contribute to the process. It then describes pruning practice as it is applied to rules and assesses the performance issues that surround this process.

2 Related work

Initial approaches to ontology change detection were based around the pre-existing Unix *diff* tool. OntoView [8] extends this by supporting the comparison of two versions of an ontology at the structural level, highlighting changes in the definitions of ontological concepts and properties. The system distinguishes between *rdf:lable* or comment changes, class or property changes and identifier changes. OntoView splits an ontology into separate definitions, which are then parsed into a group of RDF triples. Each group of triples represents a definition of a concept or a property. The algorithm then locates each group of triples in the new version and establishes a match with the corresponding group in the previous version of the ontology. The changes between these groups are then calculated. It relies on the materialization of all the *rdf:type* triples in the ontology. Blank nodes are characterised as identifier changes. Blank node matching is used indirectly with node location in the file, providing a heuristic to determine whether a particular blank node matches one in the modified ontology.

The use of heuristics in the matching process is also incorporated into PromptDiff [13] where matchers are combined with alignment to produce the structural changes between versions. The process of matching structures in PromptDiff can also be applied to processing blank nodes but no special treatment is applied to this aspect of the problem.

Also based on the *diff* approach, SemVersion [14] incorporates elements of CVS text versioning [3]. It provides for blank node enrichment, which adds properties to blank nodes in order to provide for their treatment in a manner similar to normal nodes.

x-RDF-3X [12] uses extensive indexing of triples incorporating various permutations of the triple itself as well as supporting indexing on binary projections. The indexing eliminates the problem of self-joins in table-based triple stores. Version control is maintained by timestamping triples. Updates are handled by lazy evaluation with inserts being deferred until batch incorporation becomes unavoidable, at which point, indexes are regenerated.

The RDF comparison tools reviewed in this section typically focus on high-level changes between RDF graphs. They provide for presentation of these differences in a way that is effective for supporting human interpretation such as highlighting differences with different colours [8] or representing the differences in human language rather than a language that is interpreted by machines [13]. By contrast, the approach described in this paper focuses particularly on minimising the delta between OWL 2 ontology versions and the contribution of rule pruning and blank node matching to this process.

3 Ontology change detection techniques

Following established approaches to detecting the differences between RDFS ontology versions [15], explicit differences are expressed as:

DEFINITION 1 (EXPLICIT *delta*). *Given two RDF models M and M', let* t *denote a triple in these models, Del denote triple deletion which is calculated by $M - M'$, and Ins denote triple insertion which is calculated by $M' - M$. The explicit* delta *is defined as:*

$$\Delta E = \{Del(t)|t \in M - M'\} \cup \{Ins(t)|t \in M' - M\}$$

Deltas over non-closed knowledge bases can be restricted by inference over the updates using RDFS entailment rules. This produces the explicit dense (ED) and dense (D) delta.

DEFINITION 2 (EXPLICIT DENSE *delta*). *Let M, M', Del(t), Ins(t) be as stated in Definition 1. Additionally let $C(M')$ denote the closure of M'. ΔED is defined as:*

$$\Delta ED = \{Del(t)|t \in M - C(M')\} \cup \{Ins(t)|t \in M' - M\}$$

DEFINITION 3 (DENSE *delta*). *Let M, M', Del(t), Ins(t) be as stated in Definition 1. The dense* delta *is defined as:*

$$\Delta D = \{Del(t)|t \in M - C(M')\} \cup \{Ins(t)|t \in M' - C(M)\}$$

The dense delta is non-deterministic as a result of inter-effects between insertion and deletion. This problem is overcome by the corrected dense delta (D_c) [1].

DEFINITION 4 (CORRECTED DENSE *delta*). *Let ΔE, $C(M)$ and $C(M')$ be as defined previously and additionally let $s \rightarrow t$ indicate that s is an antecedent of t. The corrected dense delta ΔD_c is defined as*

$$\Delta D_c = \Delta E - \{\{Del(t)|t \in C(M')\} \cup \{Ins(t)|t \in C(M) \wedge \{s \rightarrow t|s \notin Del(t)\}\}\}$$

These definitions provide a basis for characterising the differences between successive ontology versions. Exploiting the inference that is implied in the RDFS entailment rules permits the corrected dense delta to represent the most compact deterministic way of transforming one version of an ontology into an updated version.

3.1 OWL 2 RL/RDF rules

The RDFS rule set provides limited scope for entailment and most of the rules are not relevant to inference over RDF updates. The OWL 2 RL/RDF rule set is more extensive, comprising of 23 rules that provide considerable scope for reasoning over ontology updates[11]. Examples of the rules are shown in Table 1.

OWL 2 rules form an OR tree and as can be seen from Table 1, there are multiple possibilities for establishing a single consequence such as $\{x$ *rdf:type* $y\}$. Furthermore, the structure of these rules allows for iterative inference over a triple set. That is, each rule may produce new triples that can be added to the triple set and impact further rounds of rule application.

The application of these rules to ontology deltas can be used to find the smallest delta that can unambiguously represent the difference between two ontologies. A significant challenge in reasoning over such differences comes from the presence of blank nodes in ontologies.

3.2 Rule execution

Simple implementations of OWL 2 RL rules perform poorly in ontologies with large ABoxes [6]. However, optimization such as the parallelisation of backward inference can improve the performance of rule implementations.

This work focuses on backward-chaining for the reduction of RDF deltas.

DEFINITION 5 (DELTA REDUCTION USING BACKWARD CHAINING).
Let M, M' be as stated in Definition 1. The reduced delta δ_R is defined as: a reduced set of triples $t_I \mid t_I \notin \delta_R are entailed in M_{1,2}$ using the rules in R.

Regardless of the set of considered rules, for each update (i.e. triple) in the delta, backward-chaining first searches all the rules for a conclusion that is compatible with this update. After this, it will look at

Abb	Antecedent	Consequent
scm-eqc2	$\{x\ rdfs{:}sunClassOf\ y\}\{y\ rdfs{:}sunClassOf\ x\}$	$\{x\ owl{:}equivalentClass\ y\}$
cls-svf1	$\{x\ owl{:}someValuesFrom\ y\}\{x\ owl{:}onProperty\ p\}\{u\ p\ v\}\{v\ rdf{:}type\ y\}$	$\{u\ rdf{:}type\ x\}$
cls-hv2	$\{x\ owl{:}hasValue\ y\}\{x\ owl{:}onProperty\ p\}\{u\ p\ y\}$	$\{u\ rdf{:}type\ x\}$

Table 1: Entailment rules in OWL2 RF/RDF

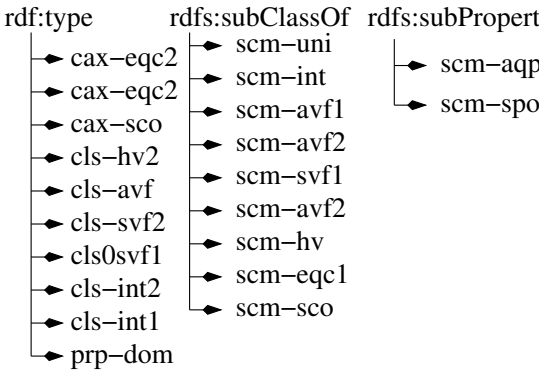

Figure 1: OWL 2 OR trees used to derive conclusions about rdfs:subPropertyOf, rdfs:subClassOf and rdf:type

the body of these rules trying to find antecedent patterns that contain variables in the same position as specified in the body of the rule. Only triples that contain properties of the type: *rdfs:subClassOf*, *rdfs:subPropertyOf* or *rdf:type* are inferable and are checked in this way. A subset of the OWL 2 RL/RDF rules can be categorised into three groups based on these properties. Each group contains a set of rules that have a property of these values as a conclusion and a body consisting of one or more antecedent patterns that lead to that conclusion. Figure 1 shows the resulting OR tree. To check if an update of a particular property type is inferable in the knowledge base, the set of rules in the appropriate or-tree are applied sequentially until the update is inferred in the knowledge base or no more rules remain to apply.
Implementation of these rules can be simplified by decomposing the antecedents into multiple database searches which are terminated when one component fails to return a value.Further simplification can be achieved by executing rule patterns in a specific order starting with the least specific. The antecedents of rule patterns in OWL 2 RL are either selective, non-selective or recursive. A selective pattern does not require further execution of the set of rules in order to entail the desired conclusion. If no triples in the knowledge base match the selective pattern then no further rules can be applied to infer that pattern. This contrasts with the recursive pattern which will generate repeated calls until the desired conclusion is found or until no more patterns can be executed.

EXAMPLE 1. *The rule cls-svf1 has antecedents*
$(?x$ *owl:someValuesFrom* $?y)$
$(?x$ *owl:onProperty* $?p)$
$(?u\ ?p\ ?v)$
$(?v$ *rdf:type* $?y)$
and consesquent $(?u$ *rdf:type*
$?x)$ *One step in reaching the consequent is to establish a list of triples that match the selective triple pattern* $(?x$ *owl:someValuesFrom* $?y)$ *which will bind only to triples containing owl:someValuesFrom as a predicate.*

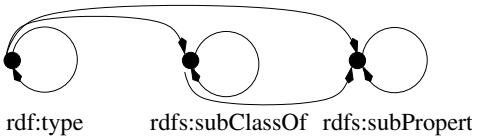

Figure 2: Overlapped OR trees. The round arrows indicate a recursive call from within the OR tree

EXAMPLE 2. *A further step in reaching the consequent of cls-svf1 is to bind triples that match the non-selective pattern* $(?u, ?p, ?v)$.

EXAMPLE 3. *Rule cls-svf1 also requires the recursive antecedent* $(?v,$ *rdf:type*, $?y)$. *This antecedent can be established by consulting any of the rules in the rdf:type or tree shown in Figure 6 which includes a call to cls-svf1.*

Non-selective rule antecedents may trigger the execution of further rules but are not in themselves recursive. For each triple in the delta, the purpose of executing the rules is to see if the triple can be inferred in the updated set (M'). At this point the OR tree for that triple can be terminated. In order to achieve this, the order of executing these patterns starts with selective patterns, followed by the non-selective patterns and finally the recursive pattern as they are the most complex in terms of execution.
Recursive patterns are potentially expensive in terms of their execution because they generate further calls until the desired conclusion is found or until no more patterns can be executed. In contrast, the execution of the selective pattern is relatively simple, because a conclusion such as $(?u$ *rdf:type* $?x)$ can be derived from the knowledge base simply by finding that the object of the triple $(?x)$ exists in the someValuesFrom table. In the case where this object does not exist, the rest of the patterns in the rule no longer require further execution. Thus, avoiding the execution of recursive pattern if exists. Thus, in this example, the patterns $(?x$ *owl:someValuesFrom* $?y)$ and $(?x$ *owl:onProperty* $?p)$ are executed first because they are both selective patterns and can save the execution of the other patterns if no triple in the knowledge base matches one of these patterns. Subsequently the pattern $(?u\ ?p\ ?v)$ and finally the pattern $(?v$ *rdf:type* $?y)$ are matched because they may require further execution of rules if no triples in the knowledge base match the pattern. Decomposing the execution of these patterns in this way may avoid the searches required in executing them as a single query and consequently reduce the execution time [9]. Decomposed sections can then be executed separately following the order described above.
As a result of the recursive patterns, the different OR trees overlap because recursive patterns in one OR tree may require further application of other rules which may be in other OR trees. Figure 2 shows the overlapped OR trees as concluded from the rules they contain.

3.3 Blank nodes

Blank nodes, are a special kind of nodes without a name. They indicate the existence of a thing for which a URI reference or literal

value is not given. Since they are anonymous, blank nodes require special treatment when matching ontologies. Despite the problems involved in processing data with these anonymous nodes, the use of blank nodes in RDF data models is an important feature, which adds flexibility when expressing information in RDF model.
The first stage in delta construction is the computation and production of the explicit delta (i.e. the syntactical differences) between the two stored models. After the computation of the syntactical differences, the blank node matching begins, although no order is required for the two processes as they do not overlap. Blank nodes are arranged in chains and the matching of these nodes can make use of both the ID of the node as well as the triple count in its chain. The equivalence of RDF graphs that contain blank nodes is defined as [2] :

DEFINITION 6 (EQUIVALENCE OF RDF GRAPHS WITH BLANK NODES).
Two generalized RDF graphs G_1 and G_2 are equivalent if there is a bijection f between the sets of triples of the two graphs, such that:
f(uri) = uri for all uri $\in U_1 \subseteq G_1$
f(lit) = lit for each lit $\in L_1 \subseteq G_1$
For each $b \in B_1$ f maps blank nodes to blank nodes, such that f (b) $\in B_2$
The triple (s, p, o) is in G_1 if and only if the triple (f(s), p, f(o)) is in G_2
It follows that if two graphs are equivalent then it certainly holds $U_1 = U_2$, $L_1 = L_2$ and $\|B_1\| = \|B_2\|$. Thus, f shows how each blank node identifier in G_1 can be replaced by a new identifier in order to give G_2

Without blank node matching, any pair of blank nodes from different knowledge bases is considered as a difference between these data structures. If $|T_{b1}|$ and $|T_{b2}|$ are the blank node counts in M and M' respectively then without blank node matching the delta for two graphs will contain at least $|T_{b1}| + |T_{b2}|$ change operations. Matching these blank nodes may reduce the size of the delta. The worst case of blank node matching is when all blank nodes in the participating triples are not matched. In this case, the delta size with blank node matching is equal to the delta size without blank node matching. Thus, if blank node matching does not reduce the delta size, it will not increase it.

4 Delta generation using pruned rulesets

The use of pruning rules in the context of RDFS knowledge bases typically follows the process of checking the subject and object of each triple to see if it exists in the knowledge base. If it does exist then it is needed for the inferencing process. If not, the triple can be pruned from the inferencing set. This works well when there is a large number of triples and few rules - as is the case with the RDFS entailment rule set a single round of rule application is provided. Where the rules are more complex, as in the case of OWL2 RL/RDF, pruning the rule set rather than the triples becomes more important. The contribution of the work described here is the process of pruning OWL 2 rulesets in the context of repeated rounds of rule application. This contrasts with previous approaches to the problem that focused on pruning the triples themselves.

4.1 Pruning OR trees

The process of pruning OR trees starts with the generation of ΔE. Each triple in the delta set is checked against the dataset to determine whether it is inferable, which would allow it to be removed from the delta set and hence reduce the delta size. This process requires the execution of each rule in the OR tree for the corresponding triple (i.e. *rdf:type*, *rdfs:subclassOf*, or *rdfs:subPropertyOf*). In a relational datastore implementaton, the execution of these rules involves joins between two table tables in the database that match the patterns in these rules. However, some execution of these rules, and therefore joins between tables in the database, are unnecessary and can be avoided as they will not lead to the desired conclusion.
In the example shown in Figure 3, M and M' are two different versions of an OWL knowledge base with M' being a newer version of M. The explicit differences (ΔE) between the two versions are shown in the same figure. This example focuses only on the deletion set of triples because the process of reducing this set does not require further checking to perform correct and valid reduction of the delta, as would be the case if the insertion set was involved. Reducing the deletion set requires the application of OWL inference rules against M', the newer version of the dataset. The deletion set in the delta contains a triple ($MathTeacher$ *rdfs:subClassOf Staff*).
In order to reduce the delta size, the triple needs to be checked to see if it is inferable in M'. This involves executing the rules in the OR tree for the *subClassOf* property shown in Figure 1 until this triple is inferred by the execution of one of these rules or until no more rules can be applied. In the former case, the triple is removed from the delta. In the latter case the triple should remain in the delta. The other rules used in this process are also identified in Figure 1. Using as an example the recursive rule scm-sco, the execution of this rule requires a recursive call to the rule until the triple is inferred or no more recursive calls can be applied.
The evaluation of the pruning algorithm (Algorithm 1) described in the work is based on a relational triple store as explained in Section 5. Each time a recursive call is made, a self-join to the $subClassOf$ table is required in order to infer the triple
($MathTeacher$ *rdfs:subClassOf Staff*). Initially, a search is carried out to find if the patterns ($MathTeacher$ *rdfs:subClassOf* $?c$) and ($?c$ *rdfs:subClassOf Staff*) exist. If they can be found, the triple is inferable and can safely be removed from the delta. However, if triples matching these patterns do not exist then the straightforward approach is to find all the patterns that have $MathTeacher$ in the subject position and apply a recursive call to this rule until the main triple (i.e.
($MathTeacher$ *rdfs:subClassOf Staff*)) is inferred or no more patterns can be generated from the dataset.
If triples such as ($MathTeacher$ *rdfs:subClassOf* $C1$),
($MathTeacher$ *rdfs:subClassOf* $C2$) etc. exist in M' then these triples are added to a list of those in M' that have $MathTeacher$ in the subject position. In the context of the scm-sco rule and consideration of the triple ($MathTeacher$ *rdfs:subClassOf* $C1$), a recursive call is made to the rule in order to infer ($C1$ *rdfs:subClassOf* y) by searching for the patterns ($C1$ *rdfs:subClassOf* $?x$) and
($?c$ *rdfs:subClassOf* y)
If these patterns do not exist in M', then all the patterns that have $C1$ in the subject position are generated and this process continues until the triple is inferred or no further patterns are generated.
This approach requires successive self-joins in the triple store despite which it may not be possible to infer the triple in order to reduce the delta size. There is potential advantage in avoiding unnecessary rule execution since this will result in potentially multiple self-joins in the triple store. This paper describes a method of pruning unnecessary rules in the OR tree. The approach is based on initially checking whether both the subject and object of a triple exist in the appropriate positions as defined by the patterns in each rule before executing that rule. If both subject and object exist then the rule is applied otherwise it is pruned from execution. The

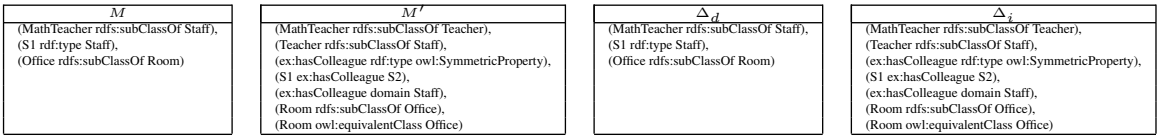

Figure 3: Sample data structure before and after update togther with the insert and delete sets

checking avoids the use of joins in the triple store, thereby reducing the effort involved in further processing.

Generally, to infer the triple $(x$ *rdfs:subClassOf* $y)$, both x and y are checked to see if they exist in the $subClassOf$ table in the subject position of one triple and the object position of another triple respectively. No joins are needed in this step. If the method returns true then the rule can be executed, otherwise this rule is pruned and no further checking of the consequent takes place. If the rule is pruned, the other rules in the OR tree are checked in the same way until a true value is applied or no more rules remain in the OR tree.

In the case of a triple such as $(S1$ *rdf:type* $Staff)$ in the deletion set, reduction in the delta size and particularly the deletion set can be achieved by checking whether the triple is inferable in M' or not by applying rules in the *rdf:type* OR tree. Before proceeding with the execution of these rules, the subject $(S1)$ and the object (*Staff*) of this triple are checked against all patterns within the rules of the *rdf:type* OR tree. The rule prp-dom, for instance, has two patterns $(?p$ *rdfs:domain* $?c)$ and $(?x\ ?p\ ?y)$ in its body which infer the conclusion $(?x$ *rdf:type* $?c)$. To check if this rule can be pruned, we need to check if the subject of the triple $(S1$ *rdf:type* $Staff)$ exists in either the subject column or the object column of the general triple table. Furthermore, it is necessary to check the object column in the rdfs:domain table to ascertain whether value *Staff* exists as the object of that triple.

Checking the existence of the subject $S1$ in either position of the triple table is an exceptional case that appears in all rules containing a non-terminological pattern (i.e. the property of the triple is a user-defined property) such as $(?x\ ?p\ ?y)$. The reason for checking the existence of the value in either the subject or the object columns of the general triple table is because triples matching non-terminological patterns can be inferred by other rules which include non-terminological patterns in their bodies that reverse the positions of the values of the subject and object. An example of such a rule is prp-symp, which has two antecedents in its body: $(?p$ *rdf:type owl:SymmetricProperty*$)$ and $(?y\ ?p\ ?x)$, and derives a conclusion $(?x\ ?p\ ?y)$. Checking the value $S1$ of the triple in only the subject column of the triple table is not enough to decide if the rule can be pruned as triples matching the non-terminological pattern $(?x\ ?p\ ?y)$ can be concluded by other rules having non-terminological patterns with the value of the subject in the object position $(?y\ ?p\ ?x)$.

According to the rule prp-dom, checking the subject of the triple $(S1$ *rdf:type Staff*$)$ in only the subject column of the general triple table will result in pruning this rule as $S1$ does not exist in the subject column in this table as shown in Figure 3. However, M' contains the triples $(ex:hasColleague\ rdf:type\ owl:SymmetricProperty)$ and
$(S2$ *ex:hasColleague* $S1)$ which according to rule prp-symp can produce as a conclusion the triple $(S1 ex:hasColleague S2)$. This has $S1$ in the subject position which is necessary for the execution of the rule prp-dom.

To summarise, pruning a rule involves checking whether the value of the subject and the object of the corresponding triples in the delta set exist in the same position as stated in the patterns of the body of that particular rule. Only when a rule contains a non-terminological pattern is the existence of a particular value is checked against either the subject position or the object position.

Algorithm 1: Reasoning with pruned rules

Data: $t \in \Delta$, orTree
Result: true if the update is inferable in the knowledge base otherwise false

```
1  rules = orTree.getRules(t)//get the rules from the corresponding orTree
2  result = false
3  while rule in rules OR result = false do
4      selectivePatterns = rule.getSelectivePatterns()
5      for selectivePattern ∈ selectivePatterns do
6          if not{selectivePattern.FindMatchInFixedPositions(update)} then
7              rule.prune()
8      nonSelectivePatterns = rule.getNonSelectivePatterns()
9      for nonSelectivePattern ∈ nonSelectivePatterns do
10         if not{nonSelectivPattern.FindMatchAnyPosition(update)} then
11             rule.prune()
12     recursivePatterns = rule.getRecursivePatterns()
13     for recursivePattern ∈ recursivePatterns do
14         if not{recursivePattern.FindMatchAnyPosition(update)} then
15             rule.prune()
16     result = rule.apply()
17 return result
```

4.2 Blank node pre-processing

RDF model theory [5] characterises blank nodes as having local scope within the file that contains them. Such nodes act as existential quantifiers over a set of resources in which the identifiers of the blank nodes are not significant. In practice blank nodes are used to describe multi-component structures represented by RDF containers, to describe reification (i.e. triples about triples) or to represent complex information. Given the local scope of blank nodes, it is not possible to rely on their identifiers being consistent between successive ontology versions. However, chains of blank node hold information that may be useful in the process of reasoning about updates between ontology versions. Loading blank nodes involves pre-processing these nodes to trace graphs of triples that that contain them. This step is useful for matching blank nodes when computing the differences between two versions of an ontology. Once the chain of triples has been traced, it is possilbe to extract information from it and exploit this to de-anonymise the blanks. This allows the blank node structure to be considered as part of the reasoning process.

Tracing blank nodes is based on the assumption that such chains

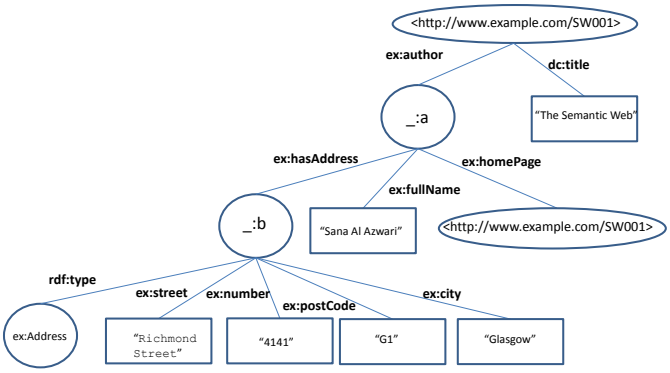

Figure 4: Blank nodes tree structures

```
<http://www.example.com/SW001> dc:title "The Semantic Web"
<http://www.example.com/SW001> ex:author _:a .
_:a ex:fullName "Sana Al Azwari" .
_:a ex:homePage <http://www.strath.ac.uk/~alazwari/> .
_:a ex:hasAddress _:b .
_:b rdf:type ex:Address .
_:b ex:street "Richmond Street" .
_:b ex:number "4141" .
_:b ex:postalcode "G1" .
_:b ex:city "Glasgow" .
```

Figure 5: Blank nodes chain example

start with a non-blank node (i.e. a URI) in the subject position of a triple. These blank node chains may form a tree such as that shown in Figure 4 that represents the N-triple shown in Figure 5. If a triple with a non-blank node in the subject position and a blank node in the object positon is encountered, the tracing process begins tracing all connected blank nodes until no more related triples are found. The length of the chain is equal to the number of triples containing the connected blank node. The example in 5 gives a chain length of 9. Each chain of blank nodes is held in the triple store along with the length of the chain in order to use it in the matching process. Effectively, each chain now has an ID to distinguish the group of triples that belongs to it.

5 Results and discussion

The process of pruning rules used in ontology updates with OWL 2 RF/RDF rules has been evaluated experimentally using the Lehigh University Benchmark (LUBM) [4] and the University Ontology Benchmark (UOBM) [10]. These Semantic Web benchmarks allow the generation of datasets of different sizes. LUBM facilitates the evaluation of Semantic Web tools and is accepted as a standard evaluation platform for OWL ontology systems. Despite this, it does not fully support the inference of either OWL lite or OWL DL profiles of OWL 2. For example, inferencing the *allValuesFrom* restrictions and the cardinality constraints cannot be tested using LUBM datasets. Furthermore, the generated instance data lacks inter-linkage between isolated subgraphs. In this context, instance data can be generated to represent individuals for a number of universities but individuals in one university do not have relations with individuals from other universities. This limits the benchmark's value for scalability tests as inference on connected subgraphs is harder than that on isolated subgraphs. As a consequence of this, LUBM is weaker in measuring the capability of inference engines as it does not trigger all the inference rules supported by these engines.

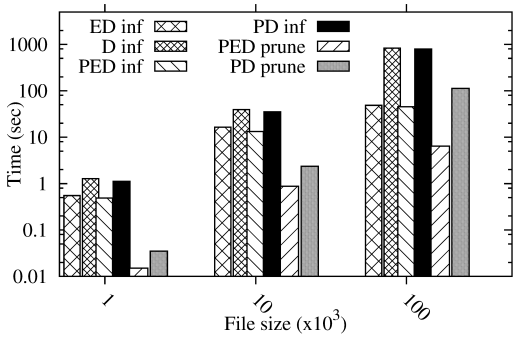

Figure 6: Reasonong time for 10% updates LUBM data set with no blank node support

For these reasons, UOBM was developed to extend LUBM and overcome its limitations with full support for both OWL lite and OWL DL as well as the generation of a more complex instance datasets by establishing links between individuals from different universities.

In the experimental work reported here both LUBM and UOBM benchmark generators were used to produce three versions nominally of 1000, 10,000 and 100,000 triples respectively. Three change ratios on each of these different sizes of datasets were produced. This involved changing the subsumption hierarchy as well as the addition of inferable triples. These inferable triples were obtained by materializing ontology versions and selecting a number of the inferred triples to be added to the corresponding dataset. Using this manipulation, four versions for each size of the datasets were generated: the original version; %5 change ratio version; %10 change ratio version and %15 change ratio version. Table 2 represents the feature of the different versions generated using both LUBM and UOBM benchmarks.

The triple store was implemented in MySQL to handle the RDF collections and the *deltas*. Each predicate was represented in a separate table. Indexing was excluded to preserve the validity of the use-case. The triple store was loaded and updates were validated using the Jena framework. All experiment were performed on Intel Xeon CPU X3470 @ 2.93GHz - 1 cpu with 4 cores and hyperthreading, Ubuntu 12.04 LTS operating system and 16GB memory. Computation of the syntactic differences between successive ontology versions starts with the generation of δE. This step takes into account non-blank node triples (i.e. triples that do not contain blank identifiers in any position). After the calculation of the explicit difference between the two versions and the blank node matching, these differences enter a reduction phase where reasoning under the semantics of OWL 2 RL/RDF is employed for the purpose of minimizing unnecessary change operations (i.e. insertions or deletions). In addition to the differential functions explained in Section 3, two pruning-based functions as proposed in [7] are also employed. These functions combine the differential functions in [15] with pruning methods to reduce unnecessary computation during the reasoning process.

Updates were calculated for each of the sample datasets and indicate that the inference load for ΔD_c exceeded that for ΔED in consequence of the latter approach only carrying out inference over the delete set (Figure 6). Similarly, the process of pruning rules in the ΔD_c approach is more costly than pruning rules for ΔED because the former, being a larger set, presents more pruning opportunities. The distinction between the UOBM and LUBM bench-

	LUBM				UOBM			
Nominal size	**Original size**	**%5**	**%10**	**%15**	**Original size**	**%5**	**%10**	**%15**
1000	**1391**	**1380**	**1418**	**1445**	**965**	**970**	**962**	**967**
10000	**10149**	**10348**	**10553**	**10853**	**10097**	**9956**	**10696**	**10729**
100000	**100448**	**102165**	**109377**	**113002**	**101133**	**101894**	**103354**	**107703**

Table 2: Triple count in LUBM and UOBM.

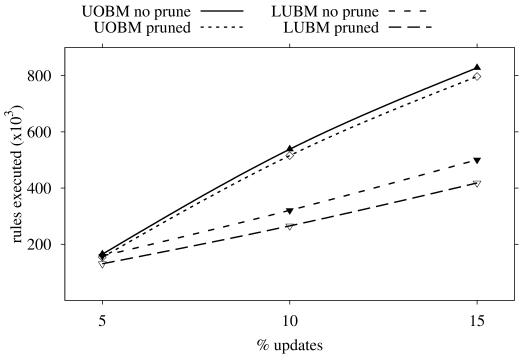

Figure 7: Reduction in rules assessed as a consequence of pruning in the 100000 triple structure

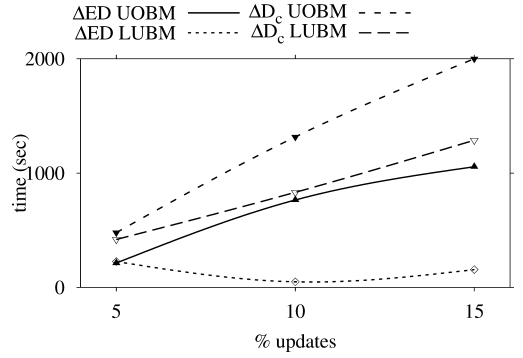

Figure 8: Inference time for ΔED and ΔD_c in the 100000 triple for both LUBM and UOBM

marks is evident from Figure 7. It can be seen here that UOBM data triggers the execution of more rules than the simpler LUBM set. Both data sets benefit from the reduction in rule operation that is supported by the pruning process described above although the benefit is more pronounced for the latter set. This indicates that the benefits produced by pruning rules is influenced by the data distribution within a particular dataset. Benchmark data can be deficient in this respect and may not reflect real world data very accurately.
The variation of inferencing in LUBM and UOBM for the 100000 triple set is shown in Figure 8. As with Figure 7 the results indicate that the UOBM set presents more of a challenge to the inference process because of its richer structure. The impact of blank node reduction is shown in Figure 9. This process saves additional triples in the delta, which has consequences for the performance time of both rule pruning and inferencing. Where blank nodes are supported, the cost of both of these tasks is reduced.
Overall the results indicate that both rule pruning and blank node matching have the potential for reducing the processing required for generating compact deltas. In the context of generating deltas

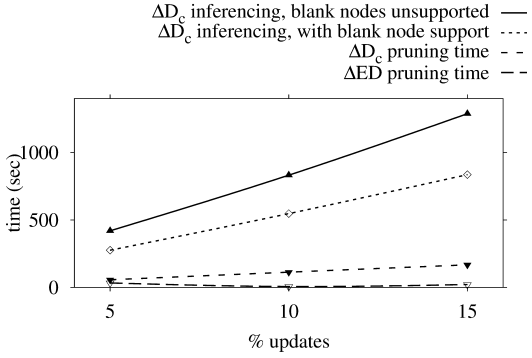

Figure 9: Performance time for 100000 triple set with and without support for blank nodes

between ontology versions, pruning rules offers an alternative approach to pruning triples [15]. It presents particular benefits when the rule set is large and arranged in an OR tree.

6 Conclusion and future work

The semantics of RDF can be exploited in order to reduce the differences between RDF versions. However the rich ruleset of ontology languages such as OWL 2 may provide a challenge to change detection techniques. In particular,the repeated application of a large ruleset that may be necessary to produce the desired conclusion can result in performance problems. Blindly applying rules will result in many such applicatione being void as a result of consequents that can not contribute to the desired outcome. Advance knowledge of which rules are applicable and which are not is very important in avoiding their unnecessary application. This paper describes a change detection technique using backward-chaining inference. It produces a small delta using a pruning method that eliminates unnecessary inference rules during the reduction of the delta size.
A further reduction in delta size is possible through blank node matching method. This method matches chains of blank nodes between ontology versions. Excluding matched blank nodes from the delta is beneficial in reducing the delta size and hence the network bandwidth when synchronizing ontology versions as well as the storage overhead for deltas.
The change detection technique described in this work is based on OR trees and is inhereently parallelisable. The opportunities for using this approach need to be addressed in future work. The rule pruning approach can also be extended to incorporate more complex rules and to investigate their effect on delta reduction. An example of these rules are those that exploit the *owl:sameAs* relation, which have been excluded from the current work due to their execution complexity.
The work presented in this paper describes a framework for delta production that starts with the process of generating a physical representation of successive ontology versions. This reprsentation is

conveniently handled by a relational data store. The process of blank node matching can then be used to avoid incorporating such content into the differences that are detected between the versions. The subsequent stage of delta generation then leads to the final step in the process, which involves delta reduction.

7 References

[1] S. M. M. Al Azwari and J. Wilson. Consistent RDF updates with correct dense deltas. In *Proc 30th BICOD*, 2015.

[2] J. J. Carroll and G. Klyne. Resource description framework (RDF): Concepts and abstract syntax. 2004.

[3] CVS - concurrent versions system. `http://www.nongnu.org/cvs/`. Accessed: 2015-06-10.

[4] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[5] P. Hayes and B. McBride. RDF semantics. W3C recommendation. *World Wide Web Consortium*, 2004.

[6] A. Hogan and S. Decker. On the ostensibly silent 'W' in OWL 2 RL. In *Web Reasoning and Rule Systems*, pages 118–134. Springer, 2009.

[7] D.-H. Im, S.-W. Lee, and H.-J. Kim. Backward inference and pruning for RDF change detection using RDBMS. *J. Info. Science*, 39(2):238–255, 2013.

[8] M. Klein. Supporting evolving ontologies on the internet. In *XML-Based Data Management and Multimedia EngineeringÛEDBT 2002 Workshops*, pages 597–606. Springer, 2002.

[9] V. Kolovski, Z. Wu, and G. Eadon. Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system. In *The Semantic Web–ISWC 2010*, pages 436–452. Springer, 2010.

[10] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu. *Towards a complete OWL ontology benchmark*. Springer, 2006.

[11] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL 2 web ontology language: Profiles. *W3C recommendation*, 27:61, 2009.

[12] T. Neumann and G. Weikum. x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *Proceedings of the VLDB Endowment*, 3(1-2):256–263, 2010.

[13] N. F. Noy, M. A. Musen, et al. Promptdiff: A fixed-point algorithm for comparing ontology versions. *AAAI/IAAI*, 2002:744–750, 2002.

[14] M. Völkel and T. Groza. SemVersion: An RDF-based ontology versioning system. In *Proceedings of the IADIS international conference WWW/Internet*, volume 2006, page 44, 2006.

[15] D. Zeginis, Y. Tzitzikas, and V. Christophides. On computing deltas of RDF/S knowledge bases. *ACM Trans on the Web (TWEB)*, 5(3):14, 2011.