# Extending LiteMat toward RDFS++

Olivier Curé[1], Weiqin Xu[1,2], Hubert Naacke[3], Philippe Calvez[2]

[1] LIGM (UMR 8049), CNRS, UPEM, F-77454, Marne-la-Vallée, France.
{firstname.lastname}@u-pem.fr
[2] ENGIE CRIGEN CSAI Lab,
philippe.calvez1@engie.com
[3] Sorbonne Universités, UPMC Univ Paris 06, F-75005, Paris, France
hubert.naacke@lip6.fr

**Abstract.** In this paper, we extend LiteMat, an encoding scheme for RDF data that currently supports inferences based on RDFS and the `owl:sameAs` property, which is used in a distributed knowledge graph data management system. Our extensions enable to reach RDFS++ expressiveness by integrating `owl:transitiveProperty` and `owl:inverseOf` properties. Considering the latter, `owl:inverseOf` property, we propose a simple solution that involves a dictionary look-up at query run-time. For the former, we present an efficient approach to encode individuals involved in chain and tree structures of a transitive property. We provide details of a distributed implementation and highlight the efficiency of our encoding and query processing approaches over large synthetic datasets.



## 1 Introduction

Large scale RDF analytics requires a reliable, distributed knowledge graph data management coupled with an efficient processing of inferences based on expressive ontologies. We consider that with today's distributed computing frameworks and cloud computing infrastructure, it is possible to achieve such a goal when a large portion of the data and knowledge reside in main-memory. In [1] and [5], we have followed this design principle by integrating a semantic-aware encoding of ontology elements, *i.e.*, concepts, properties and individuals, in an RDF store and streaming system that are both based on Apache Spark.

More precisely, LiteMat[1] is an encoding scheme for RDF data that offers a trade-off between materialization (of inferred triples) and query rewriting, in order to obtain complete result sets from queries. It uses an integer interval based encoding for the ontology elements that efficiently and effectively captures in a compressed manner the ontology entity hierarchy. In [1], we are applying this encoding to the $\rho$df[4] fragment of RDFS, *i.e.*, supporting inferences associated to `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain` and `rdfs:range` properties. More recently, *Strider*$^R$[5] extended LiteMat to support the `owl:sameAs` property which is quite popular in the Linked Data community. Intuitively, a special encoding was applied to all individuals present in `owl:sameAs` cliques and a representative was selected among them. Like LiteMat, the work presented in

[6] models the concept and property subsumption hierarchies with an intelligent integer identification that is used to rewrite SQL queries in the ontology-based data access Quest system. With the latest extension of LiteMat, we go further with a smart identification solution for individuals involved in `owl:sameAs` cliques and transitive structures taking the form of chains and trees.

The contributions presented in this paper permit to extend these two previous work toward RDFS++ expressiveness. These extensions concern both an encoding scheme that results in a more compact knowledge graph representation and an adapted query processing. We first address `owl:inverseOf` properties by applying a simple transformation of the encoded ABox and a property dictionary look-up at query processing-time. Our approach for transitive properties is more involved and is based on (i) an encoding solution of the individuals involved in chains and trees of these properties and (ii) a query processing strategy. Due to a lack of an efficient solution, directed acyclic graphs (DAG) of transitive properties, which are relatively rare in practice, are currently being materialized in the triples store.

The paper is organized as follows: in Section 2 we detail our solution toward supporting `owl:inverseOf` and `owl:transitiveProperty` . In Section 3, we present the principles of the query processing in the presence of transitive properties. Section 4 provides an evaluation on synthetic datasets over the memory footprint, encoding duration and query processing dimensions. Finally, Section 5 concludes the paper and presents some future work.

## 2 RDFS++ extensions for LiteMat

### 2.1 Support for `owl:inverseOf`

Concerning `owl:inverseOf` properties, we propose the following simple approach. For each URI property and its inverse, denoted $\langle p, p^- \rangle$, we retain in our ABox, only one of the two URIs which is therefore denoted as the property representative, *i.e.*, $p_r$. For each pair $\langle p, p^- \rangle$ in the ABox, a representative, $p_r$, is selected based on the largest number of occurrences over the pair $\langle p, p^- \rangle$.

In LiteMat's locate property dictionary, *i.e.*, URI to identifier key-value structure, both property URIs are associated to the same integer identifier: both $p_r$ and $p^-$ are associated to a unique *pid* value as computed in [1]. In the extract property dictionary, *i.e.*, id to URI key-value structure, only the representative property is stored since answers to queries requiring an extract operation on the property are expressed with $p_r$.

During the ABox encoding, all triples already expressed with $p_r$ are normally encoded using the individuals and properties dictionaries. Concerning all triples expressed with $p_r^-$, *e.g.*, $i1\ p_r^-\ i2$, they are transformed as follows: the subject and object of the original triples respectively become the object and subject of a new triple and the property is switched to the representative.

**Example:** Let $parentOf$ `owl:inverseOf` $childOf$ be a TBox axiom and $parentOf$ is selected as the representative in this property pair. Table 1 displays an original ABox and its resulting transformation.

| Original ABox | Transformed ABox |
|---|---|
| dominique parentOf jean. | dominique parentOf jean. |
| dominique parentOf pierre. | dominique parentOf pierre. |
| marie childOf pierre. | **pierre parentOf marie.** |

**Table 1:** TBox encoding facts

A similar transformation is applied to graph patterns of a SPARQL query whenever the inverse of a representative is identified in a Basic Graph Pattern (BGP). That is the query `SELECT ?x ?y WHERE {?x childOf ?y}` would be transformed into `SELECT ?y ?x WHERE {?y parentOf ?x}`. This would return an answer with 3 tuples including the pair ⟨ pierre, marie ⟩.

### 2.2 Support for `owl:transitiveProperty`

Let consider a function $trans(G, p) = G'$, with $G$ an RDF graph, $p$ a transitive property and $G'$ a subgraph of $G$ solely composed of triples with the $p$ property. Intuitively, $G'$ is composed of, not necessarily connected, chains, trees or DAGs of individuals (see Figure 1 for examples of the first two structures).

In this section, we propose two encoding schemes, one for the chain and another one for tree structures that are following the logical approach of LiteMat. That is, it provides semantic-aware identifiers to ABox individuals encountered in these structures. We leave the issue of encoding the DAG transitive structures to a future work and will consider that in the current state of the LiteMat data management system, the transitive closure of these structures is materialized.

The characteristics that we are aiming for in this encoding schemes are:

- compactness since no materialization is required for the chain and tree structures
- determinism since the identifier of each individual in a chain or tree is computed deterministically.
- scalability since all encoding tasks are performed in a distributed manner on a distributed engine, namely Apache Spark and its GraphX graph computing component..

In both the chain and tree encoding, our processing starts with the computation of $trans(G, p)$ for all transitive properties of the associated ontology, resulting in a set of subgraphs. Then, the system computes the connected components for each of these subgraphs. Intuitively, the connected components operation groups vertices into connected subgraphs. This can easily be performed in a scalable manner with Spark's GraphX component. Such a resulting connected component is assigned a distinct identifier corresponding to the lowest node identifier of the connected component. The encoding of individuals in these graphs is made of a quadruple of integer values which correspond to: $fid$ which is 0 if the transitive structure is a chain or 1 if it is a tree, $pid$ the identifier of the transitive property, $ccid$ the connected component identifier and $lid$ a local node identifier.

The chain and tree structures are distinguished by the computation of their local identifiers. Figure 1 presents in each node the label of the individuals (*i.e.*, URIs) and below them their identifier. In the case of a chain structure, it is sufficient to assign integer values that define a total order over the set of *lid* of a given triple $\langle fid, pid, ccid \rangle$. With such an approach, the computation of all descendants (respectively ancestors) of a given individuals $\langle fid, pid, ccid, lid_i \rangle$ will amount to retrieve individuals identified by $\langle fid, pid, ccid, lid_x \rangle$ for all $lid_i < lid_x$ (respectively, $lid_i > lid_x$).

The encoding of tree structures is more involved. For instance, in Figure 1(b), individuals E,F and G do not belong in the transitive closure of B or D. In this case, the incremental, naive assignment of local identifier is not sufficient to efficiently detect that a node is not in the transitive closure of another node in this graph. We adopt a local node identifier approach that is inspired by our LiteMat binary approach. Intuitively, the encoding algorithm is recursively assigning binary identifiers in a top-down manner from the root of the tree. The root node starts with a single bit set to 1. Then we identify all directly linked individuals for a node. The size of this set of individuals justifies the length of the binary encoding for each of these individuals. For instance, in Figure 1(b), A has three directly connected individuals (namely B, C and D) so two bits are necessary to encode them. The temporary local identifier at each level starts with counter set to 1 and is incremented by 1, and each of these individuals is prefixed with the identifier of their local root. Thus the identifier of B in Figure 1(b) is 101 (with the left most bit inherited from A and 01 computed at this level). This computation is performed recursively until all nodes are assigned a value. A final step consists in normalizing the temporary identifier: all identifiers have to be encoding with the same binary length. In our example, F and G are the identifiers with the longest binary encoding (*i.e.*, 6 bits) so all nodes of the tree are right-completed with bits set at 0 to reach the same length. The identifiers for each node in Figure 1(b) are displayed in each box, the gray 0 of an identifier are the results of the normalization while the local fragment is in black.

Given this local identifier strategy, we can easily find whether a given node is in the transitive closure of another node of that same graph. This operation is based on checking whether the subtraction of two identifiers is contained in a given interval. Let consider the connected component graph of a transitive property. Due to the normalization step, all identifiers of this connected component are encoded using n bits. Moreover, we introduce a function, *localLength*, that returns the non-normalized binary encoding length of a node. For instance, in Figure 1(b), *localLength* of A,C and G are respectively 1, 3 and 6. For two nodes of this graph, $\alpha$ and $\beta$, $\beta$ is in the transitive closure of $\alpha$ if $\beta - \alpha$ is included in $[0, 2^{n-localLength(\alpha)}[$.

Using this approach, we can efficiently compute that G is in the transitive closure of A, B and E but not of B, D and F.

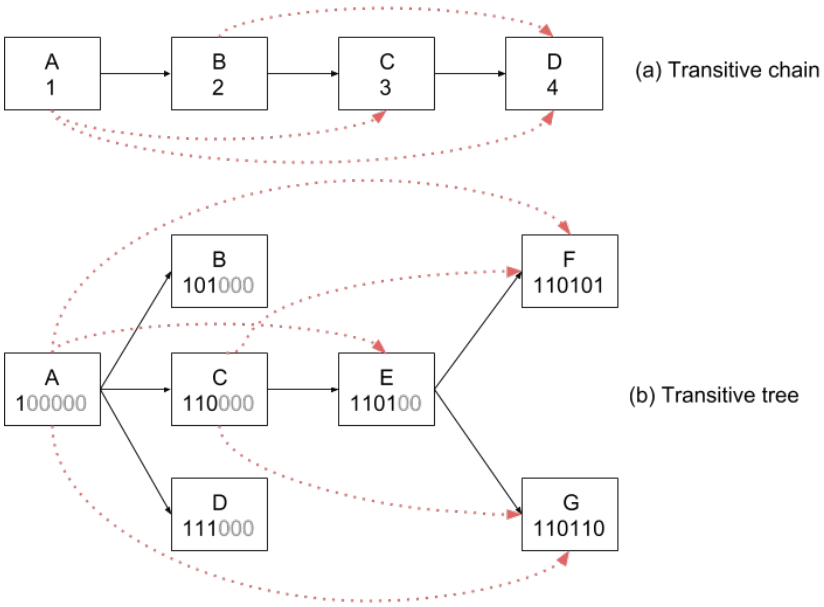


**Fig. 1:** A chain and a tree of a transitive property. Dotted red arrows correspond to the transitive closure. In each node, a label and its local identifier (lid)

Since our `owl:sameAs` encoding scheme, *i.e.*, a tuple $\langle cliqueId, localId \rangle$, does not rely on a local identification total order, it is possible to compose `owl:sameAs` identifiers with transitive ones. This means that individuals involved in a chain or tree transitive structure can be involved in a `owl:sameAs` clique by reusing their identifiers in the sameAs encoding scheme. In such a case, the $localId$ corresponds to the whole transitive identifier.

## 3 Query processing in the presence of transitive properties

We present a sketch of query processing with a BGP involving a transitive property. Due to space limitation, we consider a BGP containing a single triple pattern asking for all subjects (respectively objects) related, via a transitive property, to an object (respectively subject).

### 3.1 Query encoding

As with most RDF triples, the query needs to be translated to an identifier-based form which requires look-ups to several LiteMat dictionaries. Using the property dictionary, we obtain the identifier of the property and we will also get

the information that this property is transitive. Then, we search for the identifier of the object (respectively subject). In a full materialization case, this identifier corresponds to a single identifier while in the case of LiteMat, it corresponds to a 4-tuple identifier, *i.e.*, ⟨fid, pid, ccid, lid⟩.

### 3.2 Variable assignments

While the full materialization approach requires a complete scan over all triples plus an extraction from the individuals dictionary, the same query can be answered much more efficiently with LiteMat. Intuitively, due to our semantic-aware encoding of LiteMat, we can rely solely on some simple computation to directly search for answers in the dictionary. In fact, we will search for all individual dictionary entries where the key is of the form $\langle fid, pid, ccid, X \rangle$ where one of the following computations is performed:

- if $fid$ corresponds to a chain, *i.e.*, $fid = 0$, then the system retrieves all values lower then *lid*.
- if $fid$ is a tree, *i.e.*, $fid = 1$, then the system retrieves all values comprised between ]lid, ((lid >> lid encoding length)+1)<< lid encoding length)[



## 4 Evaluation

### 4.1 Experimental setting

The evaluation was conducted on a cluster composed of three DELL PowerEdge R410 running a Debian GNU/Linux distribution with a 3.16.0-4-amd64 kernel version. Each machine has 64GB of DDR3 RAM, a 900GB 7200rpm SATA disk and two Intel Xeon E5645 processors. Each processor is constituted of 12 cores running at 2.40GHz and allowing to run two threads in parallel (hyper threading). The machines are connected via a 1GB/s Ethernet network adapter. We used Spark version 2.3.2 and implemented all experiments in Scala version 2.11.6. More details on the scripts can be found here[4]. The Spark configuration of our evaluation runs our prototype on a subset of the cluster corresponding to 36 cores and 24GB of RAM per machine.

### 4.2 Datasets and query workload

In this evaluation, we are aiming to test and stress our approaches with different sizes of transitive chains and trees. We thus resort to a synthetic benchmark solution, namely the Lehigh University BenchMark (LUBM)[2], a well-established benchmark on the university domain that contains a transitive property, named `lubm:subOrganisationOf` . Since both the `rdfs:domain` and `rdfs:range` of

[4] https://github.com/xwq610728213/LitematPlusPlus

this property are the Organization concept, we can use it to create long chain and tree structures.

Table 2 presents the datasets that we are using throughout this experimentation. Intuitively, the name of each dataset describes the number of universities, *e.g.*, 5K or 10K for respectively 5.000 and 10.000 universities, a letter, *i.e.*, either 'c' or 't' which respectively stand for chain and tree structures and a number that corresponds to the maximum depth of the structure. Note that the 5K_c20 and 10_c20 correspond to large shallow trees which are supposed to mitigate the advantages provided by LiteMat. In total, 10 datasets are evaluated, 4 chains and 6 trees.

In this section, we are providing a preliminary evaluation of our query processing solution. Due to space limitation, we are only considering answering a single triple pattern that retrieves either all the ancestors or descendants of a group in the transitive closure of the `lubm:subOrganisationOf` property. These two queries have been executed over the some of the 10K datasets and respectively correspond to `SELECT ?X WHERE {?x lubm:subOrganisationOf C}` to compute ancestors and `SELECT ?X WHERE {C lubm:subOrganisationOf ?x }` to retrieve descendants where C is an individual involved in the queried dataset, *e.g.*, `<http://www.Department10.University1000.edu/ResearchGroup1>`.

This limited evaluation already provides some valuable insight on the potential of LiteMat query processing with transitive properties.

| Dataset name | Depth sizes [min,max] | #Branches [min,max] | #Triples | Size (MB) | #Triples materialized | Increase due to materialization |
|---|---|---|---|---|---|---|
| 5K_c20 | [10, 20] | 1 | 1.689.907 | 318,4 | 23.579.485 | x 14 |
| 5K_c100 | [20, 100] | 1 | 6.752.637 | 1.280 | 230.004.339 | x 34.1 |
| 5K_t5 | [10, 20] | [1, 5] | 5.062.616 | 957.9 | 39.362.874 | x 7.8 |
| 5K_t10 | [20, 100] | [5, 10] | 12.624.667 | 2.400 | 109.223.271 | x 8.7 |
| 5K_t20 | [2, 5] | [10, 20] | 5.898.803 | 1.120 | 14.064.044 | x 2.4 |
| 10K_c20 | [10, 20] | 1 | 3.376.055 | 636,8 | 48.712.856 | x 14.4 |
| 10K_c100 | [20, 100] | 1 | 13.522.653 | 2.560 | 461.042.361 | x 34.1 |
| 10K_t5 | [10, 20] | [1, 5] | 10.119.755 | 1.920 | 71.304.115 | x 7.0 |
| 10K_t10 | [20, 100] | [5, 10] | 25.260.771 | 4.800 | 216.690.752 | x 8.6 |
| 10K_t20 | [2, 5] | [10, 20] | 11.804.988 | 2.240 | 28.155.826 | x 2.4 |

**Table 2:** Characteristics of evaluated datasets

### 4.3 Compression and encoding performance

In this section, we are mainly interested in two performance dimensions: the memory space reduction provided by LiteMat compared to a full materialization and the duration of LiteMat's encoding against the full materialization computation.

Figure 2 presents comparisons of the memory space required by the LiteMat approach against a full materialization. In the latter approach, both the set of materialized triples as well as the dictionaries are required to answer inference-enabled SPARQL queries while in the case of LiteMat, only the dictionaries are necessary. The figure emphasizes that, for any datasets, both dictionaries are about the same size, with the ones of FullMat being a little bit more compact for trees due to the overhead LiteMat identifiers, *i.e.*, a long value for the materialization against a 4-tuple of long values and an integer for LiteMat.

Obviously, for long chains and large trees, the amount of materialized triples can be quite important, *i.e.*, for 5K_c100 and 10K_c100, the set of materialized triples of 34 times larger than to their original triple sets (Figure tab:datasets and the LiteMat approaches correspond to only 10% of their sizes. Considering 5K_c20 and 10K_c20, LiteMat's approach is still around 70% of total materialized approach.

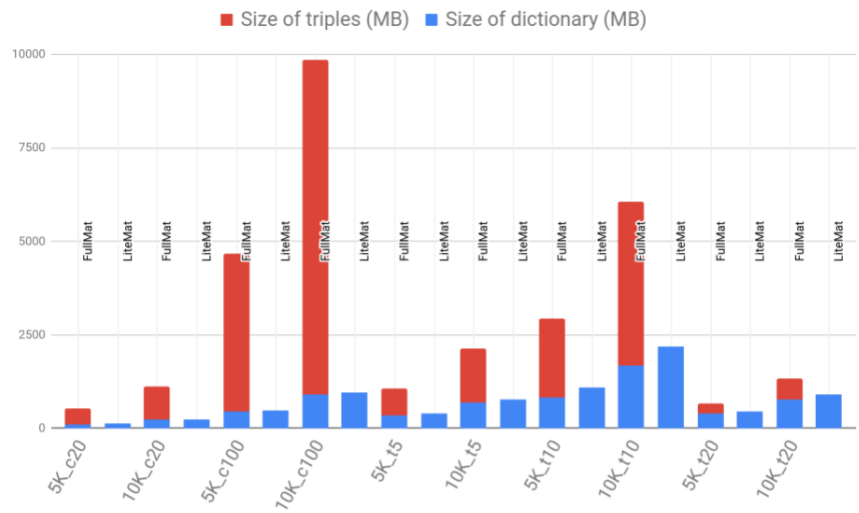


**Fig. 2:** Memory space required by LiteMat vs a full materialization

Figure 3 provides some details on the duration of the different computation steps involved in both the full materialization and LiteMat approaches. The common steps of these two approaches are the loading of the dataset and the computation of the connected components. We can see that the time taken by the former is quite negligible compared to the other tasks. Unsurprisingly, the computation of the connected components takes a lot of time on all experimentation. The LiteMat encoding and full materialization share a common naïve

encoding of individuals step (which is included in both times). Overall, we note that for chains of a transitive property, the difference between both approaches is not significant, *i.e.*, LiteMat is only between 2 and 3% faster than the full materialization. This is not true for structures taking the form of a tree. In that case, LiteMat's encoding is 45 to 50% faster when the depths of structure is relative large for transitive properties, *i.e.*, [10,20] and [20,100]. We consider that this is mainly due to the recursive parsing of the tree to compute the transitive closure. The duration difference between the two approaches is less important, *i.e.*, around 11%, when the tree structure depths lies in the [2,5] range.


**Fig. 3:** Durations (in seconds) for the full materialization and LiteMat approaches

### 4.4 Query processing

Our preliminary evaluation of the query processing consists of a cold retrieval of all descendants of a give individual and the average of five hot queries that retrieve all descendants (*i.e.*, hot1) and ancestors (*i.e.*, hot2). Figure 4 provides measures conducted on the largest datasets of our experimentation, *e.g.*, 10K_c100 and 10K_t20 of respectively 9.8GB and 6GB for the materialized approaches. In order to provide a complete overview of the approaches, In this figure, all measures (loading time, cold, hot1 and hot2) emphasize shorter execution times for LiteMat. Considering the loading times, LiteMat is between 6 to 10 times faster than the complete materialization. This is mainly due to the fact that LiteMat solely relies on the dictionaries and not on the triples set. This aspect impacts the cold runs where LiteMat is between 2 and 8 times faster then

the full materialization. Finally, for hot runs, LiteMat is 2 to 3 times faster than a complete materialization.

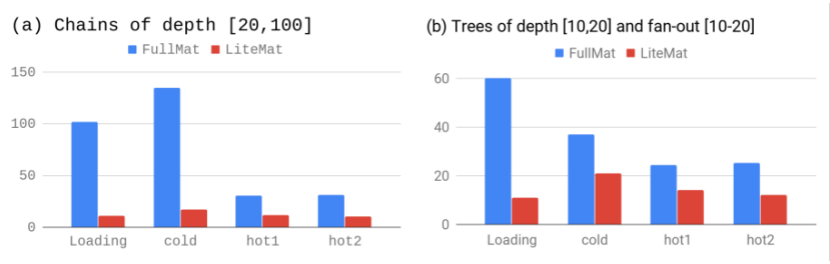


**Fig. 4:** Query processing on chains and trees with full materialization and liteMat (times in seconds)

## 5 Conclusion

In this paper we have extended the expressiveness of LiteMat to reach the level of RDFS++. This has been achieved by pushing the original logical approach of LiteMat that consists in assigning meaningful identifiers to elements of the TBox and the ABox. The evaluation of transitive structures taking the forms of chains and and trees is quite convincing is terms of memory space economized, speed of encoding and efficiency of query processing. Nevertheless, there is room for improvement in directions such as a more efficient support of graph transitive structures and the fact that certain individuals can be contained in structures of different transitive properties. On the implementation side, we are considering using indexed Spark abstractions to speed up query processing and are considering algorithms such those presented in [3] to compute connected components.

## References

1. O. Curé, H. Naacke, T. Randriamalala, and B. Amann. Litemat: A scalable, cost-efficient inference encoding scheme for large RDF graphs. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 1823–1830, 2015.
2. Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
3. R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilvitskii. Connected components in mapreduce and beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 18:1–18:13, New York, NY, USA, 2014. ACM.
4. S. Muñoz, J. Pérez, and C. Gutierrez. Simple and efficient minimal rdfs. *Web Semant.*, 7(3):220–234, Sept. 2009.

5. X. Ren, O. Curé, H. Naacke, J. Lhez, and L. Ke. Strider$^{\mathrm{r}}$: Massive and distributed RDF graph stream reasoning. In *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, pages 3358–3367, 2017.
6. M. Rodríguez-Muro and D. Calvanese. High performance query answering over dl-lite ontologies. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*, KR'12, pages 308–318. AAAI Press, 2012.