The Design and Implementation of Minimal* RDFS Backward Reasoning in 4store

Manuel Salvadores[1], Gianluca Correndo[1], Steve Harris[2],
Nick Gibbins[1], and Nigel Shadbolt[1]

[1] Electronics and Computer Science,
University of Southampton, Southampton, UK
{ms8,gc3,nmg,nrs}@ecs.soton.ac.uk
[2] Garlik Ltd, UK
steve.harris@garlik.com

Abstract. This paper describes the design and implementation of *Minimal* RDFS semantics based on a backward chaining approach and implemented on a clustered RDF triple store. The system presented, called *4sr*, uses 4store as base infrastructure. In order to achieve a highly scalable system we implemented the reasoning at the lowest level of the quad store, the *bind* operation. The *bind* operation runs concurrently in all the data slices allowing the reasoning to be processed in parallel among the cluster. Throughout this paper we provide detailed descriptions of the architecture, reasoning algorithms, and a scalability evaluation with the LUBM benchmark. *4sr* is a stable tool available under a GNU GPL3 license and can be freely used and extended by the community[1].

Keywords: Triple Store, Scalability, Reasoning, RDFS, SPARQL, 4store.

1 Introduction

RDF stores - or triple stores - implement some features that make them very attractive for certain type of applications. Data is not bound to a schema and it can be asserted directly from RDF sources (e.g. RDF/XML or Turtle files) due to their native support of Semantic Web data standards. But the most attractive characteristic is the possibility of implementing an entailment regime. Having entailment regimes in a triple store allows us to infer new facts, exploiting the semantics of properties and the information asserted in the knowledge base. To agree on common semantics, some standards have arisen for providing different levels of complexity encoded in a set of inference rules, from RDF and RDFS to OWL and RIF, each of them applicable to different scenarios.

Traditionally, reasoning can be implemented via forward chaining (FC henceforth), backward chaining (or BC), or hybrid algorithms (a mixture of the two).

* Minimal RDFS refers to the RDFS fragment published in [8].
[1] Preliminary results were presented at the Web-KR[3] Workshop [10] and demoed at ISWC 2010 [9].

G. Antoniou et al. (Eds.): ESWC 2011, Part II, LNCS 6644, pp. 139–153, 2011.

FC algorithms tend to apply a set of inference rules to expand the data set before or during the data assertion phase; with this approach, the database will contain all the facts that are needed when a query is issued. On the other hand, BC algorithms are goal directed and thus the system fires rules and/or axioms at runtime to find the solutions. Hybrid algorithms use a combinations of forward and backward chaining. The pros and cons of these approaches are well known to the AI and database community. FC approaches force the system to retract entailments when there is an update or insert, making data transactions very expensive. Complete materialisation of a knowledge base could lead to an explosion of data not manageable by current triple store technology. Backward chaining performs better for data transactions and the size of the KB is smaller, but queries tend to have worse performance.

Reasoners are also classified by their level of completeness. A reasoner can claim to be complete over an entailment regime $\mathcal{R}$ *if and only if*: (a) it is able to detect all entailments between any two expressions; and (b) it is able to draw all valid inferences; according to $\mathcal{R}$ semantics. For some types of applications a complete reasoner might be required but one should assume that higher completeness tends to degrade query response time. There is, therefore, a clear compromise: performance versus completeness and the old AI debate about speed and scalability versus expressiveness. In our specific case, *4sr* excludes a subset of RDFS semantics rarely used by Semantic Web applications and implements the semantics from the *Minimal* RDFS fragment [8]. *4sr* entailment is complete considering *Minimal* RDFS semantics but incomplete for the full normative RDFS semantics [5]. In that sense, our main contribution is a system that proves that *Minimal* RDFS semantics can scale if implemented in a clustered triple store. In comparison to our previous research, this paper formalizes *4sr* against *Minimal* RDFS, and also describes the components to be synchronized among the cluster and benchmarks the bind operation to test its scalability.

The remainder of the paper is as follows: Section 2 describes the related research in the area and introduces basic *4store* concepts and *Minimal* RDFS. Section 3 introduces and formalizes *4sr*'s distributed model. Section 4 explains the design and implementation of *4sr* explaining the modifications undertaken in *4store*. Section 5 studies the scalability of the new *bind* operation by benchmarking it under different conditions, and finally Section 6 analyses the results achieved by this work.

2 Related Work

4sr is a distributed backward chained reasoner for *Minimal* RDF/RDFS, and to our knowledge is the first system with such characteristics. However a number of related pieces of research informed our work.

Some current tools implement monolithic solutions using FC, BC or hybrid approaches. They use different types of back-ends as triple storage such as RDBMS, in memory, XML or native storage. Examples of these tools are Jena [2],

Pellet [11] and Sesame [1]. These tools can perform RDFS reasoning with datasets containing up to few million triples. But, even though they have played a key role in helping Semantic Web technologies to get adopted, their scalability and performance is still a major issue.

BigOWLIM[2] is one of the few enterprise tools that claims to perform OWL reasoning over billions of triples. It can run FC reasoning against the LUBM (90K,0) [3] , which comprises around 12 billion triples. They materialize the KBs after asserting the data which means that BigOWLIM has to retract the materialization if the data is updated. There is no information on how this tool behaves in this scenario, even though they claim their inferencing system is retractable.

In the context of distributed techniques, [15] performs FC parallel reasoning to expand the RDFS closure over hundreds of millions of triples, and it uses a C/MPI platform tested on 128 core infrastructure with the LUBM 10k dataset. [13] pursues a similar goal and using MapReduce computes the RDFS closure over 865M triples in less than two hours. A continuation of this work has been presented in [12] providing a parallel solution to compute the OWL Horst regime. This solution, built on top of Hadoop, is deployed on a cluster of 64 machine and has been tested against a synthetic data set containing 100 billion triples and a 1.5 billion triples of real data from the LDSR and UniProt datasets.

[7] presented a novel method based on the fact that Semantic Web data present very skewed distributions among terms. Based on this evidence, the authors present a FC algorithm that works on top of data flows in a p2p infrastructure. This approach reported a materialization of RDFS for 200 million triples in 7.2 minutes on a cluster of 64 nodes.

Obviously, in the last 2-3 years there has been a significant advance on materialization of closures for both RDFS and OWL languages. However very little work has been presented on how to query such vast amounts of data and how to connect those solutions with SPARQL engines. Furthermore, these types of solutions are suitable for static datasets where updates and/or deletes are sparse or non-existent. Applying this mechanism to dynamic datasets with more frequent updates and deletes whose axioms need to be recomputed will lead to processing bottlenecks.

To avoid these bottlenecks, progress on backward chained reasoning is required. To date, there has been little progress on distributed backward chained reasoning for triple stores. [6] presented an implementation on top of DHTs using p2p techniques. So far, such solutions have not provided the community with tools, and recent investigations have concluded that due to load balancing issues they cannot scale [7].

With the SPARQL/Update specification to be ratified soon, we expect more triple/quad stores to implement and support transactions, which makes BC reasoning necessary at this juncture.

[2] http://www.ontotext.com/owlim/big/index.html accessed 21/06/2010

2.1 Minimal RDFS Reasoning

RDFS extends RDF with a schema vocabulary, a regime that contains semantics to describe light-weight ontologies. RDFS focusses mostly on expressing class, property and data type relationships and its interpretations can potentially generate inconsistencies. For instance, by using `rdfs:Literal` as `rdfs:domain` for a predicate `P`, any statement (`S`,`P`,`O`) with `P` as predicate would entail (`S a rdfs:Literal`) which is clearly inconsistent since RDF doesn't allow literals to be subjects (see section 4.3 [5]). Another issue with RDFS interpretaions is decidability. There is a specific case when using container memberships (`rdfs:ContainerMembershipProperty`) that can cause an RDFS closure to be infinite [14]. Other similar cases like these appear when constructing ontologies with different combinations of the RDF reification vocabulary, `rdf:XMLLiteral`, disjoint XSD datatypes, etc. A complete RDFS reasoner must examine the existence of such paradoxes and generate errors when models hold inconsistencies. Consistency checking is computationally very expensive to deal with, and reduces query answering performance, and one should question the applicability of the semantics that generate inconsistencies for most type of applications.

Another known issue with RDFS is that there is no differentiation between language constructors and ontology vocabulary, and therefore constructors can be applied to themselves. `(P rdfs:subPropertyOf rdfs:subPropertyOf)`, for example, it is not clear how an RDFS reasoner should behave with such construction. Thankfully this type of construction is rarely used on Semantic Web applications and Linked Data.

[8] summarizes all the above problems among many others motivating the use of an RDFS fragment, called *Minimal* RDFS. This fragment preserves the normative semantics of the core functionalities avoiding the complexity described in [5]. Because *Minimal* RDFS also avoids RDFS constructors to be applied to themselves, it has been proven that algorithms to implement reasoning can be bound within tight complexity limits (see section 4.2 in [8]).

Minimal RDFS is built upon the *ρdf* fragment which includes the following RDFS constructors: `rdfs:subPropertyOf`, `rdfs:subClassOf`, `rdfs:domain`, `rdfs:range` and `rdf:type`[3]. It is worth mentioning that this fragment is relevant because it is non-trivial and associates pieces of data external to the vocabulary of the language. Contrarily, predicates left out from the *ρdf* fragment essentially characterize inner semantics in the ontological design of RDFS concepts.

2.2 4store

4store [4] is an RDF storage and SPARQL query system that became open source under the GNU license in July 2009. Since then, a growing number of users have been using it as a highly scalable quad store. *4store* provides a stable infrastructure to implement decentralized backward chained reasoning: first

[3] For the sake of clarity we use the same shortcuts as in [8] ([sp], [sc] [dom] and [type] respectively).

because it is implemented as a distributed RDF database and second because it is a stable triple store that has been proven to scale up to datasets with 15G triples in both enterprise and research projects.

4store distributes the data in non-overlapping segments. These segments are identified by an integer and the allocation strategy is a simple *mod* operation over the subject of a quad. In the rest of the paper we will represent a quad as a 4-tuple where the first element is the model URI and the remainder is the typical RDF triple structure (subject, predicate, object), and the quad members will be accessed as q_m, q_s, q_p and q_o respectively.

The distributed nature of 4store is depicted in [4], and can be simplified as follows. The data segments are allocated in *Storage Nodes* and the query engine in a *Processing Node*. The *Processing Node* accesses the *Storage Nodes* via sending TCP/IP messages. It also decomposes a SPARQL query into a query plan made of quad patterns. For each quad pattern, it requests a *bind* operation against all segments in each *Storage Node*. The bind operations are run in parallel over each segment and receive as input four lists $\{B_M, B_S, B_P, B_O\}$ that represent the quad patterns to be matched.

3 *Minimal* RDFS and *4sr*'s Distributed Model

This section takes *Minimal* RDFS algorithms from [8] and reformulates them to be implemented as goal directed query answering system (backward chain) over a distributed infraestructure.

We define a knowledge base (KB) as set of n graphs:

$\mathcal{KB} = \{G_1, G_2, G_3, ...G_n\}$

where a graph G_i is a set of quads of the form (m, s, p, o). We also define the set of segments in a distributed RDF store as:

$\mathcal{S} = \{S_0, S_1, S_2, ...S_{m-1}\}$

Quads can be distributed among segments based on different strategies. In our case, 4store's distribution mechanism applies a *mod* operation over a hash[4] of every quad's subject. Therefore, a quad (m, s, p, o)[5] from graph G_i will be allocated in segment S_j if $j = hash(s)\ mod\ m$.

According to our subject based distribution policy, a quad gets allocated to a segment regardless of the graph G_i they belong to, and a graph G_i will be split among m' number of segments where $0 < m' \leq m$. It is worth mentioning that this type of data distribution disseminates the data evenly among $\mathcal{S}$ making no assumptions about the structure or content of the resources.

We now define the vocabulary of RDFS terms supported:

$\rho df = \{sc, sp, dom, range, type\}$

A quad (m, s, p, o) is an *mrdf*-quad *iff* $p \in \rho df$ - {`type`}, and G_{mrdf} is a graph with all the *mrdf*-quads from every graph in $\mathcal{KB}$. It is important to mention that the *mrdf*-quads are the statements we need to have in all the data segments in

[4] Although 4store is parameterizable, most deployments use UMAC as hash function.

[5] For the sake of simplicity we will not develop a whole RDF graph model here, we assume that quads are just four-element entities.

order to apply the deductive rules from [8], this process to replicate G_{mrdf} in all segments is described throughout the rest of this section. The following rules, extracted from [8], implement the *Minimal* RDFS semantics:

$$(sp_0)\ \frac{(\_, A, sp, B)(\_, B, sp, C)}{(G_e, A, sp, C)} \qquad (sp_1)\ \frac{(\_, A, sp, B)(\_, X, A, Y)}{(G_e, X, B, Y)}$$

$$(sc_0)\ \frac{(\_, A, sc, B)(\_, B, sc, C)}{(G_e, A, sc, C)} \qquad (sc_1)\ \frac{(\_, A, sc, B)(\_, X, type, A)}{(G_e, X, type, B)}$$

$$(dom_0)\ \frac{(\_, A, dom, B)(\_, X, A, Y)}{(G_e, X, type, B)} \qquad (ran_0)\ \frac{(\_, A, range, B)(\_, X, A, Y)}{(G_e, Y, type, B)}$$

$$(dom_1)\ \frac{(\_, A, dom, B)(\_, C, sp, A)(\_, X, C, Y)}{(G_e, X, type, B)} \qquad (ran_1)\ \frac{(\_, A, range, B)(\_, C, sp, A)(\_, X, C, Y)}{(G_e, X, type, B)}$$

These rules have been reformulated taking into account that we are dealing with a quad system and not just with triples. The m element of the quads is irrelevant for the rule condition; in the consequence the m element takes the value of G_e which we consider the graph of entailments contained in $\mathcal{KB}$. The model element in the quad (m) does not play any role unless that the SPARQL query being processed projects named graphs into the query resultset - see section 6.1 Future Work on named graph semantics.

At this point we have the definition of $\mathcal{KB}$, $\mathcal{S}$, and a set of deductive rules for ρdf. Every segment in $\mathcal{S}$ contains a non-overlapping set of quads from $\mathcal{KB}$. One important aspect of 4store's scalability is that the bind operation runs concurrently on every segment of $\mathcal{KB}$. Therefore, we need to look at data inter-dependency in order to investigate rule chaining locks that can occur between segments.

The chain of rules dependencies for *Minimal* RDFS is shown in Figure 1. In the figure, *mrdf*-quads are marked with a '*', and quads in bold are initial quads not triggered by any rule. The rest of the quads are entailed by triggering one or more rules. There is an interesting characteristic in the chain of rules that can be triggered in *Minimal* RDFS: in any possible chain of rules, only

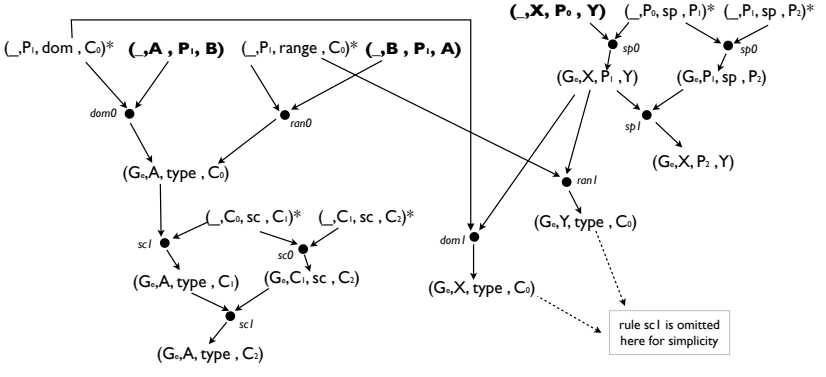

Fig. 1. Rule Chain Tree

one non-*mrdf*-quad that is not in G_e is used. This argument is backed up by the fact that the conditions in the set of deductive rules contain zero or one non-*mrdf*-quads. Therefore to implement distributed reasoning, the only data we need to replicate in every segment of $\mathcal{S}$ is G_{mrdf}, so that G_{mrdf} is accessible to the parallel execution of *bind*. This finding drives *4sr*'s design and it is a novel and unique approach to implement RDFS backward chained reasoning.

4 *4sr* Design and Implementation

The RDFS inferencing in *4sr* is based on two new components that have been incorporated into *4store's* architecture:

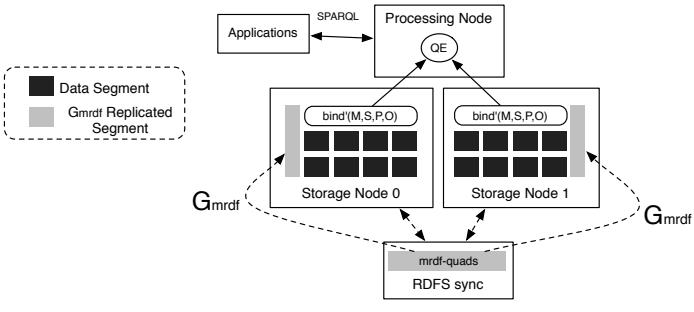

Fig. 2. 4sr Architecture

- **RDFS Sync:** A new processing node to replicate G_{mrdf} called *RDFS sync*. This node gathers all the quads that satisfy the condition to be held in G_{mrdf} from all the segments and replicates them to every Storage Node keeping them synchronized. After every import, update, or delete, this process extracts the new set of quads from G_{mrdf} in the KB and sends it to the Storage Nodes. Even for large KBs this synchronization is fast because G_{mrdf} tends to be a very small portion of the dataset.
- **bind':** The new bind function matches the quads, not just taking into account the explicit knowledge, but also the extensions from the RDFS semantics. This modified *bind'* accesses G_{mrdf} to implement backward chain reasoning. *bind'* is depicted in detail in Section 4.1.

Figure 2 shows in the architecture how *bind'* and RDFS Sync interact for a hypothetical two storage-node deployment. The dashed arrows refer to the messages exchanged between the RDFS Sync process and the Storage Nodes in order to synchronize G_{mrdf}; the arrows between the Processing Node and the Storage Nodes refer to the *bind* operation requested from the Query Engine (QE).

4.1 bind' and *Minimal* RDFS Semantics

In [10] we presented the logical model for *bind'*, a preliminary work that did not take into account the *Minimal* RDFS fragment and was built upon a subset of semantics from [5].

In this paper, we present a remodelled *bind'* that has been reimplemented according to the *Minimal* RDFS semantics and the distributed model described in Section 3. To implement the model, we first consider the following modification of rules dom_1 and ran_1. dom'_1 and ran'_1 - shown below - can replace the original dom_1 and ran_1 keeping *Minimal* RDFS original semantics the same.

$$(dom'_1)\ \frac{(\_, A, dom, B)(\_, C, sp, A)}{(G_e, C, dom, B)} \qquad (ran'_1)\ \frac{(\_, A, range, B)(\_, C, sp, A)}{(G_e, C, range, B)}$$

The rationale for such a replacement is based on the fact chaining dom_0 and dom'_1 generate equivalent entailments to just dom_1. And, similarly, chaining $range_0$ and $range'_1$ is the same as $range_1$.

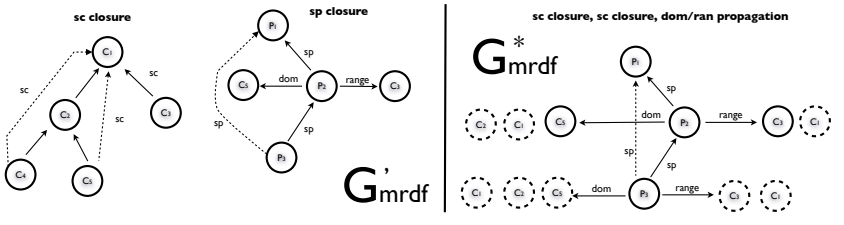

Fig. 3. Construction of G'_{mrdf} and G^*_{mrdf} from G_{mrdf}

Our design relies on the definition of two entailed graphs G'_{mrdf} and G^*_{mrdf}; to deduce these graphs we will entail dom'_1 and $range'_1$ over G_{mrdf}. That process is depicted in Figure 3, and the generated graphs hold the following characteristics:

- G'_{mrdf} is a graph that includes G_{mrdf} and the closure of `sp` and `sc` by applying rules sp_0 and sc_0.
- G^*_{mrdf} is a graph that includes G'_{mrdf} plus the deductions from dom'_1 and $range'_1$.

We also define the following operations over G'_{mrdf} and G^*_{mrdf}, where X is considered an arbitrary resource in $\mathcal{KB}$:

- $G'_{mrdf|sc}(X)$ as the subclass closure of X in G'_{mrdf}.
- $G'_{mrdf|sp}(X)$ as the subproperty closure of X in G'_{mrdf}.
- $G^*_{mrdf|dom}(X)$ as the set of properties in G^*_{mrdf} with X as domain.
- $G^*_{mrdf|range}(X)$ as the set of properties in G^*_{mrdf} with X as range.
- $G^*_{mrdf|list}()$ the set of every inferable domain or range in G^*_{mrdf}.
- $G^*_{mrdf|bind}(s, p, o)$ the list of statements that is retrieved from a normal bind operation for a triple pattern (s, p, o).

With these functions we define the access to the graphs G'_{mrdf} and $G_{mrdf}*$, which we should emphasize are accessible to every segment in $\mathcal{S}$. These operations, therefore, will be used for the definition of *bind'*.

A bind operation in 4store is requested by the query processor as we explained in section 2.2. The *bind* receives 4 multisets with the resources to be matched or `NULL` in case some part of the quad pattern is unbound, so a bind to be executed receives as input (B_M, B_S, B_P, B_O). We omit in this paper the description of how the query processor works in 4store, , but for the sake of understanding the system we depict a very simple example:

```
SELECT ?name ?page WHERE { ?x foaf:name ?name .
?x foaf:homePage ?page . ?x foaf:basedNear dbpedia:London }
```

A potential query plan in 4store would issue two bind operations, first the most restrictive query pattern:

$B_0 \Leftarrow bind(NULL, NULL, \{basedNear\}, \{London\})$
$B_1 \Leftarrow bind(NULL, B_{0s}, \{name, homePage\}, NULL\})$

The only point to clarify is that the second bind receives B_{0s} as B_S (B_{0s} refers to the subject element of B_0). The original bind operation in *4store* is made of four nested loops that traverse the indexes in an optimized manner. The ranges of the loops are the input lists (B_M, B_S, B_P, B_O) which are used to build up the combination of patterns to be matched. For simplicity, we draw the following bind function:

Simplified Algorithm of the original bind in 4store:
Input: B_M, B_S, B_P, B_O,segment **Output:** r - a list of quads

```
Let B* be the list of pattern combinations of B_M, B_S, B_P, B_O
For every pattern in B*
    Let T be the radix tree in segment with optimized iterator for pattern
    For every quad in T
        If (pattern_m = ∅ OR pattern_m = quad_m) AND
           (pattern_s = ∅ OR pattern_s = quad_s) AND
           (pattern_p = ∅ OR pattern_p = quad_p) AND
           (pattern_o = ∅ OR pattern_o = quad_o)
           append quad into r
```

4store's real implementation uses radix tries as indexes, and the index selection is optimized based on the pattern to match. This simplified algorithm plays the role of explaining our transformation from original *bind* to the new *bind'* that implements *Minimal* RDFS semantics.

Bind' Algorithm in 4sr:
Input: B_M, B_S, B_P, B_O,segment **Output:** r - a list of quads

```
Let B* be the list of pattern combinations of B_M, B_S, B_P, B_O
For every pattern in B*
```

```
(a) If |G'_{mrdf|sp}(pattern_p)| > 1
       append to r bind(pattern_m, pattern_s, G'_{mrdf|sp}(pattern_p), pattern_o)
(b) Else If pattern_p = ∅
       For every pred in segment
           append to r bind'(pattern_m, pattern_s, pred, pattern_o)
(c) Else If pattern_p = type
(c0)   If pattern_o ≠ ∅
           For s in bind(∅,∅,G*_{mrdf|dom}(pattern_o),∅)
               append to r (G_e, sol_s, type, pattern_o)
           For s in bind(∅,∅,G*_{mrdf|ran}(pattern_o),∅)
               append to r (G_e, sol_o, type, pattern_o)
           append to r bind(pattern_m, pattern_m, type, G'_{mrdf|sc}(pattern_o))
(c1)   Else
           For object in G*_{mrdf|list}()
               append to r bind'(pattern_m, pattern_s, type, object)
(d) Else If pattern_p ∈ (sc, sp, range, dom)
       append to r G*_{mrdf|bind}(pattern_s, pattern_p, pattern_o)
(e) Else
       append to r bind(pattern_m, pattern_s, pattern_p, pattern_o)
```

This algorithm can be seen as a wrapper of the original *bind* operation; it basically rewrites every combination of quad pattern to be matched according to the *Minimal* RDFS deductive rules. Each of the `if` conditions in the algorithm takes care of one case of backward chain inference. These are described below:

(a) The `if` condition tests whether the closure has more than one element. In such cases we operate `sp` inference by calling the *bind* with an extended Bp made by the closure of $pattern_p$. Such a closure is obtained through $G'_{mrdf|sp}(pattern_p)$.

(b) If the predicate pattern is unbound then we recursively call again *bind'* for every predicate in the segment, keeping intact the rest of the elements in the pattern.

(c) This branch is dedicated to the inference with higher complexity, when $pattern_p$ matches RDF `type`. It is divided in two sub-cases:

  (c0) In this case, the pattern matches a specific object ($pattern_o \neq \emptyset$). The first loop binds all the properties for which $pattern_o$ can be inferred as a domain ($G^*_{mrdf|dom}(pattern_o)$). Each solution appends (G_e,sol_s,$type$, $pattern_o$) where sol_s is the subject from the object element from a single solution.
  The second loop is analogous but for ranges. It is worth noticing that this second loop appends as subject sol_o. The reason for this is that in rule ran_o mentions the object that gets the class membership.
  The last append runs the original bind extending $pattern_o$ to the closures of subclasses $G'_{mrdf|sc}(pattern_o)$.

  (c1) This is the opposite case; the object to be matched is null. For such cases we recursively call *bind'* for every inferable class in $G^*_{mrdf|list}()$. The calls generated in this case will be processed in `c0` in subsequent steps.

(d) For patterns where the predicate is any of $(sc, sp, range, dom)$, the pattern match comes down to a simple bind operation over the replicated graph - $G^*_{mrdf|bind}(s, p, o)$.
(e) No reasoning needs to be triggered and a normal bind is processed. The rationale for no reasoning being triggered comes from the fact that in the set of deductive rules, reasoning is processed for patterns where p is one of $(type, sc, sp, range, dom)$ or p is part of a `sp` closure with more than one element. These two conditions are satisfied in (a) and (c). (b) covers the case of $pattern_p = \emptyset$ for which a recursive call for every predicate is requested.

5 LUBM Scalability Evaluation

This evaluation studies *4sr*'s distributed model and its behaviour with different configurations in terms of number of distributed processes - segments - and size of datasets. This analysis is based upon the LUBM synthetic benchmark [3]; we have used 6 different datasets LUBM(100), LUBM(200), LUBM(400), ... , LUBM (1000,0). These datasets progressively grow from 13M triples - LUBM(100,0) - triples to 138M triples LUBM(1000,0). In [10] we presented a preliminary benchmark that demonstrates that *4sr* can handle SPARQL queries with up to 500M triple datasets; this benchmark shows the overall performance of the whole system. The type of benchmark we analyse in this paper, instead of studying performance for big datasets, studies how the bind operation behaves when trying to find solutions that require *Minimal* RDFS reasoning under different conditions, i.e. to see how the bind operations behaves when adding more processors or when the data size is increased. Studying just the bind operation also leaves out components of *4store* that are not affected by the implementation of reasoning, like the query engine.

Our deployment infrastructures are made of two different configurations:

1. **Server set-up:** One Dell PowerEdge R410 with 2 dual quad processors (8 cores - 16 threads) at 2.40GHz, 48G memory and 15k rpm SATA disks.
2. **Cluster set-up:** An infrastructure made of 5 Dell PowerEdge R410s, each of them with 4 dual core processors at 2.27GHz, 48G memory and 15k rpm SATA disks. The network connectivity is standard gigabit ethernet and all the servers are connected to the same network switch.

4sr does not materialize any entailments in the assertion phase, therefore the import throughput we obtained when importing the LUBM datasets is similar to the figures reported by *4store* developers, around 100kT/s for the cluster set-up and 114kT/s for the server set-up[6].

The LUBM benchmark evaluates OWL inference, and therefore there are constructors not supported by *4sr*. We have selected a set of 5 individual triple patterns that cover all the reasoning implemented by *4sr*:

[6] Throughput obtained asserting the data in ntriples, the LUBM datasets had been converted from RDF/XML into ntriples using the rapper tool.

1. **Faculty** {`?s type Faculty`}, Faculty is an intermediate class under the hierarchy rooted by `Person`. To backward entail this query *4sr* will expand `Faculty`'s subclass closure. Moreover, `Faculty` is the `teacherOf`'s predicate domain, therefore, every subject of a triple {`?s teacherOf ?s`} will be part of the solution.
2. **Person** {`?s type Person`}. Similar to the query above, but the closure of `Person` is higher. It contains 16 subclasses, and also Person - or subclasses of it - act as domain and/or range in a number of predicates (`advisor`, `affiliateOf`, `degreeFrom`, `hasAlumnus`, ...). Moreover, these predicates are part of subproperty constructors which makes this query fire all the deductive rules in *4sr*.
3. **Organisation** {`?s type Organisation`}. Binding all the resources type of Organisation will also fire all the deductive rules in *4sr*, but in this case the level predicates that contain Organisation as domain and/or range is lower and also the subclass closure of Person contains fewer elements - just 7.
4. **degreeFrom** {`?s degreeFrom ?o`} such predicates are not instantiated as ground triples and can only be reached by inferring the subproperty closure of it that contains another three predicates - `doctoralDegreeFrom`. `mastersDegreeFrom` and `undergraduateDegreeFrom`.
5. **worksFor** {`?s worksFor ?o`} similar backward chain to be triggered - just subproperty closure.

For the server infrastructure we have measured configurations of 1, 2, 4, 8, 16, and 32 segments. For the cluster infrastructure we measured 4, 8, 16 and 32 - it makes no sense to measure fewer than 4 segments in a cluster made up of four physical nodes.

The queries used in this benchmark have a low selectivity and thus generate large number of solutions; this scenario is suitable for analysing scalability by measuring the number of solutions entailed per second in all the segments for different number of processes (segments) and different size of datasets. The 5 query triple patterns were processed 50 times each, the worst 4 and best 4 measurements were removed from the samples and the time of the worst segment in each measurement was considered, since the binds are processed in parallel and the query processor waits for all to be finished.

Figures 4 and 5 show the results of the benchmark - the Y axis shows the number of solutions and the X axis the number of segments where the data was tested. The benchmark for the server configuration - Figure 4 - shows that the system scales well up to 16 segments, some configurations only up to 8 segments, there is clear degradation for deployments of 32 segments. When the configuration has more segments than CPUs then it is clear that system degrades providing less throughput. In general, the bind operations that require domain and range inference are more expensive than the others - they generate fewer solutions per second. The worst performance was measured for the `Person` bind; this case basically needs to traverse every predicate because the class `Person` happens to be domain and/or range of almost every predicate in the LUBM ontology. The highest throughputs were for `worksFor` and `degreeFrom` binds.

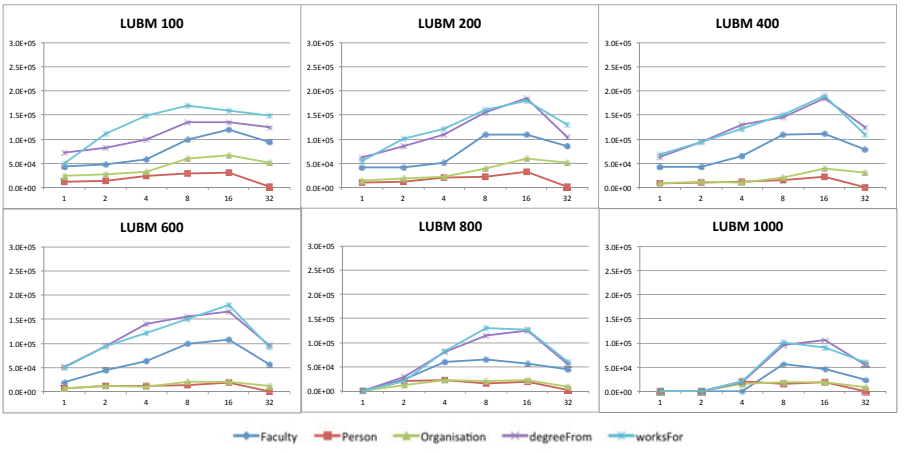

Fig. 4. Server Configuration: Solutions per second per segment

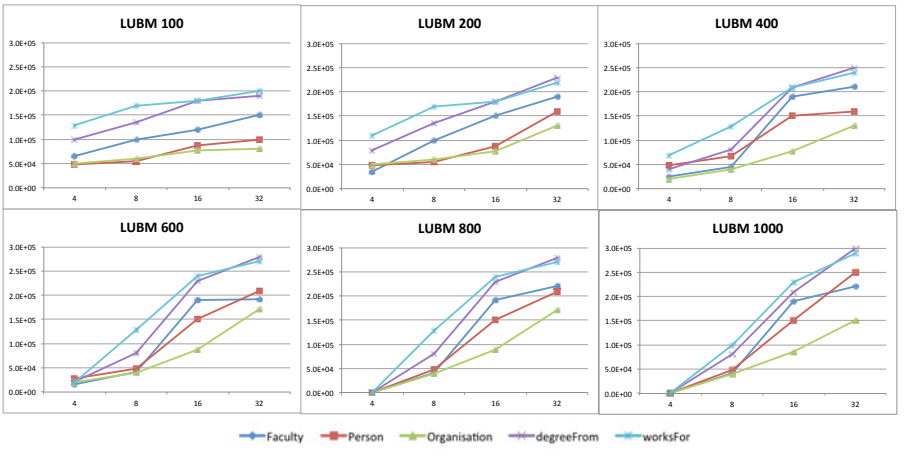

Fig. 5. Cluster Configuration: Solutions per second per segment

For the biggest datasets - LUBM800 and LUBM1000 - the system degraded drastically. For these datasets and 1,2 and 4 segment deployment the system did not respond properly.

The cluster benchmark - Figure - shows better performance. The time needed for transmitting messages over the network gets balanced by the fact that there is a lot better I/O disk throughput. The server configuration has 2 mirrored 15K RPM disks, the same as each of the nodes in the cluster but every node in the cluster can use those disks independently from the other nodes and, the segments collide less on I/O operations.

The performance of the cluster for the biggest datasets - LUBM800 and LUBM1000 - show optimal performance reaching all the binds throughputs between 150K solutions per second and 300K solutions per second. Domain and range inference for `Faculty`, `Organisation` and `Person` show linear scalability and no degradation - unlike in the server configuration. The throughput performance tends to get higher the bigger the dataset is because it generates more solutions without yet reaching the performance limits of the cluster.

In overall, the server configuration sets the scalability limit on the LUBM 400 for the 16-node configuration. For bigger datasets than LUBM400 the server configuration behaves worse. The cluster configuration seems to perform better due to its more distributed nature. It is fair also to mention that, of course, the cluster infrastructure cost is higher than the server and for some applications the performance shown by the server configuration could be good enough.

6 Conclusions and Future Work

In this paper we have presented the design and implementation of *Mininal* RDFS fragment with two novel characteristics decentralisation and backward chaining. We have also disclosed *4sr*'s distributed model and how `subProperty`, `subClass`, `domain`, `range` and `type` semantics can be parallelized by synchronizing a small subset of the triples, namely the ones held in G_{mrdf}. The scalability benchmark showed that the distributed model makes efficient usage of the cluster infrastructure with datasets up to 138M triples. The scalability analysis showed that *4sr* utilises efficiently the cluster infrastructure providing better throughput for bigger datasets.

Since no materialization is processed at the data assertion phase, *4sr* offers a good balance between import throughput and query performance. In that sense, *4sr* will support the development of Semantic Web applications where data can change regularly and RDFS inference is required.

6.1 Future Work

Our plans for future work include the implementation of stronger semantics for named graphs. At the point of writing this paper the research community is discussing how named graphs with attached semantics should behave in a quad store. Our current implementation simply makes G_{mrdf}, G'_{mrdf} and G^*_{mrdf} available to every graph and we delegate the semantics of named graphs to the query engine that will treat entailed solutions as part of G_e.

Acknowledgements

This work was supported by the EnAKTing project funded by the Engineering and Physical Sciences Research Council under contract EP/G008493/1.

References

1. Broekstra, J., Kampman, A., Van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema, pp. 54–68. Springer, Heidelberg (2002)
2. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: Implementing the Semantic Web Recommendations. In: WWW (2004)
3. Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Journal of Web Semantics 3(2-3), 158–182 (2005)
4. Harris, S., Lamb, N., Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: Scalable Semantic Web Knowledge Base Systems - SSWS 2009, pp. 94–109 (2009)
5. Hayes, P., McBride, B.: RDF Semantics, W3C Recommendation (February 10, 2004), http://www.w3.org/TR/rdf-mt/
6. Kaoudi, Z., Miliaraki, I., Koubarakis, M.: RDFS Reasoning and Query Answering on Top of DHTs. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 499–516. Springer, Heidelberg (2008)
7. Kotoulas, S., Oren, E., van Harmelen, F.: Mind the Data Skew: Distributed Inferencing by Speeddating in Elastic Regions. In: Proceedings of the WWW 2010, Raleigh NC, USA (2010)
8. Muñoz, S., Pérez, J., Gutierrez, C.: Simple and Efficient Minimal RDFS. Journal of Web Semantics 7, 220–234 (2009)
9. Salvadores, M., Correndo, G., Harris, S., Gibbins, N., Shadbolt, N.: 4sr - Scalable Decentralized RDFS Backward Chained Reasoning. In: Posters and Demos. International Semantic Web Conference (2010)
10. Salvadores, M., Correndo, G., Omitola, T., Gibbins, N., Harris, S., Shadbolt, N.: 4s-reasoner: RDFS Backward Chained reasoning Support in 4store. In: Web-scale Knowledge Representation, Retrieval, and Reasoning, Web-KR3 (2010)
11. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A Practical OWL-DL Reasoner. Journal of Web Semantics 5(2), 51–53 (2007)
12. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.E.: Owl Reasoning with Webpie: Calculating the Closure of 100 Billion Triples. In: Extended Semantic Web Conference (2010)
13. Urbani, J., Kotoulas, S., Oren, E., van Harmelen, F.: Scalable Distributed Reasoning Using MapReduce. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 634–649. Springer, Heidelberg (2009)
14. Weaver, J.: Redefining the RDFS closure to be decidable. In: W3C Workshop RDF Next Steps, Stanford, Palo Alto, CA, USA (2010), http://www.w3.org/2009/12/rdf-ws/papers/ws16
15. Weaver, J., Hendler, J.A.: Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 682–697. Springer, Heidelberg (2009)