4s-reasoner: RDFS Backward Chained Reasoning Support in 4store

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

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

Abstract

This paper describes the design and implementation of backward chained clustered RDFS reasoning in 4store. The system presented, called "4s-reasoner", adds no overhead to the import phase and yet performs reasonably well at the query phase. We also demonstrate that our solution scales over clusters of commodity servers providing an optimal solution that balances infrastructure cost and performance over tested data sets with up to 500M triples. In addition we have shared our implementation under GNU license and a first release is available to be used by the community.

1. Introduction

4store 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 high scalable quad store and different functionalities such updates, deletes, lock-free and reasoning have started to be implemented in different code branches. This paper describes the initial design and implementation of the RDFS entailment regime in *4store* part of the *rdfs-reasoner* branch [1]. Our implementation of RDFS follows a backward chained (BC henceforth) approach. We considered this type of solution suitable to keep the current data import speed, around 100kT/s for commodity servers. Moreover, a BC approach allows us to easily parametrize SPARQLs to be run with or without the RDFS entailment. Even though, at this point we consider our work in a *alpha* state we have obtained good results for well known benchmarks and datasets containing up to 500M triples.

[1] `http://4sreasoner.ecs.soton.ac.uk` 4s-reasoner documentation and source code Accessed 15/04/2010

The remainder of the paper proceeds as follows: Section 2 describes the motivation and related research in the area, Section 3 introduces basic *4store* notions and explains the modifications undertaken to implement *4s-reasoner*, Section 4 shows the results of initial benchmarks and finally Section 5 analyzes some of results achieved by this work.

2. Motivation and Background

One of the current approaches to provide SPARQL query answering with entailment regimes, as used in Sesame [1], is to forward chain (FC) the knowledge bases (KB) expanding the data before or during the assertion phase. In [7], parallel reasoning was applied to expand the RDFS closure over hundreds of millions of triples and more recent investigations even to one hundred billion triples using the OWL Horst entailment regime [6]. However very little work has been presented on how to make use of such vast amounts of data and how to connect those solutions with SPARQL engines. Furthermore, these type 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.

A different approach called BC reasoning asserts the original data, without expansions, and handles the axioms at the query phase. Historically, BC reasoning has had worse performance than FC at the query phase, however it adds no overhead on data transactions. In the near future, due to the SPARQL/Update specification to be ratified soon, we expect more triple/quad stores to implement and support transactions, which makes our solution more germane at this particular period. Current research on BC reasoning and SPARQL query answering has attempted to implement solutions that distribute the processing on top of Distributed

Hash Tables and using p2p techniques [4]. To date such solutions have not reached the community tools and recent investigations have concluded that due to load balancing issues they cannot scale [5].

4s-reasoner is attempting to make new grounds in BC reasoning exploiting a clustered RDF store and as shown in the following sections our solution scales up to hundreds of millions of triples.

3. Design and Implementation

3.1. 4store basics

4store distributes the data in non-overlapping segments. These segments are identified by an integer and the allocation strategy is a simple *mod* operation such that:

```
segment = rid(subject) mod segments
```

RIDs are 64-bit integers that either represent URIs, Literals or Blank Nodes, and an RID is made up of UMAC-64 hash function. As a quad store 4store represents its RDF together with the model/graph they belong to. The segments contain the following indexes:

- One index per-predicate that contains two radix tries, the keys for these tries are the subject or the object. The tree node data points to the quad.
- One index to store the models. This index is a hash table where the key is the model RID and the elements a list of triples (the list of triples held by a model on a given segment).

The data segments are allocated in *Storage Nodes* and the query engine in a *Processing Node*. The query engine accesses the data remotely via sending TCP/IP messages to Storage Nodes. It also decomposes a SPARQL query into its algebra and for each quad pattern requests a *bind* operation against all the segments[2]. The bind operation returns Q_s which is a set of quads of the form *(m,s,p,o)*, in a given segment *s*, and given multiset *<M, S, P, O>* such that:

$$\{(m,s,p,o) \in Q_s : m \in M \vee M = \emptyset, s \in S \vee S = \emptyset, \\ p \in P \vee P = \emptyset, o \in O \vee O = \emptyset\}$$

The quads returned by each segment are projected to produce a multiset of n-tuples where *n* is the number of quadparts to be extracted. The n-tuples multiset is handled by the query engine that in essence follow a Relational Algebra computation to return the query final resultset.

[2]In 4store there are query optimisations that avoid requesting all the segments and binds are just targeted to where the quads are known to be held (see section 8 [2])

3.2. bind' and 4s-reasoner extensions

In $bind'$ we modify the binding function making it RDFS-aware. Our modifications to the 4store architecture are shown in Figure 1 for a hypothetical 2 storage-node deployment.

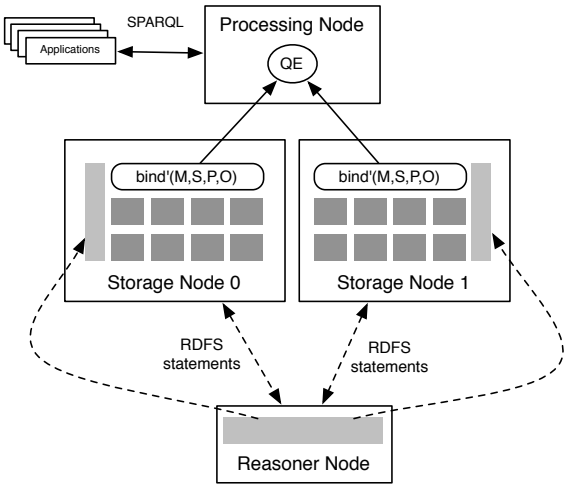

Figure 1. 4s-reasoner architecture

The new components are:

- A new entity called *Reasoner Node*. This entity gathers all RDFS statements from all the storage nodes and keeps a synchronized copy of such information accessible to the *bind'* in all the segments. After every import, update or delete this process extracts the new set of RDFS statements in the KB and sends it to the Storage Nodes. Even for large KBs this synchronization is fast because RDFS statements are a very small proportion of the dataset.
- A new bind function *bind'* that matches the quads not just taking into account the explicit knowledge but also the extensions from the RDFS semantics. *bind'* is depicted in detail in Section 3.3.

3.3. bind' and RDFS Semantics

The original 4store bind operation has been modified so as to follow the RDFS semantics. The current version of *4s-reasoner* implements rule entailments related to *rdfs:subClassOf*, *rdfs:subPropertyOf*, *rdfs:domain* and *rdfs:range*. These semantics include the rules *rdfs2*, *rdfs3*, *rdfs5*, *rdfs7*, *rdfs9*, *rdfs11*, *ext1*, *ext2*, *ext3* and *ext4* (see section 7.3 in [3]). Our solution is an implementation of the RDFS semantics for the selected rules that can decomposed in the phases described in Figure 2.

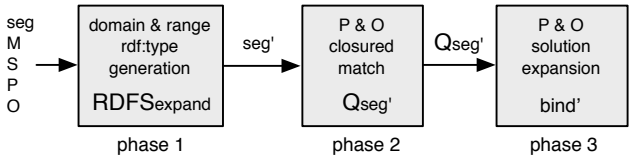

Figure 2. Bind' processing

- $rdfs_{expand}(seg, KB)$: Expands the segment quads to generate the *rdf:type* for *rdfs:range* and *rdfs:domain* (rules *rdfs2* and *rdfs3*) only when *rdf:type* is part of P. The union of $rdfs_{expand}(seg, KB)$ with the initial segment seg produces seg'.

- $Q_{seg'}$: Gets the closured match following rules: *rdfs5*, *rdfs7*, *rdfs9*, *rdfs11*. Since this evaluation is match against seg' the Extensional Entailment Rules: *ext1*, *ext2*, *ext3* and *ext4* are evaluated as well.

- $bind'(Q_{seg'})$: Expands the solutions if either P or O or both are empty. The same rules as in the $Q_{seg'}$ phase are applied at this point.

The following formulas describe the semantics tested on each of the phases:

Phase 1:

$$\{(\_, s, type, d) \in rdfs_{expand}(seg, KB_{rdfs}) : (\exists(\_, s, p, o) \in seg \wedge \exists(\_, p, domain, d) \in KB_{rdfs}) \vee (\exists(\_, o, p, s) \in seg \vee \exists(\_, p, range, d) \in KB_{rdfs})\}$$

In Phase 1 *rdfs2* and *rdfs3* are applied if rdf:type is part of P. KB_{rdfs} represents the RDFS axioms gathered by the Reasoner Node and replicated in all the segments. In the next phase seg' is considered to be: $seg' = seg \cup rdfs_{expand}(seg, KB)$

Phase 2:

$$\{(m, s, p, o) \in Q_{seg'} : m \in M \vee M = \emptyset,\ s \in S \vee S \in \emptyset, p \in closure_p(P) \vee P = \emptyset, (p \in P_{rdfs} \wedge o \in closure_o(O)) \vee o \in O \vee O = \emptyset\}$$

In Phase 2 for the bounded P and O the transitive sub-closure of properties and classes are made part of the matching process. $closure_p(P)$ is the transitive closure of the set of properties P according to *rdfs5*. Analogously, $closure_o(O)$ is the closure of subclasses following rule *rdfs11*. P_{rdfs} is considered to be $\{subClassOf, type, subPropertyOf\}$.

Phase 3:

$$\{(\_, s, p, o) \in bind'(Q_{seg'}, M, S, P, O) : (P \neq \emptyset \wedge O \neq \emptyset \wedge (\_, s, p, o) \in Q_{seg'}) \vee (P = \emptyset \wedge O \neq \emptyset \wedge (\_, s, p', o) \in Q_{seg'} \wedge p \in closure'_p(p')) \vee (P \neq \emptyset \wedge O = \emptyset \wedge (\_, s, p, o') \in Q_{seg'} \wedge o \in closure'_o(o')) \vee (P = \emptyset \wedge O = \emptyset \wedge (\_, s, p', o') \in Q_{seg'} \wedge p \in closure'_p(p') \wedge o \in closure'_o(o'))\}$$

The last step is to expand $Q_{seg'}$ if either P or O are empty. $closure'_p(p')$ are the transitive super properties for a given property p' and $closure'_o(o')$ are the transitive super classes for a given class o'. The $bind'$ expansion described in the formula above expands unbounded P or O according to the super properties or super classes respectively.

3.4. Disabling the RDFS Entailments

Disabling the reasoning is important for some type of queries, for instance when drawing a class hierarchy. One of the benefits of BC reasoning is that is trivial to enable/disable the reasoner. With FC reasoning the implementation of such functionality is complex because adds the overhead of tracking which triples are entailed.

In our implementation reasoning can be disabled with the flag *–no-reasoning*[3] in the *4s-query* tool and in the HTTP SPARQL endpoint.

4. Evaluation and Benchmark

This work is driven by the idea of enabling 4store with RDFS reasoning without degrading the import phase. In that sense we measured import throughput and benchmarked the query phase for different datasets. Our deployment infrastructure is made of 5 Dell PowerEdge R410, each of them with 4 dual core processors at 2.27GHz, 48G memory and 15k rpm disks. The network connectivity is standard gigabit ethernet. These 5 servers are set up in a 4+1 infrastructure with 1 server dedicated for the Processing Node and the Reasoner Node and the other 4 set up as Storage Nodes. We have evaluated KBs of 32 and 64 segments therefore 8 and 16 segments per server respectively. The benchmark presented in this paper measures the performance of *4s-reasoner* for *rdfs:subClassOf* and *rdfs:subPropertyOf* semantics. To test them two different datasets where selected:

- The Barton dataset[4] that contains around 80M triples contains *rdfs:subClassOf* and *rdfs:subPropertyOf* axioms. It is a suitable dataset for testing due to the

[3] `http://4sreasoner.ecs.soton.ac.uk` 4s-reasoner documentation contains samples of how to use the *-no-reasoner* flag.

[4] `http://simile.mit.edu/wiki/Dataset:_Barton` accessed 14/04/2010

combination of such axioms. However, the hierarchies are small. The import throughput for this dataset was 140kT/s.

- The Berlin SPARQL Benchmark for 100M and 500M triples, henceforth *bsb100m* and *bsb500m*. These datasets contain hierarchies of several thousand of nodes which makes them suitable to test the scalability of *rdfs:subClassOf*. The import throughput for this datasets were 110kT/s for *bsb100m* and 92 kT/s for *bsb500m*.

To benchmark the queries we measured the time that the tool *4s-query* takes to execute the query without formatting the SPARQL resultset and printing the solutions. The query templates used against the Berlin datasets are:

```
q1) SELECT * WHERE {
    %ProductInstance% a ?o }
q2) SELECT * WHERE {
    %ProductType% rdfs:subClassOf ?o }
q3) SELECT * WHERE { ?s a %ProductType% .
    ?s berlin:productFeature %Feature% }
q4) SELECT * WHERE {
    ?x a berlin:ProductType1 . }
```

Queries *q1*, *q2* and *q3* were run 128 times for random URIs of *ProductInstance*, *ProductType* and *Feature*. *q4* gets all the product in the datasets over a closure of 2010 and 3948 subclass relationships in *bsb100m* and *bsb500m* respectively. The average response times are shown in Figure 3 and no major measurement deviations where detected.

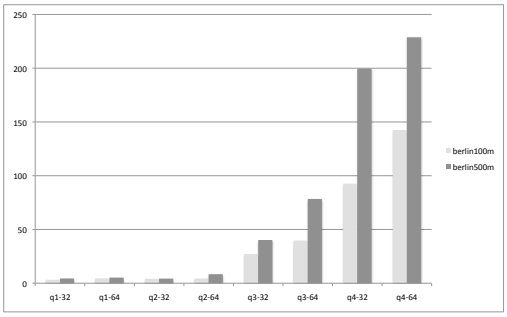

Figure 3. Query Response in milliseconds for the Berlin 100M and 500M datasets

The *rdfs:subPropertyOf* semantics have been tested against the Barton dataset with the following queries:

```
q5) SELECT * WHERE { %Item% ?p ?o . }
q6) SELECT * WHERE {
    %Item% barton:description ?o .}
q7) SELECT * WHERE {
    ?s barton:description ?o . }
```

q5 evaluates the expansions of both superclasses and super-properties. *q6* and *q7* evaluates the match of a sub-property for a given super-property. Analogously to the previous tests these queries were run 128 times for random instances of *Item*. The best time results were obtained for the 32 segment configuration where with 3.52, 1.9 and 280 milliseconds respectively. *q7* returns more than 1 million solutions, that is why the response time is much longer than in *q5* and *q6* where no more than 25 solutions are given.

Throughout the benchmark we identified a performance degradation from the 32 to the 64 segments configuration. This is due to that the optimal 4store configuration is as many segments as there are physical cores.

5. Conclusions

In this paper we presented the design and implementation of *4s-reasoner*. The design is based on the substitution of the original *bind* function for a *bind'* where quads are matched and expanded taking into account the RDFS semantics. Furthermore, we presented a preliminary benchmark with datasets up to 500 millions triples obtaining response times from 1.5 to 200 milliseconds for a 32 segment configuration.

6 Acknowledgements

This work was supported by the EnAKTing project funded by EPRSC under contract EP/G008493/1.

References

[1] J. Broekstra, A. Kampman, and F. V. Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDFS pages 54–68. Springer, 2002.

[2] S. Harris, N. Lamb, and N. Shadbolt. 4store: The Design and Implementation of a Clustered RDF store. In *Scalable Semantic Web Knowledge Base Systems - SSWS2009*, pages (p. 94–109), 2009.

[3] P. Hayes and B. McBride. RDF Semantics, W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-mt/

[4] Z. Kaoudi, I. Miliaraki, and M. Koubarakis. RDFS reasoning and query answering on top of DHTs. In *International Semantic Web Conference*, pages 499–516, 2008.

[5] S. Kotoulas, E. Oren, and F. van Harmelen. Mind the Data Skew: Distributed Inferencing by Speeddating in Elastic Regions. In *Proceedings of the WWW 2010, Raleigh NC, USA*.

[6] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. Bal. OWL Reasoning with Webpie: Calculating the Closure of 100 Billion triples. In *Proceedings of the Seventh European Semantic Web Conference*, LNCS. Springer, 2010.

[7] J. Weaver and J. A. Hendler. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In *International Semantic Web Conference*, pages 682–697, 2009.