The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)

At the 8th International Semantic Web Conference (ISWC2009), Washington DC, USA, October 26, 2009

SSWS 2009 PC Co-chairs' Message

SSWS 2009 was the fifth instance in the sequence of successful Scalable Semantic Web Knowledge Base Systems workshops. This workshop focused on addressing scalability issues with respect to the development and deployment of knowledge base systems on the Semantic Web. Typically, such systems deal with information described in Semantic Web languages like OWL and RDF(S), and provide services such as storing, reasoning, querying and debugging. There are two basic requirements for these systems. First, they have to satisfy the application's semantic requirements by providing sufficient reasoning support. Second, they must scale well in order to be of practical use. Given the sheer size and distributed nature of the Semantic Web, these requirements impose additional challenges beyond those addressed by earlier knowledge base systems. This workshop brought together researchers and practitioners to share their ideas regarding building and evaluating scalable knowledge base systems for the Semantic Web.

This year we received 15 submissions. Each paper was carefully evaluated by two or three workshop Program Committee members. Based on these reviews, we accepted ten papers, seven for full length oral presentation and three for short presentation. The topics of the selected papers span the areas of benchmarking, large scale data stores, optimized representation mechanisms, data integration, and query processing. We sincerely thank the authors for all the submissions and are grateful for the excellent work by the Program Committee members.

October 2009

Achile Fokoue
Yuanbo Guo
Thorsten Liebig

Program Commitee

Achile Fokoue
IBM Watson Research Center, USA

Yuanbo Guo
Microsoft, USA

Thorsten Liebig
Ulm University, Germany

Ian Horrocks
University of Oxford, UK

Kavitha Srinivas
IBM Watson Research Center, USA

Takahira Yamaguchi
Keio University, Japan

Raúl García Castro
Univ. Politecnica de Madrid, Spain

Aditya Kalyanpur
IBM Watson Research Center, USA

Boris Motik
University of Oxford, UK

Oscar Corcho
University of Manchester, UK

Ralf Möller
Hamburg Univ. of Techn., Germany

Marko Luther
DoCoMo Eurolabs Munich, Germany

Andy Seaborne
Hewlett-Packard, UK

Jan Wielemaker
Univ. of Amsterdam, The Netherlands

Volker Haarslev
Condordia University, Canada

Jie Bao
Rensselaer Polytechnic Institute, USA

Additional Reviewers

Sebastian Wandelt
Hamburg Univ. of Techn., Germany

Takeshi Morita
Keio University, Japan

Table of Contents

Semantic Web Reasoning by Swarm Intelligence

Kathrin Dentler, Christophe Guéret, and Stefan Schlobach

Department of Artificial Intelligence, Vrije Universiteit Amsterdam, de Boelelaan 1081a, 1081HV Amsterdam, The Netherlands

Abstract. Semantic Web reasoning systems are confronted with the task to process growing amounts of distributed, dynamic resources. This paper presents a novel way of approaching the challenge by RDF graph traversal, exploiting the advantages of swarm intelligence. The nature-inspired and index-free methodology is realised by self-organising swarms of autonomous, light-weight entities that traverse RDF graphs by following paths, aiming to instantiate pattern-based inference rules. The method is evaluated on the basis of a series of simulation experiments with regard to desirable properties of Semantic Web reasoning, focussing on anytime behaviour, adaptiveness and scalability.

1 Introduction

Motivation It is widely recognised that new *adaptive* approaches towards *robust* and *scalable* reasoning are required to exploit the full value of ever growing amounts of dynamic Semantic Web data.[8] Storing all relevant data on only one machine is unrealistic due to hardware-limitations, which can be overcome by distributed approaches. The proposed framework employs twofold distribution: the reasoning task is distributed on a number of agents, i.e. autonomous micro-reasoning processes that are referred to as beasts in the remainder, and data can be distributed on physically distinct locations. This makes reasoning fully parallelisable and thus scalable whenever beasts do not depend on results of other beasts or data on other locations. In most use-cases, co-ordination between reasoning beasts is required, and this paper explores the application of swarm intelligence to achieve optimised reasoning performance.

A second problem of current reasoning methods that focus on batch-processing where all available information is loaded into and dealt within one central location, is that the provenance of the data is often neglected and privacy-issues are risen. An interesting alternative is local reasoning that supports decentralised publishing as envisioned in [19], allowing users to keep control over their *privacy* and the ownership and dissemination of their information. Another advantage of decentralised reasoning compared to centralised methods is that it has the potential to naturally support reasoning on constantly changing data.

Adaptiveness, robustness and scalability are characteristic properties of swarm intelligence, so that its combination with reasoning can be a promising approach. The aim of this paper is to introduce a swarm-based reasoning method and to provide an initial evaluation of its feasibility and major characteristics. A model

of a decentralised, self-organising system is presented, which allows autonomous, light-weight beasts to traverse RDF graphs and thereby instantiate pattern-based inference rules, in order to calculate the deductive closure of these graphs w.r.t. the semantics of the rules. It will be investigated whether swarm intelligence can contribute to reduce the computational costs that the model implies, and make this new reasoning paradigm a real alternative to current approaches.

Method In order to calculate the RDFS or OWL closure over an RDF graph, a set of entailment rules has to be applied repeatedly to the triples in the graph. These rules consist of a precondition, usually containing one or more triples as arguments, and an action, typically to add a triple to the graph. This process is usually done by indexing all triples and joining the results of separate queries. Swarm-based reasoning is an index-free alternative for reasoning over large distributed dynamic networks of RDF graphs.

The idea is simple: an RDF graph is seen as a network, where each subject and each object is a node and each property an edge. A path is composed of several nodes that are connected by properties, i.e. edges. The beasts, each representing an active reasoning rule, which might be (partially) instantiated, move through the graph by following its paths. Swarms of independent light-weight beasts *travel* from *RDF node to RDF node* and from *location to location*, checking whether they can derive new information according to the information that they find on the way. Whenever a beast traverses a path that matches the conditions of its rule, it locally adds a new derived triple to the graph. Given an added transition capability between (sub-)graphs, it can be shown that the method converges towards closure.

Research questions The price for our approach is redundancy: the beasts have to traverse parts of the graph which would otherwise never be searched. It is obvious that repeated random graph traversal of independent beasts will be highly inefficient. The trade-off that needs to be investigated is thus, whether the overhead can be reduced so that the method offers both adaptive and flexible, as well as sufficiently efficient reasoning. The main research question of this paper is whether *Swarm Intelligence* can help to guide the beasts more efficiently, so that the additional costs are out-balanced by a gain in adaptiveness. More specifically, the following research questions are going to be answered:

1. Does a swarm of beasts that is co-ordinated by stigmergic communication perform better than the same number of independent beasts?
2. Does adaptive behaviour of the population lead to increased reasoning performance?
3. How does the reasoning framework react to a higher number of locations?

Implementation and Experiments To prove the concept, a prototypic system has been implemented based on AgentScape[15]. Each beast is an autonomous reasoning agent, and each distributed graph administered by an agent that is referred to as dataprovider and linked to a number of other dataproviders. Based

on this implementation, the feasibility and major characteristics of the approach are evaluated on the basis of simulation experiments in which beasts calculate the deductive closure of RDF graphs[2] w.r.t. RDFS Semantics[11]. To study the properties of the approach in its purest form, the focus is explicitly restricted to the most simple instantiation of the model of swarm-based reasoning. This means that the answers to the research questions are preliminary.

Findings The experiments described in this paper have two goals: proof of concept and to obtain a better understanding of the intrinsic potential and challenges of the new method. For the former, fully decentralised beasts calculate the semantic closure of a number of distributed RDF datasets. From the latter perspective, the lessons learned are less clear-cut, as the results confirm that tuning a system based on computational intelligence is a highly complex problem. However, the experiments give crucial insights in how to proceed in future work; most importantly on how to improve attract/repulse methods for guiding swarms to interesting locations within the graph.

What to expect from this paper This paper introduces a new swarm-based paradigm for reasoning on the Semantic Web. The main focus is to introduce the general reasoning framework to a wider audience and to study its weakness and potential on the most general level.

We will provide background information and discuss related work in the next section 2, before we define our method in section 3. We present our implementation in section 4 and some initial experiments in section 5, before we discuss some ideas for future research and conclude in section 6.

2 Background and Related Work

In this section, we provide a brief overview of RDFS reasoning and Swarm Intelligence and discuss existing approaches towards distributed reasoning.

The challenge we want to address is to deal with truly decentralised data, that each user keeps locally. Bibliographic data is an example that would ideally be maintained by its authors and directly reasoned over. We will use this scenario throughout the paper to introduce our new Semantic Web reasoning paradigm.

2.1 Semantic Web Reasoning

To simplify the argument and evaluation, we focus on RDF and RDFS (RDF Schema), the two most widely used Semantic Web languages.[1]

Listing 1.1 shows two simple RDF graphs in Turtle notation about two publications `cg:ISWC08` and `fvh:SWP` of members of our Department, maintained

[1] It is relatively straightforward to extend our framework to other rule-based frameworks, such as OWL-Horst reasoning, which is currently the most widely implemented form of Semantic Web reasoning.

separately by respective authors and linked to public ontologies `pub` and `people` about publications and people[2], and reasoned and queried over directly.

```
cg:ISWC08
  pub:title "Anytime Query Answering in RDF through Evolutionary Algorithms" ;
  pub:publishedAs pub:InProceedings ;
  pub:author people:Gueret ;
  pub:author people:Oren ;
  pub:author people:Schlobach ;
  pub:cites fvh:SWP .

fvh:SWP
  pub:title "Semantic Web Primer" ;
  pub:publishedAs pub:Book ;
  pub:author people:Antoniou ;
  pub:author people:vanHarmelen .
```

Listing 1.1. Two RDF graphs about publications

These two graphs describe two publications `cg:ISWC08` and `fvh:SWP` by different sets of authors and are physically distributed over the network. The statements contained in the graphs are extended with schema-information as shown in listing 1.2 and defined in the two respective ontologies, for example with the information that `pub:InProceedings` are `pub:Publications`, `people:Persons` are `people:Agents`, or that `pub:author` has the range `people:Person`.

```
pub:InProceedings rdfs:subClassOf pub:Publication
people:Person rdfs:subClassOf people:Agent
pub:author rdfs:range people:Person
```

Listing 1.2. Some RDFS statements

Given the standard RDFS semantics, one can derive that `cg:ISWC08` is a publication, and that authors are also instances of class `people:Person`, and thus `people:Agent`. The formal semantics of RDFS and OWL enable the automation of such reasoning. The task addressed in the experiments is to calculate the RDF(S) deductive closure, i.e. all possible triples that follow implicitly from the RDF(S) semantics[11]. Table 1 lists some examples of these entailment rules, where the second column contains the condition for a rule to be applied, and the third column the action that is to be performed.[3]

2.2 Swarm Intelligence

As our alternative reasoning method is swarm-based, let us give a short high-level introduction to the field of Swarm Intelligence. Inspired by the collective behaviour of flocks of birds, schools of fish or social insects such as ants or bees, Swarm Intelligence investigates self-optimising, complex and highly structured systems. Members of a swarm perform tasks in co-operation that go beyond the

[2] In the experiments, SWRC and FOAF are used, but to simplify the presentation of the example, ontology names are generic.

[3] The rules follow the conventions: p denotes a predicate, i.e. a URI reference, s a subject, i.e. a URI reference or a blank node, and o refers to an object (which might also be a subject for an ongoing triple), i.e. a URI reference, a blank node or a literal.

Rule	If graph contains	Then add
rdfs2	p rdfs:domain o_1 . and s p o_2 .	s rdf:type o_1 .
rdfs3	p rdfs:range o . and s_1 p s_2 .	s_2 rdf:type o .
rdfs4a	s p o .	s rdf:type rdfs:Resource .
rdfs7	p_1 rdfs:subPropertyOf p_2 . and s p_1 o .	s p_2 o .
rdfs9	s_1 rdfs:subClassOf o . and s_2 rdf:type s_1 .	s_2 rdf:type o .

Table 1. RDFS entailment rules

capabilities of single individuals, which function by basic stimulus → response decision rules. Swarms are characterised by a number of properties, most importantly *lack of central control*, *enormous sizes*, *locality* and *simplicity*. Those properties result in advantageous characteristics of swarms, such as adaptiveness, flexibility, robustness, scalability, decentralisation, parallelism and intelligent system behaviour. These are also desirable in distributed applications, making swarms an attractive model for bottom-up reverse engineering.

Formicidae Ant (Formicidae) colonies appear as super-organisms because cooperating individuals with tiny and short-lived minds operate as a unified entity. Large colonies are efficient due to the self-organisation and functional specialisation of their members. Another characteristic property of ant colonies is their ability to find shortest paths by indirect communication based on chemical pheromones that they drop in the environment. The pheromones act as a shared extended memory and enable the co-ordination of co-operating insects. This mechanism is known as stigmergy[4].

2.3 Related Work

Scalability issues that are risen by the growing amount of Semantic Web data are addressed by projects such as the Large Knowledge Collider[9], a platform for massive distributed incomplete reasoning systems. Calculating the deductive closure with respect to RDFS entailment rules is a standard problem on the Semantic Web and has been addressed extensively. Relevant is the recent work on distributed reasoning. The most prominent paradigms are based on various techniques to distribute data [14,12,1,6] towards several standard reasoners and to combine the results. This is different to the swarm-based methodology, where reasoning is distributed over data that remains local at distributed hosts, so that only small bits of information are moved in the network. Probably closest to this methodology is the distributed resolution approach proposed in [16], but it requires more data to be exchanged than our light-weight swarm-approach.

Biologically inspired forms of reasoning are an emerging research area that aims at modelling systems with intelligent properties as observed in nature. Semantic Web reasoning by Swarm Intelligence has been suggested in [3], that proposes the realisation of "knowledge in the cloud" by the combination of modern "data in the cloud" approaches and Triplespace Computing. Triplespace

Computing[7] is a communication and co-ordination paradigm that combines Web Service technologies and semantic tuplespaces, i.e. associative memories as embodied in Linda[10]. This combination allows for the persistent publication of knowledge and the co-ordination of services which use that knowledge. The envisioned system includes support for collaboration, self-organisation and semantic data that can be reasoned over by a swarm which consists of micro-reasoning individuals that are able to move in and manipulate their environment. The authors suggest to divide the reasoning task among co-operating members of a semantic swarm. Limited reasoning capabilities of individuals and the according reduction of required schema information result in the optimisation of the reasoning task for large-scale knowledge clouds.

3 Semantic Web Reasoning as Graph Traversal

In this section, we introduce our new reasoning methodology, which is based on autonomous beasts that perform reasoning by graph traversal. Given a possibly distributed RDF graph and corresponding schemata, beasts expand the graph by applying basic inference rules on the paths they visit.

3.1 Reasoning as Graph Traversal

When a beast reaches a node, it chooses an ongoing path that starts at the current node. This decision can be taken based on pheromones that have been dropped by previous beasts, or based on the elements of the triples, preferring triples which correspond to the pattern of the inference rule (best-first or hit-detection). If the chosen triple matches the beast's pattern, the rule will be fired and the new inferred triple added to the graph.

Given three infinite sets I, B and L respectively called URI references, blank nodes and literals, an *RDF triple* (s, p, o) is an element of $(I \cup B) \times I \times (I \cup B \cup L)$. Here, s is called the subject, p the predicate, and o the object of the triple. An *RDF graph* G (or graph or dataset) is then a set of RDF triples.

Definition 1 (Reasoning as Graph Traversal). *Let G be an RDF graph, N_G the set of all nodes in G and $M_G = (L \cup I) \times \ldots \times (L \cup I)$ the memory that each beast is associated with.* RDF graph traversal reasoning *is defined as a triple (G, B_G, M_G), where each $b \in B_G$ is a transition function, referring to a* (reasoning) beast *$rb : M_G \times G \times N_G \rightarrow M_G \times G \times N_G$ that takes as input a triple of the graph, moves to an adjacent node in the graph, and depending on its memory, possibly adds a new RDF triple to the graph.*

RDFS reasoning can naturally be decomposed by distributing complementary entailment rules on the members of the swarm, so that each individual is responsible for the application of only one rule. Therefore, we introduce different types of beasts, one type per RDF(S) entailment rule containing schema information. If a concrete schema triple of a certain pattern is found, a corresponding reasoning beast is generated. Regarding for example the rule rdfs3

from Table 1, which deals with range restrictions: whenever in the schema an axiom `p rdfs:range x` is encountered, which denotes that every resource in the range of the property `p` must be of type `x`, a beast responsible for this bit of schema information is initialised as a function $rb3$ that is associated to memory $\{p, x\}$. Table 2 lists the RDFS entailment rules, omitting blank node closure rules rdf2 and rdfs1, with the pattern that is to be recognised in column 2 and the reasoning beast with its memory requirement in column 3.

Entailment rule	**Pattern of schema triple**	**Beast: memory**
rdfs2	p rdfs:domain x .	rb2: p x
rdfs3	p rdfs:range x .	rb3: p x
rdfs5	p_1 rdfs:subPropertyOf p . p rdfs:subPropertyOf p_2 .	rb7: p_1 p_2
rdfs6	p rdf:type rdf:Property .	rb7: p p
rdfs7	p_1 rdfs:subPropertyOf p_2 .	rb7: p_1 p_2
rdfs8	c rdf:type rdfs:Class .	rb9: c rdfs:Resource
rdfs9	c_1 rdfs:subClassOf c_2 .	rb9: c_1 c_2
rdfs10	c rdf:type rdfs:Class .	rb9: c c
rdfs11	c_1 rdfs:subClassOf c . c rdfs:subClassOf c_2 .	rb9: c_1 c_2
rdfs12	p rdf:type rdfs:ContainerMembershipProperty .	rb7: p rdfs:member
rdfs13	s rdf:type rdfs:Datatype .	rb9: s rdfs:Literal

Table 2. Schema-based instantiation of reasoning beasts

Table 3 shows the beasts needed for RDFS reasoning with their pattern-based inference rules. Underlined elements correspond to the memory. Reasoning beasts $rdf1b$, $rb4a$ and $rb4b$ are schema-independent and do not require any memory. They infer for each predicate that it is of `rdf:type rdf:Property` and for each subject and object that it is of `rdf:type rdfs:Resource`. Reasoning beasts $rb2$ and $rb3$ apply the semantics of `rdfs:domain` and `rdfs:range`, while beasts $rb7$ and $rb9$ generate the inferences of `rdfs:subPropertyOf` and `rdfs:subClassOf`. From now on, they are being referred to as domain-beast, range-beast, subproperty-beast and subclass-beast.

Beast : memory	**If pattern**	**Then add**
rdf1b : $\emptyset$	s p o .	p rdf:type rdf:Property .
rb2 : {p,x}	s p o .	s rdf:type x .
rb3 : {p,x}	s p o .	o rdf:type x .
rb4a : $\emptyset$	s p o .	s rdf:type rdfs:Resource .
rb4b : $\emptyset$	s p o .	o rdf:type rdfs:Resource .
rb7 : {p_1,p_2}	s p_1 o .	s p_2 o .
rb9 : {c_1,c_2}	s rdf:type c_1 .	s rdf:type c_2 .

Table 3. Inference patterns of reasoning beasts

Let us assume that a range-beast arrives at node o from a node s via an edge (s, p, o). Because p corresponds to its remembered property, it writes the triple `(o, rdf:type, x)` to the graph. As it can walk both directions of the directed

graph, it will output the same triple when it arrives at node s from node o via edge (o, p, s). Finally, it moves on to a new destination n via a property p_i, where $(o, p_i, n) \in G$.

There are many design decisions in the creation of beasts: in our prototypical implementation, all schema triples are retrieved and pre-processed, calculating the subclass and subproperty closure, before the beasts are created. For this reason, there is no one-to-one mapping between RDFS rules and beasts. For example, the transitivity of `rdfs:subClassOf` is first exhausted, so that rule rdfs11 is encoded with several beasts of type $rb9$. The more generic approach would have been to introduce a special transitivity-beast, which also writes new subclass triples to the graph and then to have beasts picking up the schema-information that they are to apply. This option is to be investigated in future work. An advantage of the proposed mechanism is that all reasoning patterns require only one matching triple for the rule to be fired, so that inferences do not depend on remote data.

Example Let us consider again the two RDF graphs from our previous example. Fig. 1 shows the RDF graph for the first publication. Dashed arrows denote implicit links derived by reasoning.

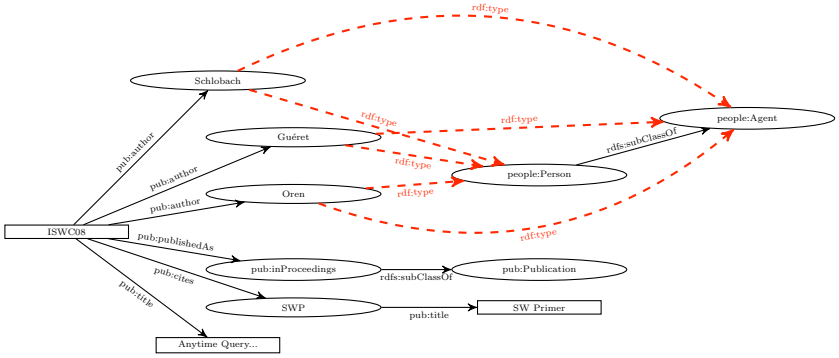


Fig. 1. An exemplary RDF graph

For the three schema axioms of the previous example, beasts are created. For the range-triple `pub:author rdfs:range people:Person`, a range-beast $rb3_1$ is created with memory `pub:author` and `people:Person`. For the subclass triple `people:Person rdfs:subClassOf people:Agent`, a beast $rb9_1$ is created which is instantiated with the memory bits `people:Person` and `people:Agent` (the other subclass-beast is generated accordingly). In this example, only one beast per instantiated type is created, in practise there will be more. The beasts are randomly distributed over the graph, say $rb3_1$ to node `fvh:SWP`, and similarly the other two beasts. Beast $rb3_1$ has now two options to walk. Moving to "SW Primer" will lead it to a cul-de-sac, which means it needs to walk back via

`cg:ISWC08` towards, e.g. `person:Oren`. At node `person:Oren`, the walked path is `cg:ISWC08 pub:author person:Oren` which means $rb3_1$'s pattern matches the walked triple, and it will add a triple `person:Oren rdf:type people:Person` to the graph. When, after walking other parts of the graph, the subclass beast $rb9_1$ chooses to follow the new `rdf:type` link from `person:Oren` to `people:Person`, it finds its memory condition matched, and will add the triple `person:Oren rdf:type people:Agent` to the graph, and so forth. This example highlights the obvious challenges faced by the approach, most importantly, that unnecessary paths need to be investigated and that the order of visiting beasts is important ($rb3_1$ had to be at `person:Oren` first, before $rb9_1$ could find anything).

Completeness The closure C^* over a dataset G contains all triples that follow from the RDF(S) semantics. In our framework, entailment rules are instantiated by the schemata and embodied by beasts. Let $b_1, \dots b_n$ be a swarm with at least one individual per type. The complete closure C^* is derived when the union of the beast-outputs $b_1(c_1) \cup \dots \cup b_n(c_n) \equiv C^*$.

Proposition 1 (Completeness). *Reasoning as graph traversal converges towards completeness.*

Proof. Sketch: To prove that the method converges towards completeness, it has to be shown that all elements of C^* are inferred eventually, i.e. that each beast b_m infers the complete closure $b_m(c^*_m)$ of the rule it incorporates. Given the beast-function as defined above, a beast infers c^*_m when it visits all triples of the graph that match its inference-pattern. This can be achieved by complete graph traversal, which is trivially possible and can be performed according to different strategies, such as random walk, breadth- or depth- first. It has to be performed repeatedly, as other beasts can add relevant triples. C^* is reached when the swarm performed a complete graph traversal without adding a new inference to the graph. Given random jumps to other nodes within the graph, which also prevents beasts from getting stuck in local maxima, the same holds for unconnected (sub-)graphs. When a swarm consists of s members $b^1_m, \dots b^s_m$ per type, the individuals of one type can infer $b_m(c^*_m)$ collectively.

The proposed method is sound but redundant, as two beasts can derive the same inference by the application of different rules and superfluous schema information can lead to inferences that are already present in the data. Beasts produce new inferences gradually, and as those are added to the corresponding graph and not deleted, the degree of completeness increases monotonically over time, so that our methodology shows typical anytime behaviour.

Jumping distributed graphs For the time being, our examples were not distributed. Our framework is more general as beasts can easily move to other locations in the network to apply their inference rules there. Challenging is the case when reasoning requires statements of two unconnected (sub-)graphs that are physically distributed over a network (which is not the case for the proposed

RDFS reasoning method but would be for distributed OWL-Horst[17] reasoning). In our current implementation, this case is covered by a simple routing strategy using Bloom filters [1].

Movement Control The reasoning beasts operate on an energy metaphor: moving in the graph costs energy and finding new inferences, which can be interpreted as food, is rewarding. This is modelled via a happiness-function. When the beasts are created, they have an initial happiness-value. Then, at each step from RDF node to RDF node, one happiness-point is subtracted. When a beast receives a reward due to an inference that is new (not-new inferences are worthless), its happiness increases. With this simple mechanism, a beast adapts to its environment: while it is successful and thus happy, it will probably find even more applications of its rule, whereas for a beast that did not find a new triple for a long time, it might be little reasonable to continue the search. When a beast is unhappy or unsuccessful (i.e. it did not find anything new for a given number of node-hops), it can change its rule instantiation, its type, the location or die.

3.2 Distributed Reasoning by Beasts in a Swarm

An advantageous property of populations that are organised in swarms is their ability to find shortest paths to local areas of points of interest. Inspired by ant colonies that lay pheromone-paths to food sources, beasts choose ongoing paths based on pheromones. The environment provides beasts with the information who has been at their current location before, which can be utilised by the swarm to act as a whole, to avoid loops and to spread out evenly, which is useful because the swarm has no gain if members of the same rule-instantiation traverse the same path more than once.

When a beast reaches a node and chooses the next ongoing edge, it parses its options, and while no application of its inference-pattern is found, which would cause it to choose the corresponding path and fire its rule, it applies a pheromone-based heuristic. It categorizes the options into sets of triples: the ones which are promising because no individual of its rule-instantiation has walked them before and the ones that already have been visited. If promising options are available, one of them is chosen at random, otherwise the ongoing path is chosen probabilistically, preferring less visited triples. Only pheromones of beasts with the same rule-instantiation are considered, other possibilities, such as preferring pheromones of other beasts to follow them, are subject to further research. Let τ_j denote the pheromones on a path j and n the number of paths. The probability p_i to choose path i is determined by equation 1. It is between 0 and 1 and higher for paths that have been walked less in the past. This formula has been inspired by Ant Colony Optimization[5], where the probability to choose a path i is τ_i, i.e. the pheromones on that path, divided by the sum of the pheromones on all options.

$$p_i = \frac{\frac{\sum_{j=0}^{n} \tau_j - \tau_i}{n-1}}{\sum_{j=0}^{n} \tau_j}. \quad (1)$$

4 Implementation

The implementation of our model is based on AgentScape [15], a middleware layer that supports large-scale agent systems. Relevant concepts are locations, where agents can reside, and agents as active entities that are defined according to the weak notion of agency [18], which includes autonomy, social ability, reactiveness and pro-activeness. Agents can communicate by messages and migrate from one location to another.

Both the environment and the reasoning beasts of our model are implemented as AgentScape agents. The environment consists of a number of agents that hold RDF (sub-)graphs in Jena [13] models and are called dataprovider. We assume that each dataprovider resides on a distinct location and does not migrate. Beasts do migrate and communicate directly with the dataprovider that resides on their current location. Reasoning beasts migrate to the data, perform local calculations and move on to the next location, so that only the code of the beasts and results that can lead to inferences on other locations are moved in the network and not the data. Other set-ups would be possible, such as beasts querying dataproviders from the distance, without being physically on the same location. Beasts communicate indirectly with each other by leaving pheromone-traces[4] in the environment. They operate on a plain N3 text representation.

5 Research Questions and Corresponding Experiments

This section presents a series of experiments. The focus is not on fine-tuning parameters, but on testing basic strategies to generate first answers to the research questions, and to provide a clear baseline for further experiments. The experiments are based on a number of publication.bib files of members of our Department. The files have been converted to RDF, following the FOAF and the SWRC ontologies.

Beasts are instantiated by our beast-creation mechanism as presented above.[5] Ignoring schema-triples that contain blank nodes, 195 beasts are generated from the employed FOAF and SWRC schemata: 11 subproperty-beasts, 87 subclass-beasts, 48 domain-beasts and 49 range-beasts, each of them with a unique rule-instantiation. In all experiments, swarms of 5 beasts per rule-instantiation are traversing the graphs. Each beast starts at a random node at one of the locations.

Each dataset is administered by a dataprovider that is residing on a distinct location, which is named after the corresponding graph. The initial happiness on each dataset and the maximum number of requests that do not lead to any new

[4] Pheromones are stored as reified triples.

[5] For the sake of simplicity, the experiments are restricted to RDFS simple reasoning. This means that RDFS entailment rules which have only one single schema-independent triple-pattern in the precedent (rdf1, rdfs4a and rdfs4b, rdfs6, rdfs8, rdfs10, rdfs12 and rdfs13) and blank node closure rules are deliberately omitted. These inferences are trivial and can be drawn after the RDFS simple closure has been derived, in case they are required.

inference, is set to 100 and the maximum number of migrations between locations and their corresponding dataproviders is set to 10. A beast's happiness increases by 100 when it inferred a triple that was new indeed. In all experiments, beasts ideally die when the closure is deduced. Then, they cannot find new inferences any more and eventually reach their maximum unhappiness after a number of unsuccessful node- and location-hops.

To evaluate the obtained results, the inferences of the beasts are compared to the output of Jena's RDFS simple reasoner.[6] All graphs show how the degree of completeness (i.e. the percentage of found inferences) per dataset is rising in relation to the number of sent messages. A message is a request for ongoing options from a beast to the dataprovider.

5.1 Baseline: random walk

To generate a baseline for the comparison of different strategies, the beasts in the first experiment choose between their options randomly, even if one of the options contains a triple that leads to an inference.

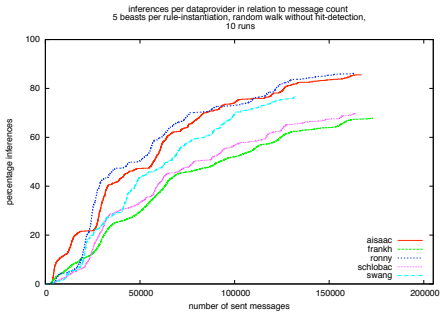


Fig. 2. Random walk without hit-detection

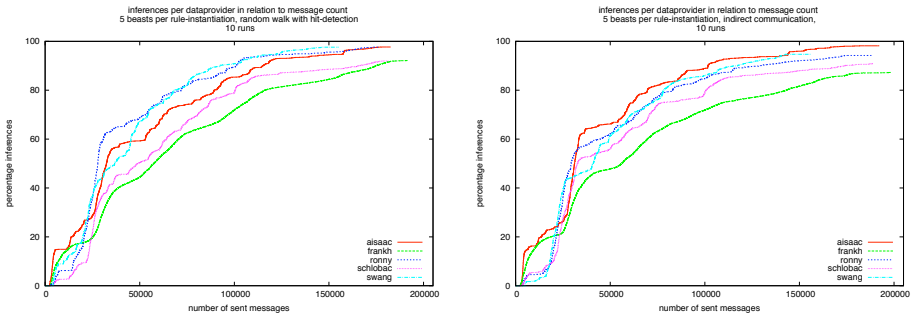
Fig. 2 visualizes the percentage of found inferences over time for each of the datasets. At the end of the experiment, 75.59% of the complete closure have been reached. The results demonstrate typical anytime behaviour: for longer computation times, more inferences are generated.

5.2 Does a swarm of beasts that employs stigmergic communication outperform the same number of independent beasts?

To investigate the question whether stigmergic communication accelerates convergence, a random walk strategy with hit-detection (so that the beasts prefer

[6] In contrast to the beasts, Jena did not generate any inference stating that a resource is of type Literal, so those triples have been added to the output of Jena.

triples that match their reasoning pattern) is compared to a strategy that employs indirect communication by repulsing pheromones, as described in section 3.2. Fig. 3(a) and Fig. 3(b) visualize the results.



(a) Random walk with hit-detection (b) Indirect communication

Fig. 3. Random walk with hit-detection compared to stigmergic communication

A first result is that compared to the random walk without hit-detection, both methods have a positive impact on the reasoning performance of the beasts. Secondly, in contrast to the hypothesis that stigmergic communication would lead to a faster convergence, the graphs are almost alike in the number of sent messages and also in their convergence-behaviour and reached percentage of the closure (94.52% for the random walk and 92.85% for the stigmergic communication).

In the start-phase of the experiments, new inferences are found easily, the difficult task is to detect the last remaining inferences. The assumption was that repulsing pheromones would guide the beasts to the unvisited sub-graphs where the remaining inferences are found. These experiments provide interesting insights into how to proceed in tuning the swarm behaviour: especially by the employment of attracting pheromones to lead fellow beasts to regions with food.

5.3 Does adaptive behaviour of the population increase the reasoning-performance?

To answer the question whether adaptive behaviour of the individuals is beneficial, a dynamic adaption of rule-instantiations in case of unsuccessfulness is tested. Note that this is only one possibility out of many for the beasts to adapt to their environment. Here, each beast is provided with a set of possible rule-instantiations instead of remembering only two arguments. For example, a subclass-beast does not only receive one subclass and one superclass to instantiate its rule, but several subclasses with the same superclass. The beasts switch their rule-instantiation to another random instantiation after each 50^{th} unsuccessful message-exchange. Fig. 4 shows the results of the adaptive beasts:

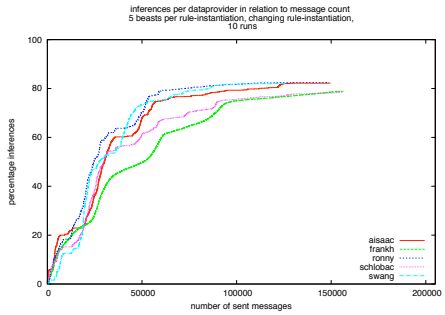


Fig. 4. Beasts that change their rule instantiation

The resulting graph shows clearly that the intuition that dynamic adaptions should lead to increased reasoning-performance did not apply to this experiment. On average, the swarms found 80.67% of the complete closure. In the future, more intelligent approaches have to be investigated, replacing e.g. the randomness in the changes, so that beasts change into successful rule-instantiations. Care has to be taken that not all beasts change to the most successful instantiations, because inferences that occur seldom also need to be found.

5.4 How does the reasoning framework react to a higher number of locations?

An important question regarding the potential scalability of our model is to study what happens when more datasets are added, so we compared the set-up with 5 datasets as shown in figure 3(b) to the same set-up with 10 datasets. We employed the same number of beasts as in the previous experiments.

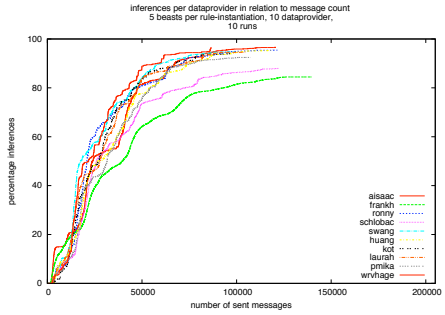


Fig. 5. 10 datasets

Fig. 5 demonstrates that the percentage of the complete closure (93.2%) per dataset is nearly as high as for a set-up with 5 datasets. This is because beasts

that can infer more new triples are increasingly happy and thus can send more messages to the dataproviders. The figure clearly indicates that more data on distributed locations lead to more activity, resulting in more inferences. This is, although not unexpected, a nice result.

6 Conclusion

We have presented a radically new method for Semantic Web reasoning based on Swarm Intelligence. The major feature of this idea is that many light-weight autonomous agents collectively calculate the semantic closure of RDF(S) graphs by traversing the graphs, and apply reasoning rules to the nodes they are currently located on. The advantage of this approach is its adaptiveness and its capability to deal with distributed data from dynamic sources. Such a reasoning procedure seems ideal for the Semantic Web and might help in setting up a decentralised publishing model that allows users to keep control over their personal data.

A metaphor for the way that the proposed paradigm envisages future Semantic Web reasoning is the *eternal adaptive anthill*, the Web of Data as decentralised accessible graphs, which are constantly traversed and updated by micro-reasoning beasts. Based on this vision it can be claimed that swarm-based reasoning is in principle more adaptive and robust than other Semantic Web reasoning approaches, as recurrently revisiting beasts can more easily deal with added (and even deleted) information than index-based approaches.

Our experiments show a proof of concept: over an (admittedly small) decentralised environment of our departmental publications we show that this idea works in principle, which gives us motivation to continue to improve the current framework in future research. Furthermore, first experiments have given clear guidelines for where more research is needed: first, the implementation framework based on AgentScape and Jena might not be ideal, as the dataprovider might be too central to the current implementation and thus become a bottleneck. However, through distribution of the reasoning, scaling is in principle straightforward. As usual, scaling comes at a price, in our case that the distribution will make it difficult for the beasts to find triples they can reason on. Initial experiments with Swarm Intelligence indicate that this might be a way forward, although it is well known that tuning such highly complex systems is very difficult (as our experiments confirm).

Ideas for future work are abundant: improving the implementation of the general model, routing strategies, security issues, dealing with added and deleted statements, trust etc. All these issues can relatively elegantly be represented in our framework, and some solutions are straightforward. Most interesting at the current stage will be to investigate more swarm strategies, such as scout models to guide beasts of similar types towards areas of interest, cloning successful beasts and much more. This paper is a first step.

References

1. M. Cai and M. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *Proceedings of the 13th international conference on World Wide Web*, pages 650–657, 2004.
2. J. J. Carroll and G. Klyne. Resource Description Framework (RDF): Concepts and Abstract Syntax. *W3C recommendation*, 2004.
3. D. Cerri, E. Della Valle, D. De Francisco Marcos, F. Giunchiglia, D. Naor, L. Nixon, D. Rebholz-Schuhmann, R. Krummenacher, and E. Simperl. Towards Knowledge in the Cloud. *Proceedings of the OTM Confederated International Workshops and Posters on On the Move to Meaningful Internet Systems: 2008 Workshops: ADI, AWeSoMe, COMBEK, EI2N, IWSSA, MONET, OnToContent+ QSI, ORM, PerSys, RDDS, SEMELS, and SWWS*, pages 986–995, 2008.
4. M. Dorigo, E. Bonabeau, G. Theraulaz, et al. Ant algorithms and stigmergy. *Future Generation Computer Systems*, 16(9):851–871, 2000.
5. M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT press, Cambridge, MA, 2004.
6. Q. Fang, Y. Zhao, G. Yang, and W. Zheng. Scalable Distributed Ontology Reasoning Using DHT-Based Partitioning. In *The Semantic Web*, volume 5367, pages 91–105. Springer, 2008.
7. D. Fensel. Triple-space computing: Semantic Web Services based on persistent publication of information. *LNCS*, 3283:43–53, 2004.
8. D. Fensel and F. van Harmelen. Unifying Reasoning and Search to Web Scale. *IEEE Internet Computing*, 11(2):96–95, 2007.
9. D. Fensel, F. van Harmelen, Andersson, and et al. Towards LarKC: a platform for web-scale reasoning. In *Proceedings of the International Conference on Semantic Computing*, pages 524–529, 2008.
10. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.
11. P. Hayes and B. McBride. RDF semantics. *W3C recommendation*, 2004.
12. Z. Kaoudi, I. Miliaraki, and M. Koubarakis. RDFS Reasoning and Query Answering on Top of DHTs. In *LNCS*, volume 5318, pages 499–516, 2008.
13. B. McBride. Jena: Implementing the rdf model and syntax specification. In *Proc. of the 2001 Semantic Web Workshop*, 2001.
14. E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, A. ten Teije, , and F. van Harmelen. Marvin: A platform for large-scale analysis of Semantic Web data. In *Proceedings of the International Web Science conference*, 2009.
15. B. J. Overeinder and F. M. T. Brazier. Scalable Middleware Environment for Agent-Based Internet Applications. *LNCS*, 3732:675–679, 2006.
16. A. Schlicht and H. Stuckenschmidt. Distributed Resolution for ALC. In *Description Logics*, 2008.
17. H. J. ter Horst. Combining RDF and part of OWL with rules: Semantics, decidability, complexity. In *Proc. of ISWC*, pages 6–10. Springer, 2005.
18. M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.
19. C. A. Yeung, I. Liccardi, K. Lu, O. Seneviratne, and T. Berners-Lee. Decentralization: The Future of Online Social Networking. Available at: `http://www.w3.org/2008/09/msnws/papers/decentralization.pdf`, 2008.

Efficient Linked-List RDF Indexing in Parliament

Dave Kolas, Ian Emmons, and Mike Dean

BBN Technologies, Arlington, VA 22209, USA
{dkolas,iemmons,mdean}@bbn.com

Abstract. As the number and scale of Semantic Web applications in use increases, so does the need to efficiently store and retrieve RDF data. Current published schemes for RDF data management either fail to embrace the schema flexibility inherent in RDF or make restrictive assumptions about application usage models. This paper describes a storage and indexing scheme based on linked lists and memory-mapped files, and presents theoretical and empirical analysis of its strengths and weaknesses versus other techniques. This scheme is currently used in Parliament (formerly DAML DB), a triple store with rule support that has recently been released as open source.

1 Introduction

As the number and scale of Semantic Web applications in use increases, so does the need to efficiently store and retrieve RDF [1] data. A wide variety of RDF and OWL [2] applications are currently being developed, and each application's scenario may demand prioritization of one performance metric or another. Current published schemes for RDF data management either fail to embrace the schema flexibility inherent in RDF or make restrictive assumptions about application usage models.

Despite the fact that RDF's graph-based data model is inherently different than relational data models, many published schemes for RDF data storage involve reductions to a traditional RDBMS [3–7]. This results in the deficiencies of RDBMS's (inflexible schemas, inability to efficiently query variable predicates) being propagated to RDF storage; arguably, avoiding these deficiencies is one of the major reasons for adopting an RDF data model. Other published approaches eschew the mapping to an RDBMS, but suffer either inadequate load or query performance for many applications. In this paper, we argue that the storage approach in Parliament provides excellent load and query performance with low space consumption and avoids the pitfalls of many other specialized RDF storage systems.

Parliament [8] (formerly DAML-DB [9]) is a triple store developed by BBN that has been in use since 2001. During that time, Parliament has been used for a number of applications from basic research to production. We have found that it offers an excellent tradeoff between load and query performance, and compares favorably to commercial RDF data management systems [10].

Recently, BBN has decided to release Parliament as an open source project. Parliament provides the underlying storage mechanism, while using Jena [11] or Sesame [12] as an external API. This paper explains in detail the underlying index structure of Parliament, and compares it to other published approaches. Our hope is that open-sourced Parliament will provide a fast storage alternative for RDF applications, create a platform upon which storage mechanism and query optimizer research can be built, and generally advance the state of the art in RDF data management.

The remainder of this paper is structured as follows. Section 2 addresses related work. Section 3 describes the index structure within Parliament. Section 4 explains how the operations on the structure are performed. Section 5 provides both worst case and average case analysis of the indexing mechanism, and Section 6 provides a small empirical comparison to supplement [10].

2 Related Work

The related work on RDF data management systems falls into two major categories: solutions that involve a mapping to a relational database, and those that do not.

2.1 RDBMS Based Approaches

A large proportion of the previously published approaches involve a mapping of the RDF data model into some form of relational storage. These include triples-table approaches, property tables, and vertical partitioning. There is a strong temptation to use relational systems to store RDF data since such a great amount of research has been done on making relational systems efficient. Moreover, existing RDBMS systems are extremely scalable and robust. Unfortunately, each of the proposed ways of doing this mapping has deficiencies.

The triples-table approach has been employed in 3store [3], and is perhaps the most straightforward mapping of RDF into a relational database system. Each triple given by (s, p, o) is added to one large table of triples with a column for the subject, predicate, and object respectively. Indexes are then added for each of the columns. While this approach is straightforward to implement, it is not particularly efficient, as noted in later work [4, 5, 13, 14, 10]. The primary problem is that queries with multiple triple patterns result in self-joins on this one large table, and are inefficient.

Property tables were introduced later, and allowed multiple triple patterns referencing the same subject to be retrieved without an expensive join. This approach has been used in Jena 2 [4]. A similar approach is used in [6]. In this approach, each database table includes a column for a subject and several fixed properties. The intent is that these properties often appear together on the same subject. While this approach does eliminate many of the expensive self-joins in a triples table, it still has deficiencies leading to limited scalability. Queries with triple patterns that span multiple property tables are still expensive. Depending

on the level of correlation between the properties chosen for a particular property table, the table may be very sparse and thus be less space-efficient than other approaches. Also, it may be complex to determine which sets of properties are best joined within the same property table. Multi-valued properties are problematic in this approach as well. Furthermore, queries with unbound variables in the property position are very inefficient and may require dynamic table creation. In a data model without a fixed schema, it is common to ask for all present properties for a particular subject. In the property table approach, this type of query requires scanning all tables. With property tables, adding new properties also requires adding new tables, a consideration for applications dealing with arbitrary RDF content. It is the flexibility in schema that differentiates RDF from relational approaches, and thus this approach limits the benefit of using RDF.

The vertical partitioning approach suggested in [5] may be viewed as a specialization of the property table approach, where each property table supports exactly one property. This approach has several advantages over the general property table approach. It better supports multi-valued properties, which are common in Semantic Web data, and does not sacrifice the space taken by NULL's in a sparsely populated property table. It also does not require the property-clustering algorithms for the general property tables. However, like the property table approach, it fails to efficiently handle queries with variables in the property position.

2.2 Other Indexing Approaches

The other primary approaches to RDF data storage eliminate the need for a standard RDBMS and focus instead on indexing specific to the RDF data model. This set of approaches tends to better address the query models of the semantic web, but each suffers its own set of weaknesses.

The RDF store YARS [15] uses six B+ tree indices to store RDF quads of a subject, predicate, object, and a "context". In each B+ tree, the key is a concatenation of the subject, predicate, object, and context, each dictionary encoded. This allows fast lookup of all possible triple access patterns. Unlike the RDBMS approaches discussed above, this method does not place any particular preference on the subject, predicate, or object, meaning that queries with variable predicates are no different than those with variable subjects or objects. This structure sacrifices space for query performance, repeating each dictionary encoded triple six times. The design also favors query performance to insertion speed, a tradeoff not necessarily appropriate for all Semantic Web applications. Our approach is more efficient both in insertion time and space usage, as will be demonstrated. Other commercial applications use this method as well [16]. Kowari [14] is designed similarly, but uses a hybrid of AVL and B trees instead of B+ trees for indexing.

The commercial quad store Virtuoso [7] adds a graph g element to a triple, and conceptually stores the quads in a triples table expanded by one column. While technically rooted in a RDBMS, it closely follows the model of YARS [15],

but with fewer indices. The quads are stored in two covering indices, g, s, p, o and o, g, p, s, where the IRI's are dictionary encoded. Several further optimizations are added, including bitmap indexing and inlining of short literal values. Thus this approach, like YARS, avoids the pitfalls of other RDBMS based work, including efficient variable-predicate queries. The pattern of fewer indices tips the balance slightly towards insertion performance from query performance, but still favors query performance.

Hexastore [13], one of the most recently published approaches, takes a similar approach to YARS. While it also uses the dictionary encoding of resources, it uses a series of sorted pointer lists instead of B+ trees of concatenated keys. Again, this better supports the usage pattern of Semantic Web applications and does not force them into a RDBMS query model. Hexastore not only provides efficient single triple pattern lookups as in YARS, but also allows fast merge-joins for any pair of two triple patterns. Again, however, it suffers a five-fold increase in space for storing statements over a dictionary encoded triples table, and favors query performance over insertion times. It is our experience that applications often do require efficient statement insertion, and thus our approach seeks to balance query performance and insertion time. Since this approach was published most recently and compares favorably to previous approaches, we will focus our empirical comparison evaluation on Hexastore.

Other commercial triple stores such as OWLIM [17] have been empirically shown to perform well, but their indexing structure is proprietary and thus no theoretical comparison can be made.

3 Index Structure

This section explains the three parts of the storage structure of Parliament: the resource table, the statement table, and the resource dictionary. This description is simplified for the sake of clarity; it does not discuss using quads instead of triples, optimizations for blank nodes, the rule engine, or some small implementation details. Parliament can be compiled in either 32 or 64 bit modes, and the width of the fields described varies accordingly.

3.1 Resource Table

The Resource Table is a single file of fixed-length records, each of which represents a single resource or literal. The records are sequentially numbered, and this number serves as the ID of the corresponding resource. This allows direct access to a record given its ID via simple array indexing. Each record has eight components:

- Three statement ID fields representing the first statements that contain this resource as a subject, predicate, and object, respectively
- Three count fields containing the number of statements using this resource as a subject, predicate, and object, respectively

- An offset into the string representations file described below, used to retrieve the string representation of the resource
- Bit-field flags encoding various attributes of the resource

The first subject, first predicate, and first object statement identifiers provide pointers into the statement table, which is described below. The subject, predicate, and object counts benefit the find operations and query optimization, discussed in Section 4. For the remainder of the paper, these counts will be referred to as $count(resource, pos)$ for the count of a resource in the given position. The usage of the offset into the string representations file will be explained below.

3.2 Statement Table

The Statement Table is the most important part of Parliament's storage approach. It is similar to the resource table in that it is a single file of fixed-length records, each of which represents a single statement. The records are sequentially numbered, and this number serves as the ID of the corresponding statement. Each record has seven components:

- Three resource ID fields representing the subject, predicate, and object of the statement, respectively
- Three statement ID fields representing the next statements that use the same resource as a subject, predicate, and object, respectively
- Bit-field flags encoding various attributes of the statement

The three resource ID fields allow a statement ID to be translated into the triple of resources that represent that statement. The three next statement pointers allow fast traversal of the statements that share either a subject, predicate, or object, while still storing each statement only once.

Figure 1 shows an example knowledge base consisting of five triples. Each triple row in the statement list table shows its resource identifier as a number and the pointers to the next statements as arrows. Omitted arrows indicate pointers to a special statement identifier, the null statement identifier, which indicates the end of the linked list.

3.3 Resource Dictionary

Like many other triple stores [13, 5, 6, 12], Parliament uses a dictionary encoding for its resources. This dictionary provides a one-to-one, bidirectional mapping between a resource and its resource ID. The first component of this dictionary is the mapping from a resource to its associated identifier. This portion of the dictionary uses Berkeley DB [18] to implement a B-tree whose keys are the resources' string representations and whose values are the corresponding resource ID's. This means that inserts and lookups require logarithmic time.

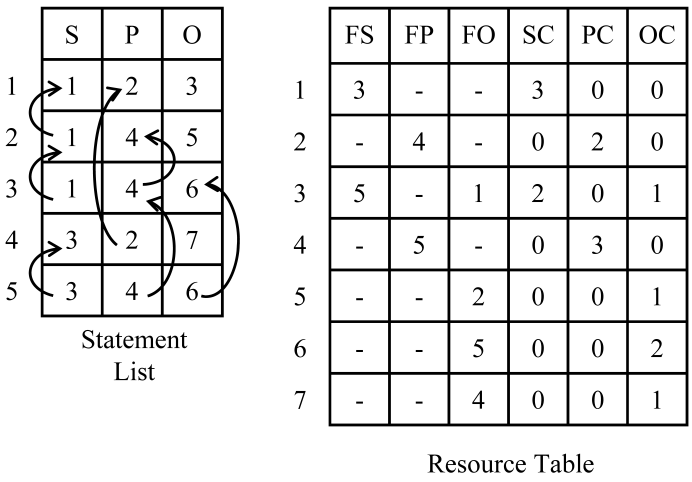


Fig. 1. Example Statement List and Resource Table

The second half of the dictionary is the reverse lookup from a resource ID to a string representation. This is implemented in a memory-mapped file containing sequential, variable-length, and null-terminated string representations of resources. A resource ID is translated into the corresponding string representation by using the resource ID to index into the resource table, retrieving the string representation offset, and using this to index into the string representations file to retrieve the associated string. Thus, looking up a string representation from a resource identifier is a constant time operation.

The current approach stores each string representation twice. Future implementations may eliminate this redundancy.

3.4 Memory-Mapped Files

Three of the four files that comprise a Parliament triple store (the resource table, the statement table, and the string representations file) are stored and accessed via a common operating system facility called "memory mapping". This is independent of the index structure of the store, but is worth mentioning because it confers a significant performance advantage. Most modern operating systems use memory mapping to implement their own demand-paged virtual memory subsystem, and so this mechanism for accessing files tends to be highly optimized and keeps frequently accessed pages in memory.

4 Triple Store Operations

The three fundamental operations that a triple store can perform are query, insertion (assertion), and deletion (retraction). These are discussed below.

4.1 Query

Parliament performs a lookup of a single triple pattern according to the following algorithm:

1. If any of the triple pattern elements are bound, Parliament uses the B-tree to translate the string representations of these resources into resource ID's.
2. If any bound elements are not found in the B-tree, then the query result is the empty set, and the query algorithm terminates.
3. If none of the elements are bound, then the query result is the entire statement list. Parliament enumerates this by iterating across all of the records in the statement table and retrieving the string representations of the elements.
4. If exactly one element is bound, then Parliament looks in the resource table for the resource table the ID of the first statement using that resource in the position the resource appears in the given triple pattern.
5. If two or three elements are bound, then Parliament looks in the resource table for those resource ID's to retrieve $count(resource, pos)$ for each. Parliament selects the resource whose count is smallest, and retrieves from the resource table the ID of the first statement using that resource in the position the resource appears in the given triple pattern.
6. Starting with that statement ID, Parliament traverses the linked list of statement records corresponding to the position of the minimal count resource.
7. If the triple pattern contains exactly one bound resource, then this list of statements is exactly the answer to the query, and again Parliament retrieves the string representations of the elements as it enumerates the list to form the query result.
8. If two or three elements are bound, then as Parliament enumerates the linked list of statements, it checks whether the resources in the positions of the non-minimal count resources are the same as the bindings in the given triple pattern. Whenever a match is found, Parliament retrieves the string representations of the elements and adds that triple to the query result.

Whenever Parliament is enumerating statements, it skips over statements whose "deleted" flag has been set. See Section 4.3 below for details.

Parliament is designed as an embedded triple store and does not include a SPARQL or other query language processor. Such queries are supported by accessing Parliament as a storage model from higher-level frameworks such as Jena or Sesame. Single find operations (as discussed above) are combined together by the higher-level framework, with Parliament-specific extensions for optimization. In particular, when using Parliament with Jena's query processors [11], we have used several different algorithms for query planning and execution, which will be detailed in subsequent publications. The basis of these optimizations is the ability to quickly access the counts of the resources in the given positions.

4.2 Insertion

To insert a triple (s, p, o), Parliament executes the following algorithm:

1. Parliament uses the B-tree to translate the string representations of the three resources into resource ID's.
2. If all three elements are found in the B-tree, then Parliament performs a query for the triple pattern (s, p, o). Note that this is necessarily a fully bound query pattern. If the triple is found, then no insertion is required, and the algorithm terminates.
3. If any elements are not found in the B-tree, then Parliament creates new resources for each of them as follows:
   (a) Parliament appends the string representation of the resource to the end of the string representations file. If the file is not large enough to contain the string, then the file is enlarged first. The offset of the beginning of the string is noted for use in the next step.
   (b) Parliament appends a new record to the end of the resource table. If the file is not large enough to contain the new record, then the file is enlarged first. The number of the record is saved as the new resource ID for use in the steps below, and the offset from the string representations file is written to the appropriate field in this record. The record's counts are initialized to zero, and the first statement ID's are set to null.
   (c) Parliament inserts a new entry into the B-tree. The entry contains the resource's string representation as its key and the new resource ID as its value.
4. Parliament now has three valid resource ID's representing the triple, and knows that the triple is not present in the statement table.
5. Parliament appends a new record to the end of the statement table. If the file is not large enough to contain the new record, then the file is enlarged first. The number of the record is saved as the new statement ID for use in the steps below, and the three resource ID's obtained above are written to the appropriate fields in this record. The record's next statement ID's are all set to null.
6. For each of the three resources, Parliament inserts the new statement record at the head of that resource's linked list for the corresponding triple position as follows:
   (a) The resource record's first statement ID for the resource's position is written into the corresponding next statement ID field in the new statement record. Note that if this resource was newly inserted for this statement, then this step will write a null into the next statement ID field.
   (b) The ID of the new statement is written into the resource record's first statement ID for the resource's position.

4.3 Deletion

The index structure of Parliament's statement table is not conducive to the efficient removal of a statement record from the three linked lists of which it is a member. These linked lists are singly linked, and so there is no way to remove a record except to traverse all three lists from the beginning.

Due to these difficulties, Parliament "deletes" statements by marking them with a flag in the bit field portion of the statement record. Thus, the algorithm consists of a find (as in the case of an insertion, this is a fully bound query pattern) followed by setting the flag on the found record. In the future, we may utilize doubly linked lists so that the space occupied by deleted statements can be reclaimed efficiently. However, in our work to date deletion has been infrequent enough that this has been deemed a lower priority enhancement.

5 Theoretical Analysis

As is readily apparent, the presented approach suffers some unfortunate worst case performance, but the average case performance is quite good. This is consistent with empirical results presented in [10] and this paper. We will address both find operations on a single triple pattern and triple insertions.

5.1 Worst Case Analysis

The worst-case performance for a single triple pattern lookup is dependent on how many of the elements in the pattern (s, p, o) are bound. If zero elements are bound, the triple pattern results in a total scan of the statement list, resulting in $O(n)$. Since all triples are the expected result, this is the best possible worst case performance. If one element is bound, the chain for that particular element will be traversed with time $O(count(bound, pos))$. Again exactly the triples that answer the pattern are traversed.

Things change slightly for the cases where two or three of the (s, p, o) elements are bound. If two elements are bound, the shorter of the two lists will be traversed. This triple pattern can be returned in

$$O(min(count(bound_1, pos_1), count(bound_2, pos_2)))$$

However, this could be $O(n)$ if all triples use the two bound elements. If all three elements are bound, the shortest of the three lists will be traversed. This shortest list will be longest when the set of statements is exactly the three-way cross product of the set of resources. In this case, if the number of resources is m, then the number of statements is m^3 and every list is of length m^2. Thus the list length is $n^{2/3}$, and a find operation for three bound elements is $O(n^{2/3})$.

Since an insertion first requires a find on the triple to be inserted, it incurs the worst-case cost of a find with three bound elements, $O(n^{2/3})$. It also incurs the cost of inserting any nodes in the triple that were not previously known into the dictionary, but this logarithmic time $O(\log m)$ is overshadowed by the worst case find time. After that, adding the triple to the head of the lists is done in $O(1)$ constant time. Thus the worst-case of the insertion operation is $O(n^{2/3})$.

Here we note that this worst-case performance is indeed worse than other previously published approaches, which are logarithmic. However, the scenarios that produce these worst-case results are quite rare in practice, as will be shown in the following section.

5.2 Average Case Analysis

While the worst-case performance is worse than other approaches, analyzing the relevant qualities of several example data sets leads us to believe that the average case performance is actually quite good.

The most relevant feature of a data set with respect to its performance within this indexing scheme is the length of the various statement lists for a particular subject, predicate, or object. For instance, the worst-case time of the insert operation and the find operation with three bound elements is $O(n^{2/3})$, but this is associated with the case that the set of triples is the cross-product of the subjects, predicates, and objects, which is a highly unlikely real world situation. Since these bounds are derived from the shortest statement list, analysis of the average list lengths in a data set is a key measure to how this scheme will perform in the real world.

Table 1. List Length Means (Standard Deviations)

Data Set	Size	Subject	Predicate	Non-Lit Object	Lit Object
Webscope	83M	3.96 (9.77)	87,900 (722,575)	3.43 (2,170)	4.33 (659)
Falcon	33M	4.22 (13)	983 (31,773)	2.56 (328)	2.31 (217)
Swoogle	175M	5.65 (36)	4,464 (188,023)	3.27 (1,793)	3.38 (569)
Watson	60M	5.58 (56)	3,040 (98,288)	2.87 (918)	2.91 (407)
SWSE-1	30M	5.25 (15)	25,404 (289,000)	2.46 (1,138)	2.29 (187)
SWSE-2	61M	5.37 (15)	83,773 (739,736)	2.89 (1,741)	2.87 (300)
DBpedia	110M	15 (39)	300,855 (3,560,666)	3.84 (148)	1.17 (22)
Geonames	70M	10.4 (1.66)	4,096,150 (3,167,048)	2.81 (1,623)	1.67 (15)
SwetoDBLP	15M	5.63 (3.82)	103,009 (325,380)	2.93 (629)	2.36 (168)
Wordnet	2M	4.18 (2.04)	47,387 (100,907)	2.53 (295)	2.39 (271)
Freebase	63M	4.45 (15)	12,329 (316,363)	2.79 (1,286)	1.83 (116)
US Census	446M	5.39 (9.18)	265,005 (1,921,537)	5.29 (15,916)	227 (115,616)

Table 1 shows the mean and standard deviations of the subject, predicate, and object list lengths for several large data sets [19]. There are a few nice properties of this data that are worth noting. First, the average number of statements using a particular subject is quite small in all data sets. The average number of statements using a particular object is generally even smaller, though with a much higher standard deviation. Finally, only the predicate list length generally seems to scale with the size of the data set.

These observations have important implications with respect to the average time of find and insert operations. For find operations, we now know several things to be generally true:

– Either the object or subject list will likely be used for find operations when all three elements are bound. Thus these operations will often touch fewer than 10 triples.

- Due to the previous, insert operations should generally be quite fast.
- The predicate list is only likely to be used for find operations when only the predicate is bound, and thus only when all statements with the given predicate must be touched to be returned anyway.
- Find operations with two bound elements, which have the most troubling theoretical worst-case performance, necessarily include either a bound subject or bound object. As a result, these too should generally be quite fast.

These conclusions collectively suggest real world performance that is much more impressive than the worst-case analysis would imply, and this is shown empirically in the following section.

6 Empirical Analysis

Since Hexastore [13] is the most recently published work in this area, and its indexing structure out performed several of the other approaches, we have focused our empirical evaluation on Parliament as compared to Hexastore. At the time of our evaluation, only the prototype Python version of Hexastore was available for comparison. Future work will compare against the newly released version. This limitation resulted in the relatively small size of this empirical evaluation; we could not go beyond the size of main memory without the comparison becoming unfair to Hexastore. Parliament was tested with 850 million triples in [10].

Evaluation was performed on a MacBook Pro laptop with a 2.6 GHz dual core CPU, 4 GB of RAM, and a 7200 RPM SATA HD, running Mac OS X 10.5.7. This platform was most convenient for execution of both systems. While Hexastore's evaluation focused on only query performance, we feel it is important to include insertion performance and memory utilization as well, as there are many Semantic Web applications for which these factors are significant. We have focused on the Lehigh University Benchmark [20], as it was used in the Hexastore evaluation and contains insertion time metrics as well. We have evaluated LUBM queries 1, 2, 3, 4, and 9. Since the version of Hexastore used does not perform inference, we were forced to modify queries 4 and 9 such that none was required.

The insertion performance graph is shown in Figure 2. The throughput of Parliament stays fairly stable at approximately 35k statements per second. This throughput is 3 to 7 times larger than that of Hexastore, which starts at approximately 9k statements per second, and declines to less than 5k statements per second as the total number of triples increases. Parliament's throughput results include both persisting the data to disk (the Python version of Hexastore is entirely memory-based) and forward chaining for its RDFS inference capabilities.

Figures 3, 4, 5, 6, and 7 show the relative query performance of Parliament and Hexastore on LUBM queries 1, 2, 3, 4, and 9 respectively.

Queries 1, 3, and 4 produce results where both systems appear to be following the same growth pattern, though Hexastore performs slightly better on queries 1 and 3 and Parliament performs better on query 4. Parliament also demonstrates more variability in the query execution times, which is likely a result of the dependency on the operating system's memory mapping functionality.

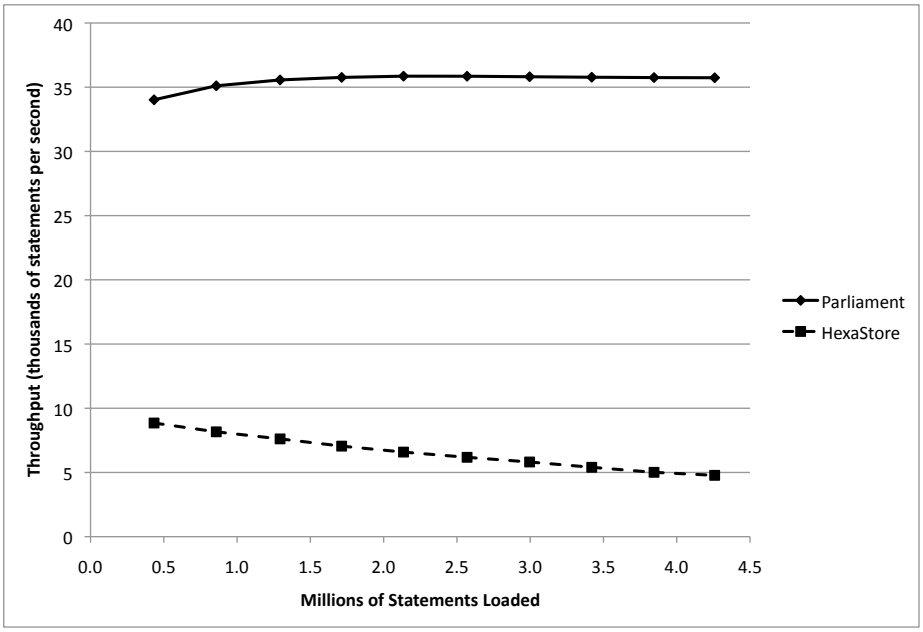


Fig. 2. Insertion Performance

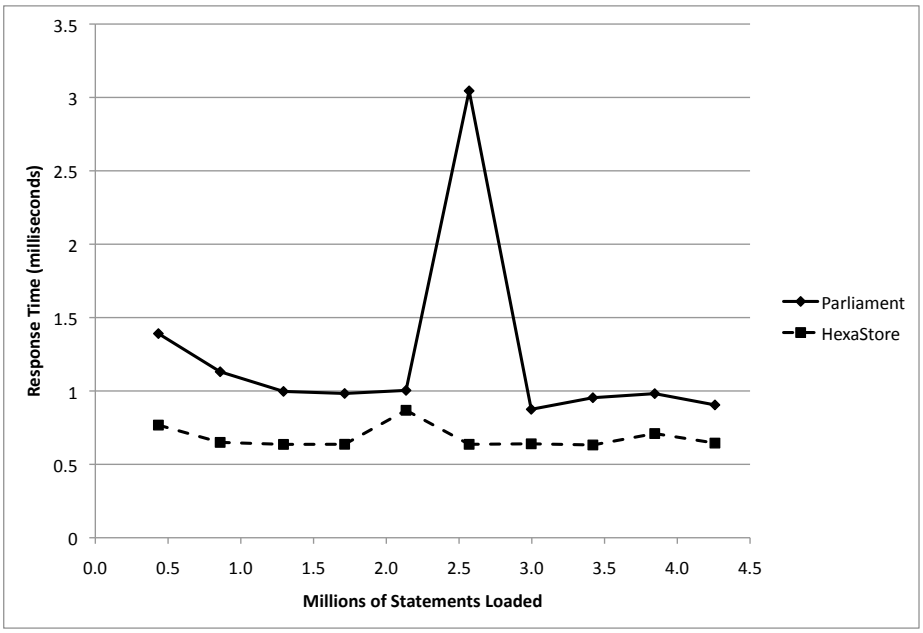


Fig. 3. LUBM Query 1 Response Time

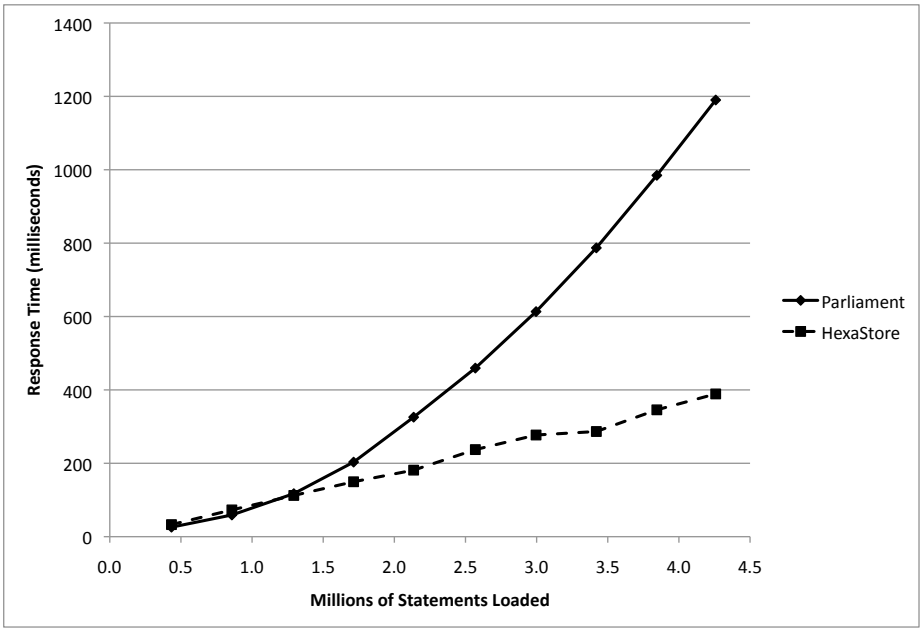


Fig. 4. LUBM Query 2 Response Time

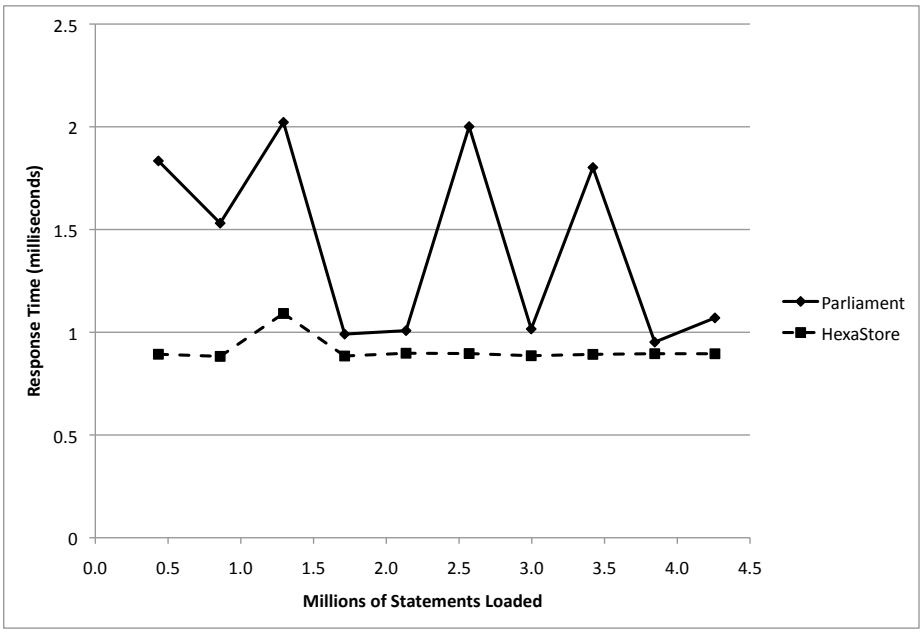


Fig. 5. LUBM Query 3 Response Time

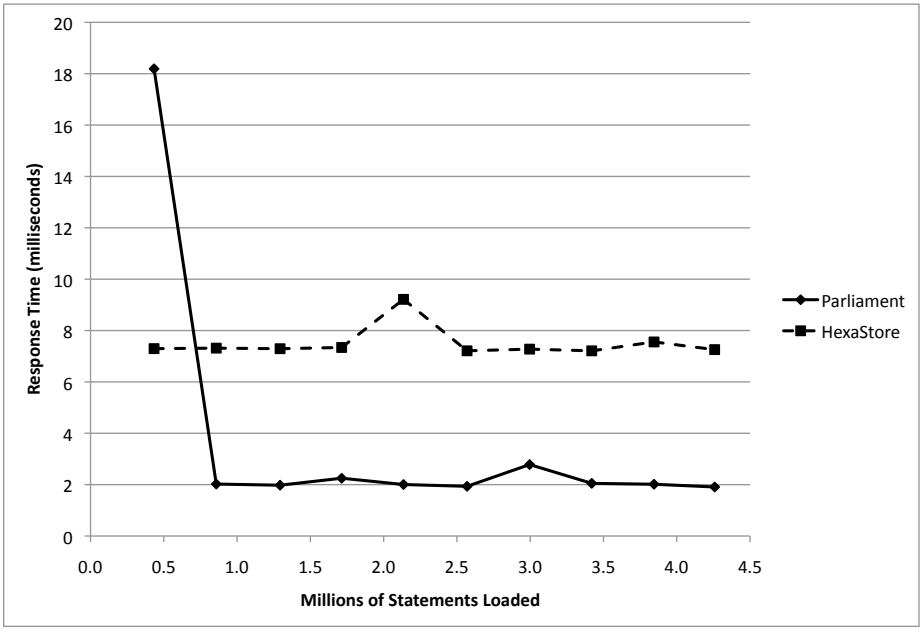


Fig. 6. LUBM Query 4 (modified) Response Time

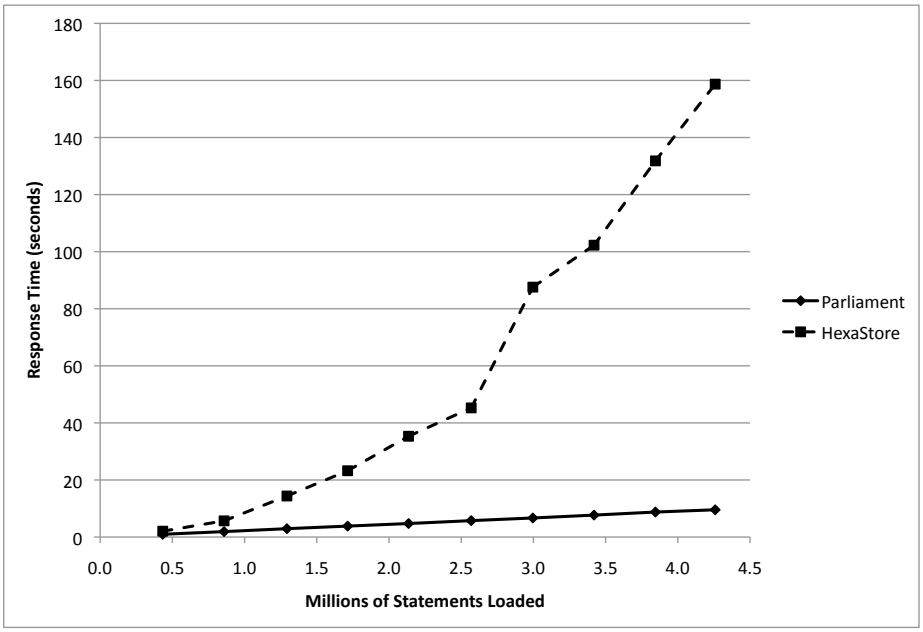


Fig. 7. LUBM Query 9 (modified) Response Time

Queries 2 and 9 show Parliament and Hexastore following different growth curves, with Parliament performing better in query 9 and Hexastore performing better in query 2. This is more likely the result of differing query plans within the two systems than a strength or deficiency of the storage structure, but without insight into the query planner of Hexastore we cannot verify this claim.

Finally, Table 2 shows an estimate of memory used by Hexastore and Parliament with all 4.3M statements loaded. These numbers are as reported by Mac OS X, but as is often the case with virtual memory management, the memory metrics are only useful as course estimates. However, they show what was expected; Parliament's storage scheme requires significantly less storage space.

Table 2. Space Utilization for 4.3M Triples (in GB)

	Hexastore	Parliament
Real Memory	2.02	0.50
Virtual Memory	2.59	1.38
Disk Space	N/A	0.36

Overall, we conclude that Parliament maintains very comparable query performance to Hexastore, while significantly outperforming Hexastore with respect to insertion throughput and required space.

7 Conclusions

In this paper, we have shown the storage and indexing scheme based on linked lists and memory mapping used in Parliament. This scheme is designed to balance insertion performance, query performance, and space usage. We found that while the worst-case performance does not compare favorably with other approaches, average case analysis indicates good performance. Experiments demonstrate that Parliament maintains excellent query performance while significantly increasing insertion throughput and decreasing space requirements compared to Hexastore. Future work will include experiments focusing on different query optimization strategies for Parliament, explanations and analysis of Parliament's internal rule engine, and further optimizations to the storage structure.

References

1. Klyne, G., Carroll, J., eds.: Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation (February 2004) http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.
2. Dean, M., Schrieber, G., eds.: OWL Web Ontology Language Reference. W3C Recommendation (February 2004) http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

3. Harris, S., Shadbolt, N.: Sparql query processing with conventional relational database systems. In: Lecture Notes in Computer Science. Springer (2005) 235–244
4. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D., Database, J.: Efficient rdf storage and retrieval in jena2. In: EXPLOITING HYPERLINKS 349. (2003) 35–43
5. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: VLDB '07: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment (2007) 411–422
6. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An efficient sql-based rdf querying scheme. In: VLDB '05: Proceedings of the 31st international conference on Very large data bases, VLDB Endowment (2005) 1216–1227
7. Erling, O., Mikhailov, I.: Rdf support in the virtuoso dbms. In Auer, S., Bizer, C., Müller, C., Zhdanova, A.V., eds.: The Social Semantic Web 2007, Proceedings of the 1st Conference on Social Semantic Web (CSSW), September 26-28, 2007, Leipzig, Germany. Volume 113 of LNI., GI (2007) 59–68
8. BBN Technologies: Parliament http://parliament.semwebcentral.org/.
9. Dean, M., Neves, P.: DAML DB http://www.daml.org/2001/09/damldb/.
10. Rohloff, K., Dean, M., Emmons, I., Ryder, D., Sumner, J.: An evaluation of triple-store technologies for large data stores. In: On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, Vilamoura, Portugal, Springer (2007) 1105–1114 LNCS 4806.
11. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, New York, NY, USA, ACM (2004) 74–83
12. Broekstra, J., Kampman, A., Harmelen, F.V.: Sesame: A generic architecture for storing and querying rdf and rdf schema. In: Lecture notes in computer science. Volume 2342., Springer (2002) 54–68
13. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proc. VLDB Endow. **1**(1) (2008) 1008–1019
14. Wood, D., Gearon, P., Adams, T.: Kowari: A platform for semantic web storage and analysis. In: XTech2005: XML, the Web and beyond, Amsterdam (2005)
15. Harth, A., Decker, S.: Optimized index structures for querying rdf from the web. Web Congress, Latin American **0** (2005) 71–80
16. Franz, Inc.: AllegroGraph http://www.franz.com/products/allegrograph/.
17. Kiryakov, A., Ognyanov, D., Manov, D.: Owlim — a pragmatic semantic repository for owl. In: Lecture Notes in Computer Science. Volume 3807/2005. Springer (2005) 182–192
18. Olson, M.A., Bostic, K., Seltzer, M.: Berkeley db. In: ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, USENIX Association (1999) 43–43
19. Dean, M.: Toward a science of knowledge base performance analysis. In: Invited Talk, 4th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2008), Karlsruhe, Germany (October 2008) slide 20 http://asio.bbn.com/2008/10/iswc2008/mdean-ssws-2008-10-27.ppt.
20. Guo, Y., Qasem, A., Pan, Z., Heflin, J.: A requirements driven framework for benchmarking semantic web knowledge base systems. IEEE Transactions on Knowledge and Data Engineering **19**(2) (2007) 297–309

BitMat: A Main Memory Bit-matrix of RDF Triples

Medha Atre and James A. Hendler

Tetherless World Constellation,
Dept. of Computer Science
Rensselaer Polytechnic Institute
Troy NY USA
{atrem, hendler}@cs.rpi.edu

Abstract. BitMat is a main memory based bit-matrix structure for representing a large set of RDF triples, designed primarily to allow processing of conjunctive triple pattern (join) queries. The key aspects are as follows: i) its RDF triple-set representation is compact compared to conventional disk-based and existing main-memory RDF stores, ii) basic join query processing employs logical bitwise AND/OR operations on parts of a BitMat, and iii) for multi-joins, intermediate results are maintained in the form of a BitMat containing candidate triples without complete materialization, thereby ensuring that the intermediate result size remains bounded across a large number of join operations, provided there are no Cartesian joins. We present the key concepts of the BitMat structure, its use in processing join queries, describe our experimental results with RDF datasets of different sizes (from 0.2 to 47 million), and discuss the use case scenarios.

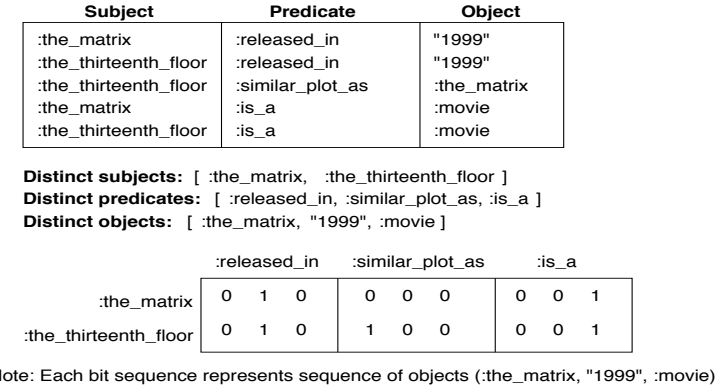
1 Introduction

RDF [4] and SPARQL [16] are gaining importance as semantic data is increasingly becoming available in the RDF format. The growing scale of RDF data necessitates novel ways of storing and querying this data in a compact form. To handle this large scale RDF data, numerous systems are being developed [13, 23, 1]. Many of these systems are implemented as a straight-forward extension of relational database systems and SQL querying techniques. These systems can be broadly classified as persistent disk-based and main-memory-based systems. The work described in this paper belongs to the latter category proposing a main-memory RDF triple store.

Most of the other RDF store systems depend on building efficient auxiliary indexes on the RDF data and using them either for specific type of queries or to improve the overall query performance. In contrast, in our approach, BitMat which is an compressed inverted index structure itself makes up the primary storage for RDF triples. Our proposed query processing algorithm voids the need to uncompress this data at any point during query processing.

Generic relational data can have varied dimensions (i.e. number of columns), and hence the SQL query processing algorithms have to encompass this nature of relational data. As opposed to that, an RDF triple is a fixed 3-dimensional (S, P, O) entity and the dimensionality of SPARQL conjunctive triple pattern queries is also fixed (which depend on the number of conjunctive patterns in the query). Hence while building the BitMat structure and query processing algorithms, we made use of this fact.

In essence, BitMat is a 3-dimensional bit-cube, in which each cell is a bit representing a unique triple denoting the *presence* or *absence* of that triple by the bit value 1 or 0. This bit-cube is flattened in a 2-dimensional bit matrix for implementation purpose. Figure 1 shows an example of a set of RDF triples and the corresponding BitMat representation.



Subject	Predicate	Object
:the_matrix	:released_in	"1999"
:the_thirteenth_floor	:released_in	"1999"
:the_thirteenth_floor	:similar_plot_as	:the_matrix
:the_matrix	:is_a	:movie
:the_thirteenth_floor	:is_a	:movie

Distinct subjects: [:the_matrix, :the_thirteenth_floor]
Distinct predicates: [:released_in, :similar_plot_as, :is_a]
Distinct objects: [:the_matrix, "1999", :movie]

	:released_in			:similar_plot_as			:is_a		
:the_matrix	0	1	0	0	0	0	0	0	1
:the_thirteenth_floor	0	1	0	1	0	0	0	0	1

Note: Each bit sequence represents sequence of objects (:the_matrix, "1999", :movie)

Fig. 1. BitMat of sample RDF data

If the number of distinct subjects, predicates, and objects in a given RDF data are represented as sets V_s, V_p, V_o, then a typical RDF dataset covers a very small set of $V_s \times V_p \times V_o$ space. Hence BitMat inherently tends to be very sparse. We exploit this sparsity to achieve compactness of the BitMat by compressing each bit-row using D-gap compression scheme [7][1].

Since *conjunctive triple pattern* (join) queries are the fundamental building blocks of SPARQL queries, presently our query processing algorithm supports only those. These queries are processed using bitwise AND, OR operations on the compressed BitMat rows. Note that the bitwise AND, OR operations are directly supported on a compressed BitMat thereby allowing memory efficient execution of the queries. At the end of the query execution, the resulting filtered triples are returned as another BitMat (i.e. a query's answer is another result BitMat). This process is explained in Section 4.

Figure 2 shows the conjunctive triple pattern *for movies that have similar plot* and the corresponding result BitMat. Unlike the conventional RDF triple stores,

[1] E.g. In D-gap compression scheme a bit-vector of "0011000" will be represented as [0]-2,2,3. A bit-vector of "10001100" will be represented as [1]-1,3,2,2.

where size of the intermediate join results can grow very large, our BitMat based multi-join[2] algorithm ensures that the intermediate result size remains bounded, (at most to (n * size of the original BitMat), where n is the number of triple patterns in the query), across any number of join operations (provided there are no Cartesian joins).

Query: (?m :similar_plot ?n . ?m :is_a :movie . ?n :is_a :movie)

	:released_in			:similar_plot_as			:is_a		
:the_matrix	0	0	0	0	0	0	0	0	1
:the_thirteenth_floor	0	0	0	1	0	0	0	0	1

Fig. 2. Result BitMat of a sample query

A conventional SPARQL join query engine produces zero or more *matching subgraphs* (each resulting row with variable bindings identifies a matching subgraph). BitMat join processing returns a set of distinct triples in the result BitMat that together form all the matching subgraphs. Currently we are working on an algorithm to enumerate all the matching subgraphs – which is not presented here. But even without this last phase of result generation, current query processing algorithms can be used for:

- *'EXISTS'* or *'ASK'* queries as very large multi-joins can be performed in memory using BitMat (existence of one or more 1 bits in the resulting BitMat indicates that the query will produce at least one matching subgraph).
- As a precursor to an in-memory query processing engine (e.g. Jena-ARQ [2]) to identify the result triples from a large triple set.

BitMat is designed specifically to process conjunctive triple pattern queries and presently the query processing interface does not support full SPARQL syntax or other SPARQL constructs.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the related work. Section 3 describes the BitMat structure and its composition in greater details. Sections 4 and 5 describe the basic algorithm of single join procedure and an algorithm for multi-join based on the single-join algorithm respectively. Section 6 gives our evaluation of the system, and Section 7 concludes the paper with strengths and weaknesses of the present structure of the BitMat and query processing system.

2 Related Work

The structure of a BitMat is somewhat similar to the idea of bitmap indexes, which are used in relational database systems and more recently by the Virtuoso RDF store [8] to efficiently process queries over low cardinality data. The key

[2] Multi-join is a conjunctive triple pattern with two or more triple patterns having two or more join variables and a single join has two triple patterns with only one join variable.

difference is that BitMat's query processor always operates on the compressed BitMat without uncompressing it anytime. This is not possible with a traditional SQL based query processor employing bitmap indexes over multi-joins.

In addition to these, there are various systems being developed for processing RDF data. Some notable ones are – Hexastore [23], RDF-3X [13, 14], BRAHMS [10], GRIN [21], SwiftOWLIM [20], Jena-TDB [1] etc.

Out of these, Hexastore and RDF-3X exploit the nature of RDF data by creating 6-way indexes (SPO, SOP, PSO, POS, OSP, OPS) on it. RDF-3X goes one step further by compressing these indexes and organizing them as clustered B+-trees. BRAHMS and GRIN mainly use their system for path-based queries on RDF graph. But they have not published results on very large RDF graphs, putting the scalability of their system under question. BRAHMS have used LUBM data of only 6 million triples and GRIN has published results for only upto 17,000 triples[3].

The notable difference between these systems and BitMat is that – BitMat's conjunctive triple pattern query processing algorithm which controls the intermediate memory utilization in a large multi-join query.

The structure that comes closest to BitMat is RDFCube [12], which also builds a 3D cube of subject, predicate, and object dimensions. However, RDFCube's design approximates the mapping of a triple to a cell by treating each cell as a hash bucket containing multiple triples. It is primarily used to reduce the network traffic for processing join queries over a distributed RDF store (RDFPeers [5]) by narrowing down the candidate triples. In contrast, BitMat structure maintains unique mapping of a triple to a single bit element and further compresses the BitMat. Our goal here is to represent large RDF triple-sets with a compact in-memory representation and support a scalable multi-join query execution *completely in-memory*.

SPARQL query language, which is structurally quite similar to the SQL query language [6], is being studied specifically with respect to the join processing [9, 19] and query benchmarking [18]. In contrast to the conventional SPARQL query processing scheme, we employ a different approach for multi-joins as elaborated in Section 5.

3 BitMat Concepts

A bit-cube of RDF triples is a 3-dimensional structure with subject (S), predicate (P), and object (O) dimensions. Individual cell is a single bit, and 1 or 0 value of the bit represents presence or absence of the triple. This conceptual bit-cube can be represented as a concatenation of (S,O) or (O,S) matrices for all the distinct predicates thereby forming a *mat of bits*, BitMat. Concatenation done along the subject dimension is referred to as a *subject BitMat* and concatenation done along the *object dimension* is referred to as an object BitMat. Altogether, there are 6 ways of flattening a bit-cube into a BitMat (as is the case of six-way

[3] GRIN focuses on path-like queries which are not even expressible in current SPARQL query language.

indexes on the RDF data). For the current set of experiments, we have used subject BitMats. Exploring other structures of BitMat is a part of future work.

3.1 BitMat Structure

BitMat is constructed from a set of RDF triples as follows: Let V_s, V_p, and V_o represent the sets of distinct subject, predicate, and object values occurring in a RDF triple set. Let V_{so} represent the $V_s \cap V_o$ set. These four sets are mapped to the integer sequence based identifiers as shown below:

- *Common subjects and objects*: Set V_{so} is mapped to a sequence of integers: 1 to $|V_{so}|$ in that order.
- *Subjects*: Set $V_s - V_{so}$ is mapped to a sequence of integers: $|V_{so}| + 1$ to $|V_s|$.
- *Predicates*: Set V_p is mapped to a sequence of integers: 1 to $|V_p|$.
- *Objects*: Set $V_o - V_{so}$ is mapped to a sequence of integers: $|V_{so}| + 1$ to $|V_o|$.

Basically, each ID-space is treated independently with the exception that URIs which appear as subjects as well as objects are assigned the same sequence identifiers. This is done to facilitate the subject-object (S-O) cross dimensional join as discussed in Section 4. Cross-dimension joins over subject-predicate (S-P) or predicate-object (P-O) dimensions are rare in the context of *assertional* RDF data. Since the large scale RDF data available on the web is predominantly assertional, presently we do not handle S-P or P-O cross dimensional joins.

The above mapping is a direct representation of the position of those triples in the BitMat structure. For the example given in Section 1, *:the_matrix* as a subject is mapped to *1*, *:the_matrix* as an object is also mapped to *1*, *:the_thirteenth-_floor* is mapped to *2*, *:similar_plot_as* is mapped to *2* etc. Hence triple (*:the_thirteenth_floor :similar_plot_as :the_matrix*) is represented as (*2 2 1*) indicating to set the first bit (O-position) in the second row (S-position) of the second (S,O) matrix (P-position) (see Figure 1). A complete BitMat is built this way by setting the bit corresponding to each encoded RDF triple. Although this is the conceptual structure of a BitMat, we build the compressed BitMat directly from the encoded triple set as explained further in Section 6.

3.2 BitMat Operations

The process of evaluating conjunctive triple pattern (join) queries is carried out with three primitive operations on a BitMat. They are as follows:

(1) **Filter:** Filter operation is represented as *'filter(BitMat, TriplePattern) returns BitMat'*. It takes an input BitMat and returns a new BitMat which contains only triples that satisfy the *TriplePattern*.

Effectively, *filter* operation on a BitMat involves clearing the bits of the triples that are *filtered out*. For example, a triple pattern with only bound subject value like *filter(BitMat, ':s1 ?p ?o')*, clears all the bits in all the rows other than the row corresponding to the bound subject value *:s1* etc.

(2) **Fold:** Fold function represented as *'fold(BitMat, RetainDimension) returns bitarray'* folds the input BitMat along the two dimensions other than the *RetainDimension*.

For example, if *RetainDimension* is set to *'object'*, then BitMat is folded along the subject and predicate dimensions resulting into a single bitarray. Intuitively, bits set to 1 in this bitarray indicate the presence of *at least* one triple with the object corresponding to that position in the given BitMat. Typically *fold* is called on the BitMat returned by *filter*. E.g. *fold(filter(BitMat, ':s1 ?p ?o'), 'object')*.

(3) **Unfold:** Specified as *'unfold(BitMat, MaskBitArray, RetainDimension) returns BitMat'* takes a BitMat, a bitarray, and *unfolds* the bitarray on the BitMat.

Intuitively, in the *unfold* operation, for every bit set to *0* in the *MaskBitArray* all the bits corresponding to that position of the *RetainDimension* in a BitMat are cleared. Typically *MaskBitArray* is generated by a bitwise AND of the bitarrays returned by *fold* operations before. E.g. *unfold(BitMat, '101001', 'predicate')* would result in clearing all the bits in second, fourth, and fifth (S,O) matrices which correspond to predicates mapped to $\{2, 4, 5\}$.

Filter, *fold*, and *unfold* operations are implemented to operate on a compressed BitMat without uncompressing it.

4 Single Join Processing

A conventional SPARQL join query processing engine produces zero or more *matching subgraphs* (each resulting row with the variable bindings identifies a matching subgraph) (see Figure 3). The query being evaluated is (:s1 ?p ?x . :s3 ?p ?y). Intuitively, every resulting matching subgraph is a proper subgraph of the original RDF graph G, which satisfies the SPARQL query graph pattern (provided there are no Cartesian joins).

BitMat based join algorithm is given in Algorithm 1 and is elaborated as follows.

Algorithm 1 BitMat_SingleJoin(BM, tp_1, tp_2) returns BitMat

1: Let BM be the BitMat of the original triple-set
2: Let tp_1 and tp_2 be the two triple patterns in the join
3: /* filter and fold */
4: BM_{tp1} = filter(BM, tp_1)
5: BM_{tp2} = filter(BM, tp_2)
6: $BitArr_1$ = fold(BM_{tp1}, $RetainDimension_{tp1}$)
7: $BitArr_2$ = fold(BM_{tp2}, $RetainDimension_{tp2}$)
8: $BitArr_{res}$ = $BitArr_1$ AND $BitArr_2$
9: BM_{tp1} = unfold(BM_{tp1}, $BitArr_{res}$, $RetainDimension_{tp1}$)
10: BM_{tp2} = unfold(BM_{tp2}, $BitArr_{res}$, $RetainDimension_{tp2}$)
11: /* Produce the final result BitMat */
12: Let BM_{res} be an empty BitMat
13: BM_{res} = BM_{tp1}OR BM_{tp2}

On lines 4, 5 *filter* operation is used to get two BitMats containing only triples satisfying the first and second triple pattern respectively. This resembles the *selection* operator used in SQL style queries. *Fold* is used on these two BitMats to get $BitArr_1$ and $BitArr_2$. *Fold* is analogous to the *projection* operator

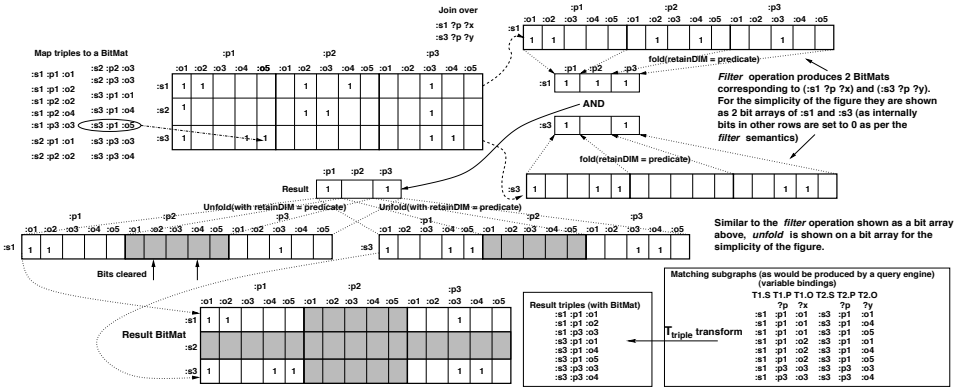


Fig. 3. Single Join on a BitMat

of SQL queries. If RDF triples are presented in a 3-column table (S, P, O), then these bitarrays correspond to a single column in the table and bit positions set to 1 indicate presence of the S, P, or O values corresponding to those bit positions. Bitwise AND is performed on these bitarrays which is same as a relational join on the column represented by *RetainDimension*. The result of the bitwise AND is *unfolded* back on the filtered BitMats BM_{tp1} and BM_{tp2}. Finally the two BitMats obtained after *unfold* are combined using bitwise OR on the corresponding rows of them. This procedure is depicted in Figure 3.

It can be shown and proved step-by-step that *filter*, *fold*, and *unfold* operators can be mapped to equivalent SQL operations and the correctness of the algorithm can be proved. For the scope of this paper, we have omitted these details, but they can be referred in our technical report [3].

For simplicity of presentation of the algorithm, we have shown it only for a single join with two triple patterns, but the same algorithm can be extended to *'n'* triple patterns joining over a single join variable occurring in the same dimension by performing filter and fold on each triple pattern, ANDing all the bitarrays generated by the fold operation, unfolding the AND results on each of the filtered BitMats, and finally combining all these BitMats with bitwise OR on the corresponding rows to get the result BitMat. The procedure for subject-object cross dimensional join (as shown by an example in Section 3.1) is slightly different and is elaborated in Section 4.1.

4.1 Cross Dimensional Joins

Bitwise AND operation can be performed on two bitarrays only if the corresponding bit positions have the same URI or literal values mapped to them. This is the case for the same dimension joins.

Cross-dimensional joins need special handling. (*?s* :p1 ?x . ?y :p2 *?s*) is an example of subject-object (S-O) cross-dimensional join, for which we need to perform bitwise AND on the subject and object bitarrays. As elaborated in

Section 3.1 and Section 3.2, every bit position in the folded bitarray corresponds to a unique identifier assigned to a URI or literal in the respective subject, predicate, object ID space. Since URIs which appear as subjects as well as objects are allocated the same IDs sequentially from 1 to $|V_{so}|$, for a S-O join, bitwise AND is performed only on the first $|V_{so}|$ bit positions of the respective subject and object bitarrays and rest all bits are cleared.

As mentioned earlier in Section 3.1, other cross-dimensional joins (S-P and P-O) are not that common in the Semantic Web instance (assertional) data and hence are not supported at present.

5 Multi Join Processing

In a multi-join two or more triple patterns join over two or more join variables, e.g. (:s1 *?p ?o* . :s2 *?p* ?y . ?z :p3 *?o*). In a conventional relational join processing, multi-join evaluation can be presented as an operator tree where each internal node is a self-contained representation of the materialized results of the join subtree below it. BitMat single join procedure, as elaborated in Section 4, does not materialize the query results (i.e. does not produce matching subgraphs), but represents the candidate result triples with the result BitMat. Thus, if we simply extend the single-join BitMat algorithm to multi-joins, evaluation of a later join can change the variable bindings produced by an earlier join. Specifically, if the dependency between different join variables is not captured and resolved then a BitMat join can produce *false positives*. Hence is the need for a different algorithm for multi-join queries.

5.1 BitMat Multi-Join Algorithm

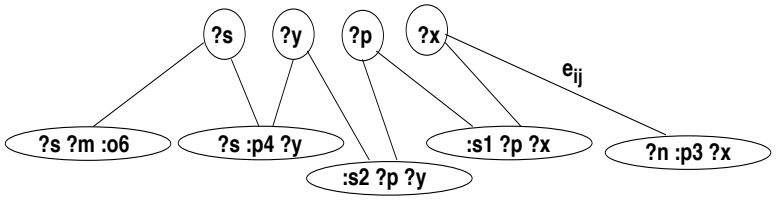


Fig. 4. Multi-join graph $\mathcal{G}$

For the present considerations, we do not handle joins with Cartesian products. For better understanding of the algorithm, we develop the theory by constructing a bipartite graph $\mathcal{G}$ which captures the conjunctive triple pattern and join variable dependencies (see Figure 4).

- Each join variable in the multi-join is a node (denoted as *jvar-node*).
- Each triple pattern is a node (denoted as *tp-node*).

– There is an edge between a jvar-node and a tp-node if the join variable represented by the jvar-node appears in the triple pattern represented by the tp-node.

Algorithm 2 BitMat_MultiJoin(BM, $\mathcal{G}$) returns BitMat

```
1: /* BM is the BitMat of the original triple-set */
2: /* Initialize graph G */
3: for all v_k in graph G
4:    if v_k is jvar-node then
5:       Set BitArr_k to a bitarray with all bits set to 1.
6:    else
7:       BitMat_k = filter(BM, getTriplePattern(v_k))
8:    end if
9: end for
10: repeat
11:    Set changed = false
12:    for each v_i as jvar-node in G do
13:       Let PrevBitArr_i = BitArr_i
14:       /* TP is a set of all triple patterns
15:       * having join-var v_i */
16:       Let TP = {v_j | ∃e_ij}
17:       for each v_j in TP do
18:          Let dim = getDimension (v_i, getTriplePattern(v_j))
19:          Let TempBitArr = fold(BitMat_j, dim)
20:          BitArr_i = (BitArr_i) AND (TempBitArr)
21:       end for
22:       if PrevBitArr_i not equal BitArr_i then
23:          Set changed = true
24:       end if
25:       /* Now unfold the result */
26:       for each v_j in TP do
27:          Let dim = getDimension (v_i, getTriplePattern(v_j))
28:          BitMat_j = unfold(BitMat_j, BitArr_i, dim)
29:       end for
30:    end for
31: until (changed == true and there are more than one join var)
32: Let B_res be an empty BitMat
33: for each v_j as tp-node in G do
34:    B_res = B_res OR BitMat_j
35: end for
```

Although for graph $\mathcal{G}$ shown in Figure 4 there are exactly two edges per jvar-node corresponding to a join variable, one could have more than two edges per join variable.

The algorithm to evaluate a multi-join using graph $\mathcal{G}$ is given in Algorithm 2. For simplicity, we assume the existence of certain methods without describing them in the algorithm, viz. method *getTriplePattern(v_j)* returns the triple pattern associated with the tp-node node v_j, and *getDimension(v_i, tp)* returns the position (dimension) of the join variable v_i in triple pattern *tp*, e.g. *getDimension(?s, (?s :p1 ?x))* returns *subject*, *getDimension(?s, (?y :p2 ?s))* returns *object*, and *getDimension(?s, (?s :p2 ?s))* returns *subject* and *object* (this is a special form of S-O cross dimensional join and is captured by the implementation of the BitMat multi-join algorithm, but not shown in Algorithm 2 for simplicity).

Associated with each jvar-node is a bitarray of the most recent result of a join evaluated over that join variable. Initially all the bits in these bitarrays are set to 1 (as explained in the Algorithm 2). Each tp-node has a BitMat associated with it, which is initially set to the result of the *filter(BM, getTriplePattern(v_j))* where v_j is the tp-node. The *repeat-until* loop (between lines 10 and 31) in Algorithm 2 iterates over all the join variables in the query, processing those joins until none of the $BitArr_i$ change. For each join variable, it *folds* the BitMats associated with all the triple patterns which have that join variable (lines 16, 19) and performs a bitwise AND on the generated *BitArrays* (line 20). At the end of the loop (17-21) the final AND result ($BitArr_i$) is *unfolded* back on all the BitMats in the set TP (line 28). Lastly, after the *repeat-until* loop ends, result BitMat is generated by a bitwise OR of all the BitMats associated with all the triple patterns (line 34) in the query.

Although the multi-join algorithm is constructed as a continuous loop, it can be proved that this loop will converge in a finite number of iterations. In each iteration of the loop, we are performing a bitwise AND on the previous $BitArr_i$ and the new $TempBitArr$ folded from the BitMat of the triple pattern having that join variable. After each bitwise AND, the resulting $BitArr_i$ is unfolded on the BitMats associated with all the triple patterns having that join variable. Since we are doing a bitwise AND operation and *unfold* which expands the $BitArr_i$ on the BitMats, only a bit set to 1 can be flipped to 0. Hence the number of set bits (and hence the triples in the BitMats) reduce monotonically per iteration of the loop and the loop ends at a point when none of the $BitArr_i$ change after an AND (lines 20, 22) (in the worst case when all the bits in all the BitArrays are set to 0, in which case the final join result is *null*).

We provide experimental results in Section 6 for the typical number of iterations taken by the loop. It can be seen that for each join variable, we employ the same basic operations as used for a single join operation.

6 Experiments

This section describes our experiments and evaluation of the BitMat structure and join queries.

6.1 Programming Environment

The BitMat structure and join algorithms have been developed as a C program meant to be run on a Linux distribution. All the experiments were carried out on Gentoo Linux distribution on a Dual Core AMD Opteron Processor 870 with 8GB of RAM. The BitMat program was run as a normal user process with the default priority as set by the Linux system. *Gcc ver. 4.1.1* compiler is used to compile the code with compiler optimization flag set to *-O6*. The RDF N-triple file is first preprocessed using a Perl script to generate a raw RDF triple file by encoding all the triples using the sequence based identifiers allocated to URIs and literals (refer to Section 3.1). Simple bash `sort` command

is used to sort these encoded triples on subject-ID, predicate-ID, object-ID. This preprocessing is needed to be carried out only once per dataset, and the time taken by it varies linearly with respect to the size of the triple-set (e.g. it takes around 30 minutes for Wikipedia 47 million triple-set). BitMat's load function expects either a raw RDF triple file or a disk image of the previously generated BitMat. Given a conjunctive triple pattern (join query), another small script is used to transform the conjunctive triple pattern by encoding all the fixed URIs and literals present in the query using the corresponding identifiers, so that the BitMat join processing can operate on the ID-based values.

6.2 Loading a BitMat

We used different RDF triple sets of varying sizes for testing BitMat structure's memory utilization. UniProt-0.2million and UniProt-22million triple sets were extracted from a larger UniProt dataset [22] (∼730 million triples). Uniprot-0.2million and 22 million datasets were selected from first 50 million triples in the Uniprot 730million triple file. LUBM 1 million and 6 million triple sets were generated using LUBM's [11] RDF data generator program. LUBM 1 million was generated by generating data of 10 universities and LUBM 6 million was generated by generating data of 50 universities. Wikipedia 47 million [24] was used as is available on the web without any modifications in it. The characteristics of these datasets are given in Table 1.

Table 1. Dataset characteristics

Dataset	#Triples	#Subjects	#Predicates	#Objects
Uniprot 0.2million	199,912	30,007	55	45,754
LUBM 1million	1,272,953	207,615	18	155,837
LUBM 6million	6,656,560	1,083,817	18	806,980
Uniprot 22million	22,619,826	5,328,843	91	4,516,903
Wiki 47million	47,054,407	2,162,189	9	8,268,864

Table 2 lists size of the BitMats of respective datasets plus the size of forward and reverse mapping dictionaries to map each literal or URI to an integer ID. A compressed BitMat and these dictionary mappings represent the original RDF data completely. The results given in Table 2 show that this representation is much smaller than the original raw RDF data presented in N-triples format.

The small size of the compressed BitMat is due to two reasons: i) since the actual RDF triple set covers only a small set of the total $V_s \times V_p \times V_o$ space (refer to Section 3.1), BitMat makes a very sparse structure, ii) D-gap compression scheme achieves superior results on sparse bit-vectors.

The raw RDF triple file read by the BitMat load procedure is sorted on (subject, predicate, object) IDs. Hence although conceptually we use the D-gap compression scheme, internally our algorithm exploits the sorted triple list to build a compressed BitMat directly instead of building uncompressed bitarrays

Table 2. BitMat Size and Load Time

Dataset (#triples in millions)	Size of compressed BitMat + mapping dictionary / Size of raw RDF data (MB)	Time to load (sec) from Raw file / from Disk Image
UniProt (0.2)	1.5 + 6.5 / 23	0.34 / 0.04
LUBM (1)	11.6 + 47 / 222	1.54 / 0.36
LUBM (6)	60.8 + 248 / 1193	8.35 / 1.95
UniProt (22)	213.5 + 1367 / 4037	17.11 / 8.4
Wikipedia (47)	371.1 + 932 / 7054	34.4 / 4.5

and then compressing them. This results into smaller load times to construct a BitMat from a raw RDF file. The memory-image of a compressed BitMat can be written out to the disk as a binary file. Loading from a disk-image just reads this binary file into a BitMat structure in memory, hence loading from a disk image takes even lesser time than loading the BitMat from sorted triple-ID file.

Note that each conjunctive triple pattern query is converted into an internal representation where all the fixed values in the query are replaced by their corresponding integer mapping IDs (refer Section 3.1). Although the mapping dictionary consumes larger space than the primary BitMat structure, it does not need to be kept in memory while processing the queries.

6.3 Join Query Performance

To test our implementation of the join algorithm, we executed a list of single join queries on a smaller dataset (UniProt 0.2 million) and also measured the response times (for the list of queries, see Table 3). Typically, the subject join query times varied from 0.019sec to 0.04sec, for predicate joins the variation was from 0.0041sec to 0.062sec, for object dimension join it was from 0.0094sec to 0.128sec, and for S-O cross dimensional joins it was from 0.08sec to 0.28sec. Variation in the time depended on different factors such as the selectivity[4] of the triple pattern and join condition, number of total variables in the query, dimension of the variables in the query, etc. as explained further below. For multi-joins, we used a mix of queries taken from UniProt queries [17], LUBM queries available on OpenRDF [15], and some constructed by us for the Wikipedia dataset. Table 4 lists some of these queries (due to space limitations we cannot enlist all the queries). We noted several parameters that characterize these queries as given in the columns of Table 4.

Memory requirements: The "Sum of BitMat sizes" is the maximum memory size of all the BitMats associated with the triple patterns including the result BitMat at any point during the query execution. Note that BitMat size for a single pattern is the size obtained after applying the *filter*, which usually is much smaller than the original BitMat since a majority of the triple patterns have a fixed value in at least one of the S, P, O positions. But we have successfully tried queries having all variable positions in one or more triple patterns as well (e.g.

[4] A lower selective triple pattern has more triples associated with it and vice versa.

Table 3. Single Join Queries on UniProt-0.2million

Query	#Result Triples	Time (sec)
Subject Joins		
(?s :author ?x)(?s rdf:type ?y)	31,044	0.019
(?s ?p :taxonomy:5875)(?s rdf:type ?y)	2	0.04
(?s ?p ?o)(?s rdf:type ?y)	199,912	0.042
(?s ?p ?o)(?s :author ?y)	43,408	0.02
Predicate Joins		
(?x ?p :P15711) (?y ?p :Q43495)	10	0.062
(:UniProt.rdf#_F ?p ?o) (?y ?p :Q43495)	6	0.034
(:UniProt.rdf#_A ?p ?x) (:UniProt.rdf#_F ?p ?y)	10	0.0041
(?s ?p ?x) (:UniProt.rdf#_F ?p ?y)	75,572	0.062
Object Joins		
(?x :created ?o)(?y :modified ?o)	2056	0.016
(:P15711 ?p ?o)(?y :modified ?o)	33	0.010
(?x ?p ?o)(?y :modified ?o)	2056	0.128
(?x :created ?o)(:P28335 :modified ?o)	19	0.0094
S-O Joins		
(?o :begin ?x)(?y :range ?o)	11,830	0.28
(?x ?p ?o)(?o rdf:type ?y)	51,232	0.08
(?x ?n ?o)(?o ?m ?y)	135,325	0.081

a Wikipedia query in Table 4). The sizes of these BitMats are highest at the beginning of the query, and they go on reducing monotonically as the multi-join algorithm executes. This is due to the fact that *filter*, *fold*, and *unfold* operate on a compressed BitMat, and in every iteration of the multi-join algorithm, triples get eliminated monotonically, hence the BitMat size shrinks. Also we do not materialize the intermediate join results, but represent them as candidate triples in the result BitMat. The size variation also depended on the selectivity of the triple patterns. The higher the selectivity, the lower the BitMat size (due to D-gap compression).

The variation of the query execution times can be attributed to three key factors: i) join-dimension (e.g. whether it is a subject, predicate, object, or S-O cross dimension join), ii) selectivity of the triple patterns, and iii) the order of the join evaluation.

Join dimension: Since we are using a subject BitMat for joins on all the dimensions, subject-dimension joins inherently benefited, as folding and unfolding of a compressed BitMat by retaining the subject dimension involves only updating the relevant subject rows without accessing the compressed content. Since the number of distinct predicates is usually low in the datasets, predicate joins performed well too. However, accessing object dimension in a compressed subject BitMat needed special handling, and hence we observe that the structure of the subject BitMat is unsuited for the object joins since *unfold* requires accessing every O-bit position within each subject row, and for each predicate in

Table 4. Multi-join Queries

Query	**Dataset (million triples)**	**Sum Bit-Mat sizes**	**#Resulting triples**	**Time (sec) / #multi-join loop-iterations**	**#Join var / #all vars / #triple patterns**
(?protein rdf:type :Protein) (?protein :annotation ?annotation) (?annotation rdf:type :Transmembrane_Annotation) (?annotation :range ?range) (?range :begin ?begin) (?range :end ?end)	UniProt (0.2)	355KB	3712	0.066 / 3	3 / 5 / 6
(?p1 rdf:type :Protein)(?p1 :enzyme :enzymes:2.7.1.105) (?p2 rdf:type :Protein) (?p2 :enzyme :enzymes:3.1.3.-) (?interaction rdf:type rdf:Statement) (?interaction rdf:subject ?p1) (?interaction rdf:subject ?p2)	UniProt (0.2)	434KB	0	0.068 / 3	3 / 3 / 7
(?X rdf:type ub:GraduateStudent) (?Y rdf:type ub:University) (?Z rdf:type ub:Department) (?X ub:memberOf ?Z) (?Z ub:subOrganizationOf ?Y) (?X ub:undergraduateDegreeFrom ?Y)	LUBM (1)	2.1MB	994	0.39 / 3	3 / 3 / 6
	LUBM (6)	10.4MB	20,808	3.24 / 3	3 / 3 / 6
(?X rdf:type ub:UndergraduateStudent) (?Y rdf:type ub:FullProfessor) (?Z rdf:type ub:Course) (?X ub:advisor ?Y) (?Y ub:teacherOf ?Z) (?X ub:takesCourse ?Z)	LUBM (1)	4.6MB	10,113	2.17 / 15	3 / 3 / 6
	LUBM (6)	23.1MB	52,029	55.53 / 18	3 / 3 / 6
(?s :title “Dilbert_Bit_Characters”) (?s ?p ?x)(?s2 ?p ?y) (?s2 rdf:type wiki:Article) (?s2 ?n ?z) (?s2 wiki:internalLink ?m)	Wikipedia (47)	1.9GB	1	5.04 / 2	3 / 9 / 6
(?s :title “Dilbert_Bit_Characters”) (?s ?p :Bully)(:Johnny_the_Homicidal_Maniac ?p ?o) (?o rdf:type wiki:Article)	Wikipedia (47)	26.5MB	128	3.34 / 2	3 / 3 / 4

turn. Also for the queries with a triple pattern having variable predicate dimension and a fixed object dimension, e.g. *(?s ?p :o1)*, *fold* operation will require to access a single bit position within all the subject rows, for all the predicates. This effect was observed specifically on very large datasets when the selectivity of the triple pattern was low. This further brought our attention to the aspect of using all or some of the six possible BitMats that can be flattened from a 3D bit-cube, as we explained in Section 3. Usage of different BitMat structures requires changes in the present multi-join algorithm, as *fold* and *unfold* operations have to interoperate between multiple types of BitMats. We are currently exploring this aspect.

Selectivity: Initial selectivity of the triple pattern as well as the selectivity of join played a role in the faster convergence of the multi-join loop. Selectivity of the triple pattern or join result also plays key role in the memory utilization. Higher selectivity of the triple patterns and join results reduces the BitMats' size faster.

Join order: Although in our experiments the multi-join algorithm's loop typically converged in 3 or 4 iterations, as it can be noted from Table 4, the

second LUBM query took many more iterations (15-18) for 1 million as well as 6 million dataset. As join ordering affects the query execution time and size of the intermediate results in a relational scheme, it affects the number of iterations of the BitMat multi-join algorithm as well. In case of a BitMat join, due to the use of compressed BitMats and the fact that we are not materializing the intermediate results, memory utilization was not affected though. For the second LUBM query, we observe that evaluating join over $?Y$ first brought down the multi-join algorithm iterations from 15 to 12. We plan to use an augmented version of the multi-join bipartite graph $\mathcal{G}$ (not shown in Figure 4) that can be used to capture the cyclic or acyclic dependency, so that we can minimize the number of iterations needed to complete the multi-join processing.

7 Conclusions and Future Work

As shown by our experiments, one of the main advantages of the BitMat structure and joins is that the memory requirement of the system is very low. Since the intermediate or final results in a multi-join are not completely materialized, the result size is always bounded by the size of the original BitMat. If the size of the original BitMat is $Size_{BM}$ and n is the number of triple patterns in a multi-join query, then the instantaneous memory utilization while performing a join is always bounded by $O(n * Size_{BM})$ and the final join result size is always bounded by $O(Size_{BM})$.

We are presently developing an efficient algorithm to enumerate all the matching subgraphs from a result BitMat (i.e. final variable bindings). With this the BitMat main-memory triple store can be used as an independent SPARQL join query engine. Also as pointed out in the experimental section, we are exploring the usage of BitMats created by flattening the 3D bit-cube on different dimensions for further improving the performance of the multi-join queries. Development of this algorithm for a fair comparison of query performance and memory utilization with any other state-of-the-art RDF triplestore, is a part of our ongoing work.

The present BitMat query processing algorithm only performs the step of pruning the candidate RDF triples but does not produce final *matching subgraphs*. Hence in this paper, we have just presented empirical results of query performance and memory utilization.

The key aspect of BitMat's algorithms is that they always work on compressed data without uncompressing it at any point.

Due to the ability to create a disk image of a BitMat in memory (as explained in Section 3.1), the BitMat system can be used as a persistent data-store as well, by exploiting the benefits of performing all the operations in main-memory once the BitMat is reloaded from the disk image.

Finally, we conjecture that by creating clusters of machines, each deploying BitMats, extremely large RDF stores can be created and processed in memory. Our future work includes exploring how this can be realized on server-farm like clusters as well as shared memory supercomputers for very large RDF datasets.

Acknowledgements

We would like to thank Dr. Jagannathan Srinivasan of Oracle Corp. for his invaluable inputs during the early development of this work.

References

1. TDB - A SPARQL Database for Jena. http://jena.sourceforge.net/TDB/.
2. ARQ - A SPARQL Processor for Jena. http://jena.sourceforge.net/ARQ/.
3. M. Atre, J. Srinivasan, and J. Hendler. BitMat: A Main Memory RDF store. *Technical Report*, TW-2009-02, January 2009.
4. D. Beckette and B. McBride. RDF/XML Syntax Specification. W3C Recommendation, February 2004. http://www.w3.org/TR/rdf-syntax-grammar/.
5. M. Cai and M. Frank. RDFPeers: A Scalable Distributed RDF Repository based on a Structured Peer-to-Peer Network. In *Proceedings of WWW*, May 2004.
6. R. Cyganiak. A Relational Algebra for SPARQL. *Technical Report, HP Laboratories Bristol*, September 2005.
7. D-gap Compression Scheme. http://bmagic.sourceforge.net/dGap.html.
8. O. Erling. Advances in Virtuoso RDF Triple Storage (Bitmap Indexing), October 2006. http://virtuoso.openlinksw.com/-wiki/main/Main/VOSBitmapIndexing.
9. O. Hartig and R. Heese. The SPARQL Query Graph Model for Query Optimization. In *Proceedings of ESWC*, 2007.
10. M. Janik and K. Kochut. BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. In *Proceedings of ISWC*, 2005.
11. Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/.
12. A. Matono, S. M. Pahlevi, and I. Kojima. RDFCube: A P2P-based Three-dimensional Index for Structural Joins on Distributed Triple Stores. In *DBISP2P in Conjunction with VLDB 2006*, September 2006.
13. T. Neumann and G. Weikum. RDF-3X: A RISC-style Engine for RDF. In *VLDB*, 2008.
14. T. Neumann and G. Weikum. Scalable Join Processing on Very Large RDF Graphs. In *SIGMOD*, 2009.
15. OpenRDF LUBM SPARQL Queries. http://repo.aduna-software.org/viewvc/org.openrdf/?pathrev=6875.
16. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, January 2008. http://www.w3.org/TR/rdf-sparql-query/.
17. Queries on UniProt RDF dataset. http://dev.isb-sib.ch/projects/expasy4j-webng/query.html#examples.
18. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL Performance Benchmark. *CoRR*, abs/0806.4627, 2008.
19. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *Proceedings of WWW*, April 2008.
20. SwiftOWLIM Semantic Repository. http://www.ontotext.com/owlim/index.html.
21. O. Udrea, A. Pugliese, and V. Subrahmanian. GRIN: A Graph Based RDF Index. In *AAAI*, 2007.
22. UniProt RDF. http://dev.isb-sib.ch/projects/uniprot-rdf/.
23. C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *Proceedings of VLDB*, 2008.
24. Wikipedia RDF Dataset. http://labs.systemone.at/wikipedia3.

On-disk storage techniques for Semantic Web data - Are B-Trees always the optimal solution?

Cathrin Weiss, Abraham Bernstein

University of Zurich
Department of Informatics
CH-8050 Zurich, Switzerland
{weiss,bernstein}@ifi.uzh.ch

Abstract. Since its introduction in 1971, the B-tree has become the dominant index structure in database systems. Conventional wisdom dictated that the use of a B-tree index or one of its descendants would typically lead to good results. The advent of XML-data, column stores, and the recent resurgence of typed-graph (or triple) stores motivated by the Semantic Web has changed the nature of the data typically stored.
In this paper we show that in the case of triple-stores the usage of B-trees is actually highly detrimental to query performance. Specifically, we compare on-disk query performance of our triple-based Hexastore when using two different B-tree implementations, and our simple and novel vector storage that leverages offsets.
Our experimental evaluation with a large benchmark data set confirms that the vector storage outperforms the other approaches by at least a factor of four in load-time, by approximately a factor of three (and up to a factor of eight for some queries) in query-time, as well as by a factor of two in required storage. The only drawback of the vector-based approach is its time-consuming need for reorganization of parts of the data during inserts of new triples: a seldom occurrence in many Semantic Web environments.
As such this paper tries to reopen the discussion about the trade-offs when using different types of indices in the light of non-relational data and contribute to the endeavor of building scalable and fast typed-graph databases.

1 Introduction

The increasing interest in the Semantic Web has motivated a lot of recent research in various areas. That is because the dynamic graph-structured character of Semantic Web data is challenging many traditional approaches, for example those of data indexing and querying. So it does not come as a surprise that there has been done a lot of work to improve state of the art Semantic Web engines. Recent publications show how to index Semantic Web data efficiently [1, 8, 17], how to improve query optimization processes [9, 14], and how to represent the data. Most of them aim to avoid mapping data to the relational scheme.

However, beyond optimizing in-memory indices, state-of-the-art systems still make use of traditional data structures, such as B-trees, when it comes to on-disk storage. It is surprising that so far no effort has been made on analyzing whether those storage structures are a good match for the new indices. Will traditional approaches still work well with the newly developed indexing techniques?

In this paper we discuss this issue and propose a novel, but simple lookup-based, on-disk vector storage for the Hexastore indices. The vector storage employs key offsets rather than a tree structure to navigate large amounts of disk-based data. The use of fixed-length indices makes lookups highly efficient ($O(1)$, with at most three page reads) and loads efficient. While updates are a bit more costly, the fast load time makes it, oftentimes, simpler to "just" reload the whole data. We benchmark our vector storage for Hexastore with two Hexastore-customized B-tree based approaches. One has a B-tree index for each of the Hexastore indices. Another has one B-tree index which combines all indices. We will see that for typical queries the vector storage outperforms a B-tree based structure by a factor of eight.

In summary our contributions in this paper are the following:

- We propose a simple but novel approach for on-disk storage of triples that relies on an offset-based vector storage. The approach allows for a highly compact representation of the data within the index while preserving a fast retrieval forgoing some insert/update efficiency – a tradeoff that doesn't seem too disadvantageous given the nature of the data.
- We experimentally compare the load-time and space requirement performance of the vector index with two different implementations of the a B-tree style index and two different versions of a traditional table-based triple store. We show that the vector storage based Hexastore has a smaller storage footprint than a B-tree based approach and that it is much faster in loading the data. Furthermore, we show that the vector storage based Hexastore has about three to eight times the speed of the B-tree based approaches in answering queries confirming the theoretical considerations.

The remainder of the paper is structured as follows. First we discuss the work related to investigating non-tree data structures. Section 3 summarizes the structure of Hexastore, introduces the novel vector storage structure, and explains how the two B-tree-based back-ends for Hexastore are constructed. Section 4 discusses the advantages and disadvantages of each of the indices and is complemented by the experimental evaluation. We close with a discussion of limitations and our conclusions.

2 Related Work

We found three areas of related work: other work on typed-graph (or RDF) stores, projects focusing on the native storage of XML data, and other papers investigating the limitations of usefulness of tree-based storage.

Efficient indexing of Semantic Web data has become an active research topic [1, 8, 13, 17]. Most approaches propose methods how to rearrange data in-memory or in given database systems respectively such that query processing can be performed more efficiently compared to straightforward approaches like triple tables. Abadi et al [1] store their vertical partitioning approach in a column-oriented database [2, 15]. Specifically they store each predicate in a two-column $< subject, object >$ table in a column store, indexed with an unclustered B-tree. Neumann et al. [8] as well as Guha [12] use native B-tree storage approaches. The goal with our proposed vector storage is to avoid storage in existing DBS and also the usage of B-tree structures and to show that applying these approaches is detrimental to query performance.

One of the oftentimes used serializations of an RDF graph is in XML. Projects in the XML domain have investigated a plethora of approaches to efficient storage and querying of this type of data. We can distinguish between non-native storage strategies, which map the data onto the relational model, and native strategies, that try to store the data more according to its nature. Native XML databases [3, 5, 7] typically store their data either as the XML document itself, or they store the tree structure, i.e., the nodes and child node references. For indexing, some index-related information may be stored as well, such as partial documents, sub-tree information, and others. All of these approaches have in common that they store their data (and usually their indices) in tree like structures, as the underlying data is also in that format. The one exception is the native on-disk XML-storage format proposed by Zhang et al. [18]. It provides an optimized, non-tree disk-layout for XML-trees optimized to answer XPath queries. Akin to this last project, we also propose to shed the limitations of the underlying data format. Indeed, our on-disk structure consists only of the indices themselves, which helps to answer queries about the underlying data and, hence, enables their reconstruction.

It was most difficult to find work investigating the boundaries of tree-based indices. Idreos et al. [4] address the slow build-time of an index in general by proposing not to build an index at load time but to initially load the data in its raw format and reorganize it to answer each query. They show that under certain conditions the data organization converges to the ideal one. In the most radical attack on the general applicability of trees, Weber et al. [16] discuss similarity searches in high-dimensional spaces. They find that the performance of tree-based indices radically degrades below the performance of a simple sequential scan. They propose a novel vector-approximation scheme called VA-file that overcomes this "dimensionality curse". Our work can be seen in the spirit of these studies in that we also try to explore the limitations of the predominant tree structures and propose alternatives that excel under certain conditions.

3 The Storage Structure

In this section we describe our vector storage for Hexastore indices and data. Hexastore, proposed in [17] as an in-memory prototype, is an efficient six-way

indexing structure for Semantic Web data. In order to explain the functionality of the vector storage we first briefly review the functionality of Hexastore itself and its requirements towards an index. We then introduce the vector storage and discuss its technical and computational characteristics. This is followed by a brief explanation on how to build the B-tree based Hexastore back-ends.

3.1 Hexastore: A Sextuple index based graph store

A Semantic Web triple $< s, p, o >$ consists of a *subject* s, which is a node in the graph, a *predicate* p designating the type of edge, and an *object* o, which is either a node in the graph or a literal. In RDF, the nodes are identified by a URI. To limit the amount of storage needed for the URIs, Hexastore uses the typical dictionary encoding of the URIs and the literals, i.e. every URI and literal is assigned a unique numerical *id*. Furthermore, Hexastore recognizes that queries are constructed by a collection of graph patterns [14] which may (i) bind any of the three elements of the triples to a value, (ii) may use variables for any of the triple elements effectively resulting in a join with other graph pattern, and (iii) define any element with a wild card to be returned. Consequently, any join order between triple patterns in the query is possible. Hexastore, therefore, indexes the data to allow for retrieving values for each order of a triple pattern respectively joining over every element (s, p, or o) of a triple pattern resulting in six indices designated in the order in which the triple elements are indexed (e.g., SPO, OSP, etc.). This structure allows to retrieve all connected triple residuals with one index lookup.

Figure 1 illustrates the structure of the six indices. It shows that each index essentially consists of three different elements: a first-level `Type1` index, a second-level `Type2` index, and a third-level `Type3` ordered set, where the `TypeN`'s are one of the three triple elements $\{subject, predicate, object\}$. Given a key a_i the first-level `Type1` index returns a second-level `Type2` index. Given a key b_j the `Type2` index returns a `Type3` ordered set, which lists all the matches to the query $< a_i, b_j, ? >$. As an example consider trying to match a query that tries to find all papers authored by "Bayer". This query could result in the triple structure $< Bayer, authored, ? >$ and could be executed by consulting the *spo* index (i.e., `Type1` $= s$, `Type2` $= p$,and `Type3` $= o$). Hence, first the `s` index would be asked to return the `p` index matching the key "Bayer", then the returned index would be asked to return the ordered set for the key "authored", which would be the result of the query. Note that this structure has the advantage that every lookup is of amortized cost of order $O(1)$.

Our implementation of Hexastore presented in [17] used an in-memory prototype for all experiments. Storing the first and second level indices on disk so that all Hexastore performance advantages can be preserved is not straightforward. Clearly the proposed indexing technique does not adhere to the traditional relational model. Still, taking inspiration from RDF-3X, we could use B-trees as the well established indexing technique – an approach that we use to compare our results to (see Section 3.3). But as we argued in the introduction we believe that since our data adheres to different underlying assumptions, we would loose

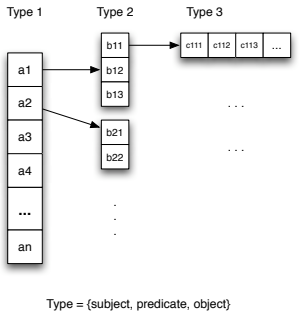


Fig. 1. Sketch of a Hexastore index

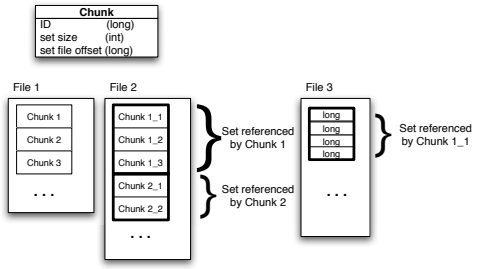


Fig. 2. Vector storage file layout

some of the advantages of the Hexastore index. Consequently, we need an on-disk index structure/storage layout that adheres to the following requirements:

1. Given an item a of `Type1` and a desired target `Type2`, there should be an operation on the first-level index that efficiently retrieves the associated second-level index of `Type2`, denoted as $\mathfrak{I}$(`Type2`). Hence we need an efficient implementation of the operation:

$$getIdx(a) : \texttt{Type1} \mapsto \xrightarrow{PointerTo} \mathfrak{I}(\texttt{Type2})$$

2. Given an item b of `Type2` and a desired target `Type3`, there should be an operation operating on the second-level index that efficiently retrieves the associated third-level ordered set `Type3`, denoted as $\mathfrak{S}$. Hence we need an efficient implementation of the operation:

$$getSet(b) : \texttt{Type2} \mapsto \xrightarrow{PointerTo} \mathfrak{S}(\texttt{Type3})$$

3. The operations $getIdx(a)$ and $getSet(a)$ should require as few read operations as possible.
4. The third-level sorted set should be accessible in as few read operations as possible.

Obviously, this linked structure can be implemented in various ways. Given that the structure is reminiscent of linked vectors one approach would be to store each vector-like structure in a column store. This would fulfill requirements 3 and 4 above. In practice, however, this approach does not scale, as the number of vectors is huge: while the number of first-level indices is only six, the number of second level indices has an upper bound of $2|S| + 2|P| + 2|O|$ and the number of third-level ordered sets would be, due to the fact that the third-level sets can be shared by two indices, $|S|(|P| + |O|) + |O||P|$ – a prohibitively large number. MonetDB [2], for example, allocates one file for each column. Since most vectors are small (requiring far less space than 4 KB, which is the default minimal size

for each new file), this approach would lead to a tremendous waste of space. In the following, we discuss three implementations. First, we introduce our own vector storage. Then, we show two different approaches of how the two necessary structure operations (*getSecondLevelIndex*(a) and *getThirdLevelSet*(a)) can be implemented using a traditional B-tree.

3.2 Vector Storage

The Hexastore structure favors vector-like indices connected with pointers. To mimic this behavior on-disk we needed to establish the analogues of vectors and pointers on-disk whilst limiting the number of files needed (to avoid having a large numbers of almost empty second and third level files). As an analogue for a pointer we chose a storage structure we call a *chunk* (see also Figure 2 at the top). A chunk contains the particular *id* (the *id* that results of the dictionary encoding) of a URI or literal represented as a `long integer`, the number of elements in the associated second-level index (or third-level sorted set) it points to represented as `integer`, which can be chosen as `long integer` as well, if necessary, and the offset information where to find the associated set data in the level-two, respectively level-three file, again represented as `long integer`.

Chunks allow for the efficient storage of the first and second-level indices in a single file each. Figure 2 shows the layout for each of the six Hexatore indices. If an element with *id* i is stored, the appropriate chunk in the ith position of File 1 has the value `ID` set to i and `size` set to the number of chunks it points to in the second-level index. The value `offset` contains the position of the start of the files referenced by the chunk within File 2. Some nodes that appear as *subjects* might also be *objects* in some triples, while other nodes are only referred to as either *subjects* or *objects*. In the latter case the first-level index of the SPO, SOP, OSP, and OPS has to return a NULL value, as those ids are not used as *subject* respectively *object*. If that is the case the particular entry is filled with a "zero" chunk, i.e. the `ID`, `size` and the `offset` are set to 0. While this approach "wastes" some space to "zero" chunks, it maintains a placeholder for every possible *subject* and/or *object* id allowing to compute the location of a chunk associated with a given id by simply multiplying the *id* times the disk footprint of a chunk (i.e., $(\texttt{id} - 1) \cdot \texttt{sizeof}(\texttt{chunk})$). In the *predicate*-headed first-level indices (belonging to the overall Hexastore indices PSO and POS) we avoid the necessity of "zero" chunks all together by using a different key-space for the dictionary encoding. The second-level index is also stored as a collection of chunks grouped in a single file denoted as File 2 in Figure 2. Since the lookup in the first-level index provided us with the offset as well as the size (or number) of chunks associated with its key in the second-level index we can again start to directly read the relevant chunks and know how far we need to read. Note that the second-level index does not use "zero" chunks, since the entries associated with a first-level key are typically much fewer than the number of nodes (or edges). Consequently, the offset-jump method of finding a second-level chunk associated with a key requires a search for the chunk. The size of the second-level group of chunks corresponds to the degree of a given node (or the number

of differing types of predicates it is associated with) in the case of a *subject* or *object* first-level index, or the number of nodes connected with a certain type of edge in the case of a first-level *predicate*. Hence, in most cases, the typical size of a second-level group of chunks is going to be small enough to fit into main memory allowing an efficient binary search. The third-level sorted sets are again all stored in a single file denoted as File 3 in Figure 2. Reading the sorted set associated with a second-level chunk results in a simple reading of the `size` ids starting from the `offset` in the file.

The presented on-disk structure allows to store each index in 3 files resulting in a total of 15 files (in addition to the dictionary store). Note that the number of File 3s can be halved, as two Hexastore indices can share them (e.g., the SPO and PSO index can share their third-level list File 3). Returning to our requirements of Section 3.1 we can summarize:

1. $getIdx(a)$ can be implemented as a simple lookup based on an off-set calculation (i.e., $(\texttt{id} - 1) \cdot \texttt{sizeof}(\texttt{chunk})$). It requires at most one page read.
2. $getSet(b)$ can be implemented as a search over an ordered set of chunks in File 2: In the best case it will involve one page read followed by a binary search (or a simple binary search if the page is already in memory). In the worst case it might involve multiple page reads (if the chunk group is larger than the page size) and partial binary searches.
3. Few page reads (first and second level): In the first-level index lookup only the relevant page is read. In the second-level index only the pages associated with the relevant chunk group are read.
4. Third-level page reads: only pages containing the relevant third-level ordered set are read.

3.3 B-tree Based Implementations

As a base-line comparison and following the conventional wisdom we implemented two different Hexastore implementations based on B-trees. We decided to use BerkeleyDB B-trees rather than from-the-scratch implemented ones, as they a) are efficiently implemented, and b) allow for flexible key-value definitions per default.

The first approach stores each Hexastore index in a separate B-tree. We refer to it as the *One-For-Each (OFE)* approach. The second approach stores all Hexstore indices in a single B-tree. Therefore, we refer to it as the *All-In-One (AIO)* approach.

OFE: Storing each Hexastore index in one B-tree In this approach we store each Hexastore index in a separate B-tree. Thus, instead of using the structure shown in Figure 2 we implement it as a B-tree with a compound key consisting of the lookup value for the first-level and the second-level index in the form of:

$$
\begin{aligned}
&BtreeLookup(< a, b >): \\
&\qquad < \texttt{Type1}, \texttt{Type2} > \mapsto \delta(\texttt{Type3}) \quad \text{if} \quad b > 0 \\
&\qquad\qquad < \texttt{Type1} > \mapsto \delta(\texttt{Type2}) \quad \text{otherwise.}
\end{aligned}
$$

This provides an easy lookup for each of the operations necessary to implement a Hexastore operation. The typical lookup of the sorted set of third-level keys for a given set of first-level and second-level keys is a straightforward call of the $BtreeLookup(a, b)$ function. Note that the vector storage requires calling $b = getIdx(a)$ followed by calling $getList(b)$ to achieve the same operation. If only the second-level keys are needed then the second-level key is passed as 0. Hence, $getIdx(a) = BtreeLookup(< a, 0 >)$. The result of this implementation is that all lookups are optimized using the chosen B-tree implementation

AIO: Storing Hexastore in one B-tree In this approach we even further reduce the number of needed B-trees by adding the type of index (SPO, SOP, ...) to the compound key. Thus:

$$\begin{aligned} &BtreeLookup(< a, b, t >) : \\ &< \texttt{Type1}, \texttt{Type2}, \texttt{idxType} > \mapsto \S(\texttt{Type3}) \quad \text{if} \quad b > 0 \\ &\qquad < \texttt{Type1}, \texttt{idxType} > \mapsto \S(\texttt{Type2}) \quad \text{otherwise,} \end{aligned}$$

where $\texttt{idxType} \in \{\mathsf{SPO}, \mathsf{SOP}, \mathsf{PSO}, \mathsf{POS}, \mathsf{OSP}, \mathsf{OPS}\}$. Again partial lookups are achieved with setting b to 0 and efficiency is handled by the B-tree implementation.

Note, that assuming an efficient implementation of a B-tree the main difference between *OFE* and *AIO* lies in the implementation ability to reuse/share elements of the tree and the approach to dealing with compound keys. The simpler key structure of *OFE* suggests a faster index build time. But an efficient handling of compound keys in *AIO* could lead to a better reuse of already read pages and could lead to better retrieval times.

4 Experimental Evaluation

In our evaluation we wanted to provide empirical evidence for our claim that B-trees can be suboptimal in some situations and that they are outperformed by our vector storage. To compare the vector storage, which we implemented in C++, with B-tree based approaches, we implemented both B-tree variants, *AIO* and *OFE*, described in Section 3.3 using the Oracle Berkeley DB [10,11] library in release 4.7. Inspired by [13], in which the authors show that proper B-tree indices over triple tables can already be highly efficient and scalable, we also inserted the data into a standard MySQL 5 database table with three columns, one for subject, one for predicate, and one for object. We then created indices over all three columns. This indexed MySQL table is referred to as *iMySQL*. Finally, we also used the unindexed MySQL database as a baseline for some of the comparisons.

According to our goal we wanted to benchmark the systems with respect to index creation times, required disk space, and data access (or retrieval) time. All experiments were performed on a 2.8GHz, 2 x Dual Core AMD Opteron machine with 16GB of main memory and 1TB hard disk running the 64Bit version of Fedora Linux.

Data Sets As we wanted to measure the behavior of the different storages under various sizes we looked for a suitable typed-graph data set. We chose the Lehigh University Benchmark (LUBM) [6] data set, which models typical academic setting of universities, classes, students, instructors, and their relationships. The advantage of this synthetical data set is that it has an associated data generator which can generate an arbitrary number of triples within the given schema and that it has a number of associated queries with their correct answers. It is, therefore, widely used for benchmarking Semantic Web infrastructure applications. Table 1 summarizes the number of subjects, predicates, objects, and triples for the used data sets.

# Triples $\|T\|$	$\|P\|$	$\|S\|$	$\|O\|$	$\|N\|$
5 Mio	18	787,288	585,784	1,191,500
25 Mio	18	3,928,780	2,921,363	5,948,606
50 Mio	18	7,855,814	5,842,384	11,894,568

Table 1. Number of triples, predicates, subjects, objects, and nodes of the used LUBM data sets used

Experimental Procedure To ensure that all systems would have the same starting conditions regardless of any string-handling optimizations we replaced all URIs to unique numerical ids. In addition we also replaced all literals with unique numerical ids mimicking a dictionary index. Note that according to the approach chosen in Hexastore the numerical ids came from two different sets of numbers: one for *predicates* as well as a second one for *subjects* and *objects* (since nodes can be both *subjects* and *objects*).
To further ensure that the data would be loaded uniformly in the same way and equalize possible differences between data-loaders we first built an in-memory index from the data source files and then built the different on-disk indices in exactly the same order. The exception was the MySQL loads, which had to rely on SQL `INSERT` statements instead of calling a direct index API function.

4.1 Creation Time

To compare the index creation time under practical conditions using the LUBM data set we measured the time from the first API-call until the data was entirely written to disk. The results are presented in Figures 3 and 4. Surprisingly, as depicted in Figure 3 the vector storage takes only marginally longer than the simple (unindexed) MySQL insert batch process and already outperforms the indexed MySQL table. As Figure 4 clearly indicates, both Berkeley DB B-tree implementations take a very long time to create the on-disk index and are clearly outperformed by the vector storage: to write a 50 million triples index, vector storage requires about 50 minutes, whereas *OFE* requires 11.5 hours, and *AIO*

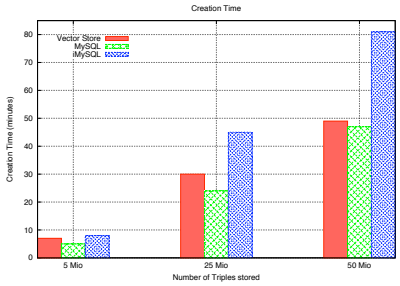


Fig. 3. Comparison of index creation times of the vector storage and indexed (iMySQL) and unindexed MySQL

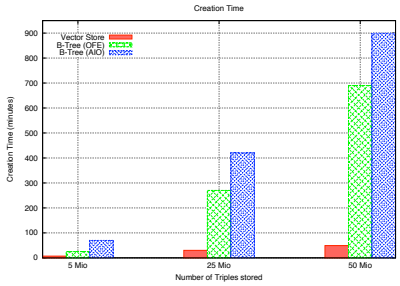


Fig. 4. Comparison of index creation time of vector storage and OFE/AIO

15 hours. These numbers show clearly that the vector storage index is built much faster than any of the B-tree based indices.

4.2 Required Disk Space

The required disk space was determined by looking at the sum of the file sizes of the respective stores containing a particular amount of triples. The results for that are shown in Figures 5 and 6. We can see in Figure 5 that the ordinary unindexed MySQL triples table requires the least amount of space, which is no surprise. In fact it requires constantly approximately 4.3 times less space than our vector storage approach. The B-tree-indexed MySQL version requires about 33% less space than the vector storage. These findings weaken the original criticism of Hexastore that its six indices use too much space. Indeed we argued in in [17] that Hexastore would use at most 5 times the space than a single index variant under worst-case conditions. We see here that a relational triple store with index on each column (which is necessary given the types of queries typically used in typed graph stores) uses only 33% less space. As shown in Figure 6 both Hexastore-customized Berkeley DB approaches clearly require more disk space than the vector storage. The *OFE* approach requires most storage, i.e. 2.3 times as much as vector storage. The *AIO* approach requires less but still twice as much space as the vector storage. A final observation is that for the LUBM data the vector storage requires about 100 MB per million triples. This linear growth is probably observed due to the uniformity of the interconnections of the data generated by the LUBM generator, which adds new universities when additional numbers of triples are required.

4.3 Retrieval Time

The most important question was how the different systems would compare in terms of retrieval time. To ensure that we measure the time spent by querying

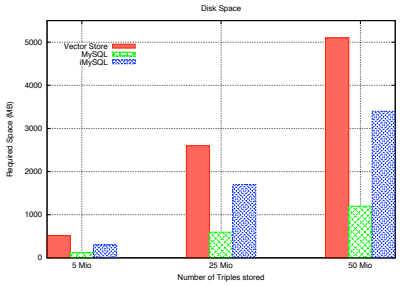


Fig. 5. Comparison of required disk space for indices stored with vector storage and (i-)MySQL

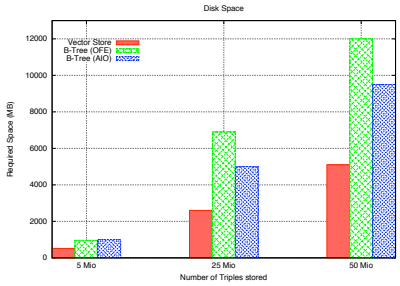


Fig. 6. Comparison of required disk space for indices stored with vector storage and AIO/OFE

the index and not any performed pre- or post-processing operations, we measured retrieval time from the moment all strings in the queries were converted to ids until the final numerical results were retrieved. We also did not take into account the final materialization step because we wanted to compare the immediate behaviors of the different Hexastore implementations without being biased by an additional lookup index. Thus the total time includes finding the key position(s) and fetching the desired data from disk. To avoid measuring the quality of query optimization we restricted ourselves to single triple patterns. Note that we refrained from including retrieval times for MySQL in this experiment, since they were significantly worse than those of the other approaches. Specifically, we evaluated the following requests, each as cold and warm run (i.e., uncached and cached):

1. Retrieve all predicates for a given subject: $< s, ?p, \cdot >$
2. Retrieve all objects for a given subject and predicate: $< s, p, ?o >$
3. Retrieve all subjects for a given predicate: $<?s, p, \cdot >$

The ids chosen for the given subject or predicate were determined by a random generator.

The results of the first request are shown in Figures 7 (cold run) and 8 (warm run). This request is highly selective and does not fetch much data. For the vector storage approach it chooses the SPO index and fetches the subject information from the first file and the set of associated predicates. There is no need to fetch any information from the third file. Both B-tree approaches have to perform key searches and fetch a small corresponding data chunk (in our particular experiment run the given subject had three associated predicates). We can see that the vector storage approach allows data retrieval in this case for a cold run 2–3 times faster than the B-tree approaches and for a warm run about 8 times faster. Given its structure the vector storage can probably answer this query with only two page reads, which are then cached in the warm-run. The B-trees have to perform more page-reads as they had to "navigate" down the tree. Interestingly, this navigation takes longer for the *OFE*, which has six

smaller indices, in the cold-run case but exactly the same time in the warm-run case.

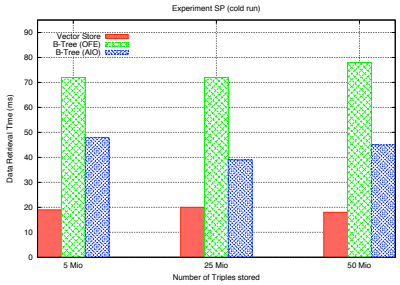


Fig. 7. Comparison of retrieval time for vector storage and AIO/OFE, all predicates for a given subject $< s, ?p, \cdot >$ (cold run)

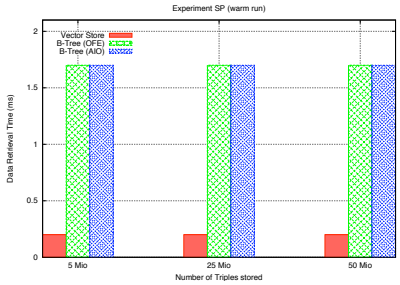


Fig. 8. Comparison of retrieval time for vector storage and AIO/OFE, all predicates for a given subject $< s, ?p, \cdot >$ (warm run)

The second request fetches all objects for a given subject and predicate. The vector storage needs to touch all three files to collect the required data, including the predicate position determination in the second file within the known range (compare Section 3.2). Both B-tree approaches work similarly to the first request except for searching for a different key. This request is again highly selective, i.e. in our experiment we had five objects connected to the given subject over the given predicate.
The results are presented in Figures 9 and 10. Again we can see that the B-trees are outperformed by vector storage by a factor of 1.5 – 3 for a cold run and 8 for warm runs respectively. All three approaches have in common that retrieval times remain constant with increasing number of stored triples for highly selective data retrieval. Comparing the results of this query to the last one it is interesting to observe that the cold-run case of this higher specified query is actually evaluated faster than the one that "only" resolves one level. In the warm-run case any advantage is lost due to caching.

The last request retrieving all subjects for a given predicate is analogous to the first one, but has a low selectivity and requires a lot of data to be fetched from disk. In this concrete case, the number of subjects associated with the particular predicate were $376,924$ in the 5 million triples case, $1,888,258$ in the 25 million triples case, and $3,776,769$ in the 50 million triples case. It becomes therefore obvious that in this case page size for a single retrieval is clearly exceeded. Figures 11 and 12 show the corresponding results, where each of the approaches has to retrieve a significant amount of more result triples as the size of the data set increases. Correspondingly, the time needed for retrieval also increases. The large number of retrieved result keys necessitates an increasing number of serial page reads in the second-level index file for the vector storage. Due to the big

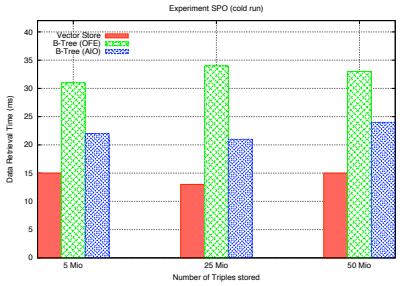


Fig. 9. Comparison of retrieval time for vector storage and AIO/OFE, all objects for a given subject and predicate $< s, p, ?o >$ (cold run)

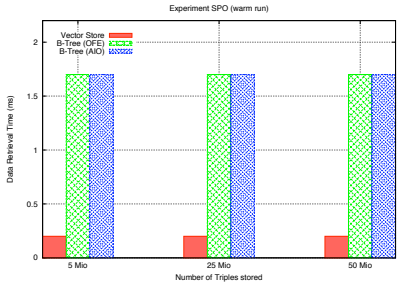


Fig. 10. Comparison of retrieval time for vector storage and AIO/OFE, all objects for a given subject and predicate $< s, p, ?o >$ (warm run)

data amount to be read, the B-tree needs to read more pages and, in case of overflows, determine their positions respectively beforehand. The vector storage outperforms both B-trees by a factor of 1.5 as it can leverage the sequential structure of the second-level index, which is cheaper than traversing overflow pages.

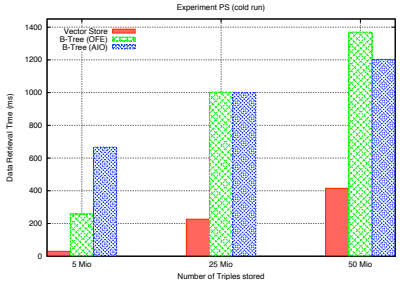


Fig. 11. Comparison of retrieval time for vector storage and AIO/OFE, all subjects for a given predicate $<?s, p, \cdot >$ (cold run)

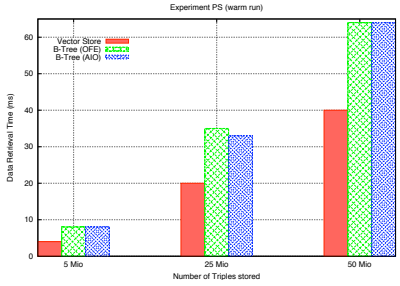


Fig. 12. Comparison of retrieval time for vector storage and AIO/OFE, all subjects for a given predicate $<?s, p, \cdot >$ (warm run)

4.4 Results Summary

The presented results clearly show the advantages of our vector storage scheme. We have shown empirically that it provides significantly lower data retrieval times compared to B-trees. The differences are "only" in the tens or hundreds of milliseconds. However, real world applications are likely to store even more triples and to combine a series of such single graph pattern to a query. This

would require a series of lookups such as the ones we discussed here and further widen the gap.

Most computer science algorithms employ some type of time/space tradeoff. In this case, we do not: besides being faster, the novel vector storage approach also only requires half of the space of B-trees. Creation of the on-disk vector storage is fast. Indeed, it is several orders of magnitude faster than creating the corresponding B-trees and almost on par with creating a simple triple table in MySQL without indices.

5 Limitations and Future Work

The goal of our work was twofold. First, we tried to re-open the discussion on the general applicability of trees as the "one-size-fits-all" index and second to present our vector storage format as an efficient on-disk format for semantic web data storage. Even though the experimental evaluation shows that the vector storage exhibits better performance characteristics our findings have some limitations.

First and foremost, our findings are limited by the introductory permanence assumption. If the data stored in a Hexastore would entail many updates, then the overall performance balance might not be so clearly in favor of the vector storage. Again, our basic assumption, supported by many usages of RDF stores in the semantic web, is that such updates are rare and that reloading the whole store in those rare occasions would be faster than using a slower index.

Second, our approach is limited to storing numerical ids requiring a dictionary index for URI-based ids and literals. This approach requires that literal-based query processing elements would be handled by the dictionary index, for example for SPARQL[1] `FILTER` expressions. To address this limitation we are currently investigating an extension of the dictionary index to efficiently handle such elements. Its discussion was, however, beyond the scope of this paper.

Third, as illustrated in the evaluation of the $<?s, p, \cdot >$ query in the last section, all approaches suffer when they need to retrieve very large amounts of data from the second-level index. This is oftentimes the case if the first-level index is not very selective such as when the number of predicates $|P|$ is small compared to the number of triples $|T|$ (18 versus 50 millions in the example query). If such a triple pattern would be queried in the context of a query containing many patterns then the query optimizer would call it at a later stage due to its low selectivity. To speed things up in other cases (or for very loosely bounded queries with lots of results) one could consider to forgoing some of the vector storage compactness for highly unselective first-level indices and introduce the zero-chunks at the second-level. We would foresee that such an approach would only seldomly be chosen: mostly in the PSO and POS second-level indices when $|P| << |T|$. We hope to investigate this further in future work.

Fourth, the LUBM data set is obviously only one possible choice and has its limitations. It is a synthetic data set and has the limitations associated with

[1] `http://www.w3.org/TR/rdf-sparql-query/`

such data. Nonetheless, it is realistic in that it has a small number of relationship types but many entries/triples – a typical observation in real-world data. Also, it is a heavily used data set, which makes our results comparable to a series of other studies. We hope to extend our evaluation to other large data sets in the future. Given our careful theoretical evaluation we are, however, confident that the new experiments will reflect the results presented here.

Last but not least, our approach was conceived in the context of RDF data and the evaluation only employs such data. As a consequence our findings should only be generalized beyond graph-basd data with caution. In particular, the vector storage index we proposed was geared towards serving as a back-end to Hexastore and might not be quite as useful in other setting. Nonetheless, we believe that scrutinizing the basic assumption of the ubiquitous applicability of B-trees is a fruitful takeaway in itself and should be considered in all areas of database research.

6 Conclusions

In this paper we set out to question the universal superiority of B-tree-based index structures and presented a simple vector-based index structure that outperforms the former in typical graph-based RDF-stye data. Specifically, we departed from three assumptions about RDF data: its structurelessness, its permanence, and its mostly path-style queries. Based on these assumptions we proposed to exploit the structurelessness in favor of a novel storage format based on storing the data in indices rather than in their raw format. Exploiting the difference in permanence (compared to traditional transaction-focused RDBMS) we optimized the indices for loads and reads. We showed empirically that the proposed vector storage index for Hexastore outperforms state-of-the-art B-tree implementations both in terms of load time (by over one order of magnitude) and retrieval time (up to eight times faster). We also showed, that the proposed structure had a load time comparable to an unindexed MySQL $< s, p, o >$ table and even slightly outperformed the load into a similar table providing an index over all three columns (which would be needed to answer any realistic queries). Consequently, as the vector-storage-backed Hexastore so clearly outperformed the other solutions, we can confirm that under certain conditions following conventional wisdom can be considered harmful.

As such the presented paper and its vector storage index can be seen as a first step in developing new on-disk storage structures that are better suited for Semantic Web data. The goal of this endeavor is to gather the building blocks for truly scalable fast stores for typed graph (and thus Semantic Web) data.

7 Acknowledgments

This work was partially supported by the Swiss National Science Foundation under contract number 200021-118000.

References

1. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB '07*, pages 411–422. VLDB Endowment, 2007.
2. P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, Asilomar, CA, USA, January 2005.
3. G. Feinberg. Native XML database storage and retrieval. *Linux J.*, 2005(137):7, 2005.
4. S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, Asilomar, CA, USA, January 2007.
5. C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. In *ICDE '00*, page 198. Society Press, 2000.
6. Lehigh University Benchmark. LUBM Website. `http://swat.cse.lehigh.edu/projects/lubm/`, September 2008.
7. X. Meng, D. Luo, M. L. Lee, and J. An. OrientStore: a schema based native XML storage system. In *VLDB '2003*, pages 1057–1060. VLDB Endowment, 2003.
8. T. Neumann and G. Weikum. RDF-3X: a RISC-style Engine for RDF. In *Vol. 1 of JDMR (formely Proc. VLDB) 2008*, Auckland, New Zealand, 2008.
9. T. Neumann and G. Weikum. Scalable Join Processing on Very Large RDF Graphs. In *SIGMOD 2009*, Providence, USA, 2009. ACM.
10. M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.
11. Oracle. Berkeley DB Reference Guide, Version 4.7.25. `http://www.oracle.com/technology/documentation/berkeley-db/db/ref/toc.html`.
12. R. V. Guha. rdfDB : An RDF database. `http://www.guha.com/rdfdb/`.
13. L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-Store Support for RDF Data Management: not all swans are white. In *Vol. 1 of JDMR (formely Proc. VLDB) 2008*, Auckland, New Zealand, 2008.
14. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW '08*, pages 595–604, New York, NY, USA, 2008. ACM.
15. M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. In *VLDB '05*, pages 553–564. VLDB Endowment, 2005.
16. R. Weber, H.-J. Schek, and S. Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB '98*, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.
17. C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *Vol. 1 of JDMR (formely Proc. VLDB) 2008*, Auckland, New Zealand, 2008.
18. N. Zhang, V. Kacholia, and M. T. Özsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *ICDE '04*, page 54, Washington, DC, USA, 2004. IEEE Computer Society.

OneQL: An Ontology-based Architecture to Efficiently Query Resources on the Semantic Web

Tomas Lampo[1] and Edna Ruckhaus[1] and Javier Sierra[1] and María-Esther Vidal[1] and Amadís Martínez[1,2]

[1] Universidad Simón Bolívar, Caracas, Venezuela
{tomas,ruckhaus,javier,mvidal}@ldc.usb.ve
[2] Universidad de Carabobo, Valencia, Venezuela {aamartin}@uc.edu.ve

Abstract. The widespread explosion of Web accessible resources has led to new problems on the traditional tasks of query evaluation and efficient data access. With this is mind, we have developed the ontology-based OneQL system which provides optimization and query evaluation techniques to scale up to large RDF/RDFS documents and complex queries, i.e., queries of any shape and with a large number of triple patterns. Efficiency of OneQL relies on the following components:

- Query optimization and evaluation techniques that focus on cost models to estimate the execution time of a plan, and on searching the space of plans of any shape, i.e., bushy plans can be generated according to their estimated cost.
- *Bhyper*: A directed hypergraph-based representation of RDF documents to directly access triples that share the same subject and property values, or the same property and object values.

We report on the quality of the developed strategies, and have observed that implementing RDF documents with *Bhyper* and producing low estimated cost bushy plans, can speed up the evaluation time by up to four orders of magnitude.

1 Introduction

Emerging infrastructures such as the Semantic Web, the Semantic Grid and Service Oriented architectures, support on-line access to a wealth of ontologies, data sources and Web services. Ontologies play an important role in the Semantic Web and provide the basis for the definition of concepts and relationships that make information integration possible. Knowledge represented in ontologies can be used to annotate data, distinguish similar concepts, and generalize and specialize concepts published by data sources or produced by Web services. A great number of ontologies have become available under the umbrella of the Semantic Web; some of these ontologies can be very large, impacting in this way the tasks of ontology query answering and reasoning; for instance, MeSH, NCI Cancer, and GO are good examples of ontologies comprised of thousands of concepts. Furthermore, the number of available Web data sources and services has exploded during the last few years. For example, currently, the molecular biology databases collection includes 1,078 databases [12], that is 110 more than the previous year [11]; tools and services as well as the number of instances published

by these resources, follow a similar progression [6]. In addition, thanks to this wealth, users rely more on various digital tasks such as data retrieval from public data sources and data analysis with Web tools or services organized in complex workflows. Thus, in order to be capable of scaling up, Web architectures have to be tailored for query processing on large number of resources and instances. We have aimed at these problems, and have proposed the OneQL system.

OneQL is based on query optimization and evaluation techniques to efficiently execute SPARQL queries. Ontologies are implemented as a deductive database whose predicates represent knowledge explicitly expressed in the ontology, and the semantics of the vocabulary terms. To efficiently store and index the RDF documents where ontologies are defined, we have proposed a directed hypergraph formal model named *Bhyper*. Basically, a *Bhyper* structure is defined by a set of nodes and a set of hyperarcs; each hyperarc connects a set of source nodes to a set of target nodes. In a *Bhyper* structure, the information is stored only in the nodes, and the hyperarcs preserve the role of each node and the concept of direction of RDF graphs. Thus, each resource (subject, property, or value) is stored only once, and the space complexity of an RDF document is reduced if a resource appears several times in the document. Besides, *Bhyper* structures define implicit position-based indices [25] for an RDF document, which can support efficient evaluation of queries over the document.

This paper is comprised of seven sections. The next section summarizes the related work. In section 3 we briefly describe the OneQL system architecture. We then discuss our research in query optimization and evaluation. Section 5 describes *Bhyper*, a *hypergraph*-based representation for RDF documents. The experimental study is reported in section 6, and finally, section 7 outlines our conclusions and future work.

2 Related Work

In the context of the Semantic Web, several query engines have been developed to access RDF documents efficiently [4, 13–16, 20, 31]. Jena [15, 32] provides a programmatic environment for SPARQL, and it includes the ARQ query engine and indices which provide an efficient access to large datasets. The ARQ-Optimizer is a system that implements heuristics for selectivity-based Basic Graph Pattern optimization, proposed by Stocker et al. [28]. These heuristics range from simple triple pattern variable counting to more sophisticated selectivity estimation techniques; the optimization process is based on a greedy optimization algorithm which may explore a reduced portion of the space of possible plans, i.e., only left linear plans. Hence, ARQ-Optimizer query plans can sometimes be far from the optimal plans. Tuple Database or TDB [16] is a persistent graph storage layer for Jena. TDB works with the Jena SPARQL query engine (ARQ) to support SPARQL together with a number of extensions (e.g., property functions, aggregates, arbitrary length property paths). It is a pure-Java component that employs memory mapped I/O, and a customized implementation of B^+-trees to index three different triple patterns permutations, i.e., *spo*, *pos*, and *osp*.

Sesame [31] is an open source Java framework for storage and querying RDF data. It supports SPARQL and SeRQL queries which are translated to Prolog; the join operator is implemented as sideways-passing of variable bindings, which is similar to our

Index Nested Loop Join (NJoin) operator. YARS2 (Yet Another RDF Store, Version 2) [13] is a federated repository for queries against indexed RDF documents. YARS2 supports three types of indices that enable keyword lookups, perform atomic lookup operations on RDF documents, and speed up combinations of patterns or values. Indices are implemented by using an in-memory sparse index data structure that refers to on-disk block entries which contain the indexed entry; six combinations of triple patterns are indexed. A general query processor on top of a distributed Index Manager was implemented, and SPARQL queries are supported; however, no SPARQL specific optimization or evaluation techniques have been developed.

RDF-3X [20] focuses on an index system, and its optimization techniques were developed to explore the space of plans that benefit from these index structures. RDF-3X query optimizer implements a dynamic programming-based algorithm for plan enumeration, which imposes restrictions on the size of queries that can be optimized and evaluated. Indeed, in certain cases, these index-based plans could coincide with OneQL optimized plans; however, the RDF-3X optimization strategies are not tailored to identify any type of bushy plans or to scale up to queries with at least one Cartesian product.

AllegroGraph [4] uses a native object store for on-disk binary tree-based storage of RDF triples. AllegroGraph also maintains six indices to manage all the possible permutations of subject (s), predicate (p) and object (o). The standard indexing strategy is to build indices whenever there are more than a certain number of triples. The query optimizer is based on join ordering for the generation of execution plans; no bushy plans are generated. Hexastore [30] is a main memory indexing technique that uses the triple nature of RDF as an asset. RDF data is also indexed in six possible ways, one for each possible triple pattern permutation. However, the prime drawback of the Hexastore lies in storage space usage; it may require a five-fold increase in storage space compared to a triple table; also, the same resource can appear in multiple indices. Furthermore, two second memory index-based representations and evaluation techniques are presented in [8, 19]. [8] propose indexing the universe of RDF resource identifiers, regardless of the role played by the resource; although they are able to reduce the storage costs of RDF documents, since the proposed join implementations are not closed, the properties of the index-based structures can only be exploited in joins on basic graph patterns. In contrast, [19] propose an index-based representation for RDF documents that maintains the results for subject-subject joins, object-object joins and subject-object joins. Although these structures can speed up the evaluation of joins, this solution may not scale up to strongly connected very large RDF graphs.

GiaBATA [14] is a SPARQL engine built on top of the dlvhex reasoning engine for HEX-programs, and the DLVDB [29] ASP solver with persistent storage. GiaBATA does not implement an RDF-based cost model, but purely relies on join reordering optimizations of DLV and optimizations of the underlying relational database system.

Finally, [1, 2, 27] propose different RDF store schemas to implement an RDF management system on top of a relational database system. They empirically show that a physical implementation of vertically partitioned RDF tables, may outperform the traditional physical schema of RDF tables. Similarly to some of the existing state-of-the-art RDF systems, the optimization techniques are not tailored to identify bushy plans.

3 Architecture

Figure 1 presents the architecture of the OneQL system; it is comprised of a *Query Planner*, a *Query and Reasoning engine*, and an *Ontology catalog* [23].

Ontologies are modeled as a deductive database (DOB) which is composed of an extensional and an intensional database. The extensional database (EDB) is comprised of meta-level predicates that represent the information explicitly modeled by the ontology; for each ontology language built-in vocabulary term, there exists a meta-level predicate (e.g., $subClassOf$). The intensional database (IDB) corresponds to the deductive rules that express the semantics of the vocabulary terms (e.g., the transitive properties of the $subClassOf$ term).

Queries are described as SPARQL queries and are posted to OneQL through a SPARQL-based API which translates each query into a conjunctive query on the predicates in DOB. The conjunctive query is then passed to the optimizer which uses statistics stored in the catalog and Magic Sets rewriting techniques to identify an efficient query execution plan. Next, the plan is given to the query and reasoning engine, which evaluates it against DOB.

The statistics that describe the ontologies stored in DOB include: cost of inferring intensional facts, cardinality of extensional and intensional facts, and number of resources. These statistics are used by the *hybrid cost model* to estimate the cost of a given query plan. Finally, a *hypergraph*-based structure named *Bhyper* is used to index predicates in DOB and to speed up the tasks of query processing and reasoning.

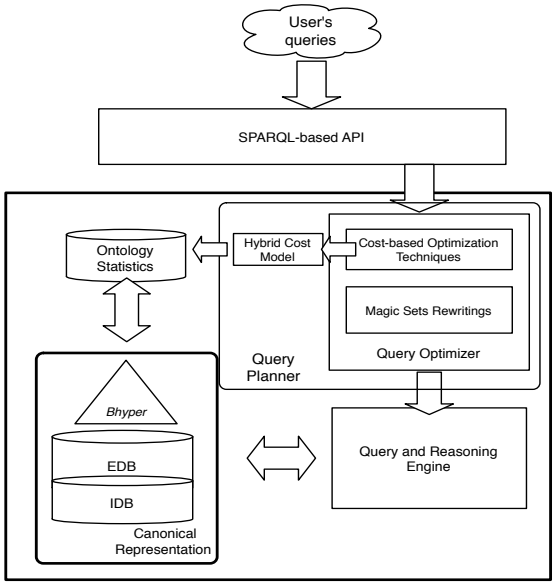


Fig. 1. The OneQL System Architecture

4 Optimizing and Evaluating SPARQL queries

OneQL implements optimization and evaluation techniques to support the execution of SPARQL queries. The proposed optimization techniques are based on a cost model that estimates the execution time or facts inferred during query evaluation; they are able to produce query plans of any shape.

The *Query Planner* component in OneQL (Figure 1) is built on top the following two sub-components [22–24]: a hybrid cost model that estimates the cardinality and evaluation cost of the predicates that represent the ontology's explicit and implicit facts, and a twofold optimization strategy to identify bushy plans. The *Query Engine* relies on several physical operators and *Bhyper*-based indices to efficiently evaluate SPARQL queries.

4.1 The Hybrid Cost Model

In the hybrid cost model, evaluation cost is measured in terms of the number of intermediate inferred facts, and the cardinality corresponds to the number of valid answers of the query pattern. This model estimates the cost and cardinality of explicit and implicit facts, as follows:

- To estimate the cardinality and cost of the intensional predicates that represent implicit facts, we have applied the Adaptive Sampling Technique [17]. This method does not need to extract, store or maintain information about the data that satisfies a particular predicate, and does not make any assumptions about statistical characteristics of the data, such as distribution. Sampling stop conditions are defined to ensure that the estimates are within an appropriate confidence level.
- To estimate the cardinality and cost of the extensional predicates, and the cost of a query plan, we use a cost model *à* la System R [26]. Similarly to System R, we store information about the number of ground facts corresponding to an extensional predicate, and the number of different values (constants) of each predicate variable. Formulas for computing the cost and cardinality are similar to the different physical *join* formulas in relational queries.

4.2 The TwoFold Optimization Technique

A twofold optimization strategy that combines cost-based optimization and Magic Sets techniques was developed. In the first stage of the query optimization component, dynamic-based or randomized algorithms can be applied to identify a good ordering or grouping of the patterns in a SPARQL query. On one hand, the dynamic-programming algorithm works on iterations, and during each iteration the best intermediate sub-plans are chosen based on the cost and the cardinality that were estimated using the hybrid cost model. In the last iteration of the algorithm, final plans are constructed and the best plan is selected in terms of the estimated cost. This optimal ordering reflects the minimization of the number of intermediate inferred facts using a top-down evaluation strategy. This dynamic-based algorithm is performed on queries with a small number of

triple patterns in the where clause, and it is able to produce only left-linear plans which are not always the best solution for RDF-based queries.

On the other hand, the randomized algorithm performs random walks over the search space of bushy execution plans; the query optimizer implements a Simulated Annealing algorithm. Random walks are performed in stages, where each stage consists of an initial *plan generation step* followed by one or more *plan transformation steps*. An equilibrium condition or a number of iterations determines the number of transformation steps. At the beginning of each stage, a query execution plan is randomly created in the plan generation step. Then, successive *plan transformations* are applied to the query execution plan during the plan transformation steps, in order to obtain new plans. The probability of transforming a current plan p into a new plan p' is specified by an acceptance probability function $P(p, p', T)$, that depends on a global time-varying parameter T called the *temperature*; it reflects the number of stages to be executed. The function P may be nonzero when $cost(p') > cost(p)$, meaning that the optimizer can produce a new plan even when it is worse than the current one, i.e., it has a higher cost. This feature prevents the optimizer from becoming stuck in a local minimum. Temperature T is decreased during each stage and the optimizer concludes when $T = 0$. Transformations applied to the plan during the random walks correspond to the SPARQL axioms of the physical operators implemented by the query and reasoning engine. The Simulated Annealing-based optimizer scales up to queries of any shape and number of triple patterns, and is able to produce execution plans of any shape.

In the second stage, the optimizer applies Magic Set optimization techniques [21] to the execution plan obtained in the first stage. Magic Sets combines the benefits of both, top-down and bottom-up evaluation strategies and tries to avoid repeated computations of the same subgoals, and unnecessary inferences. The deductive database program DOB is rewritten w.r.t. the optimal execution plan, and then evaluated with a bottom-up strategy. "Magic predicates" are inserted into the program to represent bounded arguments in the query, and "Supplementary predicates" are included to represent sideways information-passing in rules. It should be noted that we implemented the general Magic Sets technique for Datalog with the two improvements suggested by [3] to eliminate the first and last redundant supplementary predicates, and to merge consecutive sequences of extensional predicates in rule bodies.

4.3 The OneQL Query Engine

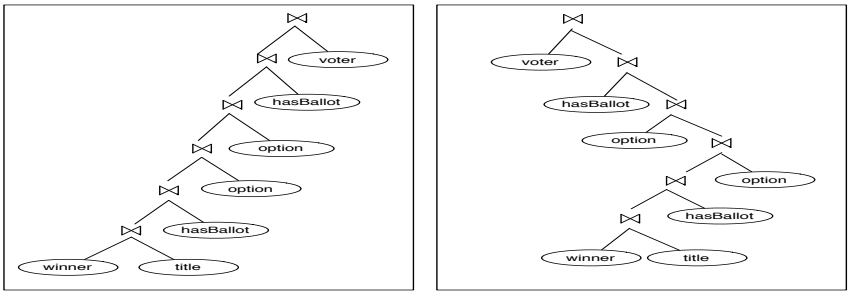
The query and reasoning engine implements different strategies (operators) used to retrieve and combine ontology facts. We have defined different operators that implement the retrieval and combination of ontology facts, and make use of the direct access provided by the *Bhyper*-based structures:

1. Index Nested-Loop Join. For each matching triple in the first pattern, we retrieve the matching triples in the second pattern, i.e., the join arguments[3] are instantiated in the second pattern through the sideways passing of variable bindings. The Index Nested-Loop Join was implemented by extending the sideways-passing of information inherent to Prolog rules, with *Bhyper* indices that allow a direct access to

[3] The join arguments are the common variables in the two predicates that represent the patterns.

the inner pattern triples that match the join variable values of each outer pattern triple; once a result is produced, the computation of the operator is forced to fail, and backtracking takes place to produce a new answer.

2. Group Join. The main idea of this operator is to partition the patterns that appear in the 'WHERE' clause of a query into groups that are comprised of a relatively small number of triples. The Group Join was implemented by first evaluating each group independently, and then asserting in the SWI-Prolog main memory database the results produced by each group; the main memory predicates used to temporally store the results of each group, are indexed by using SWI-Prolog indices. Finally, the main memory stored results are checked to identify matches. Similarly to the Index Nested-Loop Join, the Group Join control is implemented by forcing the computation of the operator to fail when a solution is produced, and using backtracking to generate more solutions.



(a) Left-linear tree (b) Bushy tree

Fig. 2. Query Execution Plans

To illustrate the behavior of the proposed optimization and evaluation techniques, consider the dataset that publishes the US Congress bills voting process [4]. Suppose that the following SPARQL query is posed against OneQL: *Select all the bills and their title where "Nay" was the winner, and at least one voter voted for the same option than the voter L000174.*

```
PREFIX  vote: <tag:govshare.info,2005:rdf/vote/>
PREFIX  dc: <http://purl.org/dc/elements/1.1/>
PREFIX people:
<http://www.rdfabout.com/rdf/usgov/congress/people/>
SELECT  ?E ?T FROM <http://example.org/votes>
WHERE {?E  vote:winner 'Nay' .
       ?E dc:title ?T . ?E vote:hasBallot ?I .
       ?I  vote:option ?X .?J vote:option ?X .
       ?E vote:hasBallot ?J .
       ?J vote:voter 'people:L000174'}
```

[4] http://www.govtrack.us/data/rdf/

Following the optimization techniques reported in [28], only left linear plans as the one reported in Figure 2(a) will be produced; for this left linear plan, the evaluation time is 8,466 secs. On the other hand, our proposed optimization techniques are able to produce bushy trees as the one reported in Figure 2(b) whose evaluation time is 122 secs, i.e., one order of magnitude cheaper than the left linear plan.

5 Bhyper: A Hypergraph-based representation for RDF/RDFS documents

OneQL stores RDF triples using a directed hypergraph-based representation [18]. Basically, a directed hypergraph is defined by a set of nodes and a set of hyperarcs, each one of them connecting a set of source nodes (named tail of the hyperarc) to a set of target nodes (named head of the hyperarc). Directed hypergraphs have been successfully used as a modeling tool to represent concepts and structures in many application areas: formal languages, relational databases, production and manufacturing systems, public transportation systems, topic maps, among others [5, 9, 10]. An RDF directed hypergraph is defined as follows:

Let D be an RDF document. We define a *Bhyper* RDF representation D as a tuple $\mathcal{H}(D) = (W, E, \rho)$ such that:

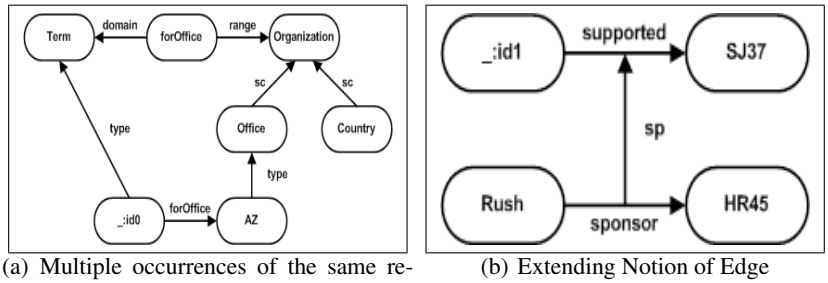
- $W = \{w : w \in univ(D)\}$ is the set of nodes.
- $E = \{e_i : 1 \leq i \leq |D|\}$ is the set of hyperarcs.
- $\rho : W \times E \rightarrow \{s, p, o\}$ is the role function of nodes w.r.t. hyperarcs. Let $t \in D$ be an RDF triple, $e \in E$ an hyperarc, and $w \in W$ a node such that $w \in head(e) \cup tail(e)$. Then the following must hold:
  - $(\rho(w, e) = s) \Leftrightarrow (w \in tail(e)) \wedge (w \in sub(\{t\}))$
  - $(\rho(w, e) = p) \Leftrightarrow (w \in tail(e)) \wedge (w \in pred(\{t\}))$
  - $(\rho(w, e) = o) \Leftrightarrow (w \in head(e)) \wedge (w \in obj(\{t\}))$

The *Bhyper* representation reduces space complexity to store the RDF document and speeds up the data recovery process. To illustrate the benefits of the *Bhyper*-based representation and the main drawbacks of the traditional graph-based representation, we use some examples extracted from the US Congress bills voting process dataset.

First, consider the RDF document D_1 ={*(_:id0, type, Term), (_:id0, forOffice, AZ), (AZ, type, Office), (Office, subClassOf, Organization), (Country, subClassOf, Organization), (forOffice, range, Organization), (forOffice, domain, Term)*}, where the resource *forOffice* occurs as a predicate and a subject. This situation can be modeled by allowing multiple occurrences of the same resource in the resulting labeled directed graph, as arcs or nodes labels (Figure 3(a)). However, this violates one of the most important aspects of graph theory: the intersection between the nodes and arcs labels must be empty.

Second, a predicate may relate other predicates in an RDF document. For example, in the RDF document D_2 = {*(Rush, sponsor, HR45), (_:id1, supported, SJ37), (sponsor, subPropertyOf, supported)*} the predicate *subPropertyOf* relates the predicates *sponsor* and *supported*. This situation can be modeled extending the notion of arc by allowing the connection between arcs (Figure 3(b)). However, the resulting structure is not a

graph in the mathematical sense, because the set of arcs must be a subset of the Cartesian product of the set of nodes. Since these two simple situations violate some of the graph constraints, it is not possible to use concepts and search algorithms of graph theory to manipulate RDF documents. Thus, while the labeled directed graph model is the most widely used representation, it cannot be considered a formal model for RDF [7].



(a) Multiple occurrences of the same resource

(b) Extending Notion of Edge

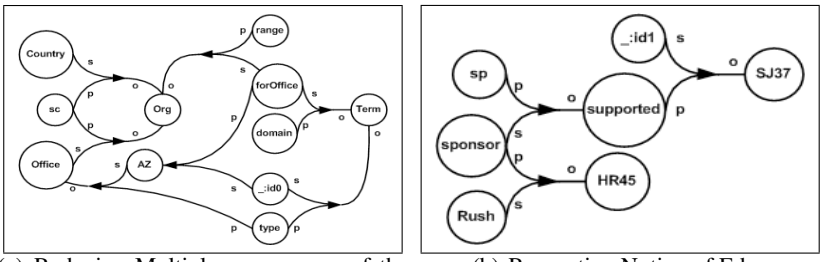
Fig. 3. RDF document properties

Figures 4(a) and 4(b) show the RDF directed hypergraphs representing RDF documents D_1 and D_2, respectively. In *Bhyper*, given an RDF document D, each node corresponds to an element $w \in univ(D)$. Thus, the information is only stored in the nodes, and the hyperarcs only preserve the role of each node and the concept of direction of RDF graphs. An advantage of this representation is that each resource (subject, property, or value) is stored only once, and the space required to store an RDF document is reduced if a resource appears several times in the document. In this way, the space complexity of our approach is lower than the complexity of the graph-based RDF representation. Besides, concepts, techniques, and algorithms of hypergraph theory can be used to manipulate RDF documents more efficiently.

The *Bhyper* indices were implemented in Prolog by using two extensional predicates: *subject(S,P,Lo)* and *object(O,P,Ls)*. The predicate *subject* associates a given subject value S with the property P that relates it with the object values in the list Lo. Similarly, the predicate *object* maps an object value O with the property P that relates it with the subject values in the list Ls. Both predicates are indexed on the first and second arguments with the SWI-Prolog indices. OneQL predicates *subject* and *object* resemble the property tables implemented in Jena2 to speed up queries over the same subject or object values [32].

6 Experimental Results

We conducted an experimental study to empirically analyze the effectiveness of the OneQL optimization and evaluation techniques. We report on the evaluation time performance of bushy plans comprised of groups and identified by our proposed query optimizer.



(a) Reducing Multiple occurrences of the same resource

(b) Respecting Notion of Edge

Fig. 4. The *Bhyper*-based representation

Dataset and Query Benchmark: We use the real-world dataset on US Congress vote results of the 2004 bills voting process described in Figure 5(b). The entire dataset was downloaded and locally stored in flat files; the total size is 3.613 MB and 67,392 triples. We considered two sets of queries. Benchmark one is a set of nine queries which are described in Figure 5(a) in terms of the number of patterns in the WHERE clause and the answer size; all the queries have at least one pattern whose object is instantiated with a constant. Benchmark two is a set of 60 queries which have between one and seven GJoin(s) among small size groups of patterns and have more than 12 triple patterns. These two benchmarks are published in http:www.ldc.usb.ve/˜mvidal/OneQL/datasets.

Evaluation Metrics: We report on runtime performance, which corresponds to the *user time* produced by the *time* command of the Unix operation system. OneQL was implemented in SWI-Prolog (Multi-threaded, 64 bits, Version 5.6.54). The randomized optimizer was run for 20 iterations at an initial temperature of 700. The experiments were evaluated on a Solaris machine with a Sparcv9 1281 MHz processor and 16GB of RAM.

query	#patterns	answer size
q1	4	3
q2	3	14033
q3	7	3908
q4	4	14868
q5	4	10503
q6	4	47
q7	3	6600
q8	3	963
q9	7	13177

(a) Benchmark One Query Set

property	# triples	# values subject	# values object
voter	21600	21600	100
winner	216	216	2
hasBallot	21600	216	21600
option	21600	21600	3
title	216	216	216

(b) Cardinality and number of values govtrack.us 2004

Fig. 5. Experiment Configuration Set-Up

6.1 Predictive Capability of the Hybrid Cost Model

We studied the predictive capability of the OneQL hybrid cost model. We generated 190 different bushy tree plans for a query with four patterns (Figure 5(a)), and computed the estimated evaluation time using the OneQL hybrid cost model. Additionally, we executed the 190 plans in the OneQL query engine and measured the evaluation time in terms of the total number of inferences. Figure 6 plots the actual versus the estimated evaluation costs; we can observe a positive trend between the estimated and actual cost, and a correlation between both costs of 0.76. Both results indicate that there is a linear relation between the estimated and the actual costs, and they suggest that the OneQL hybrid cost model is able to predict the runtime performance of the OneQL query engine.

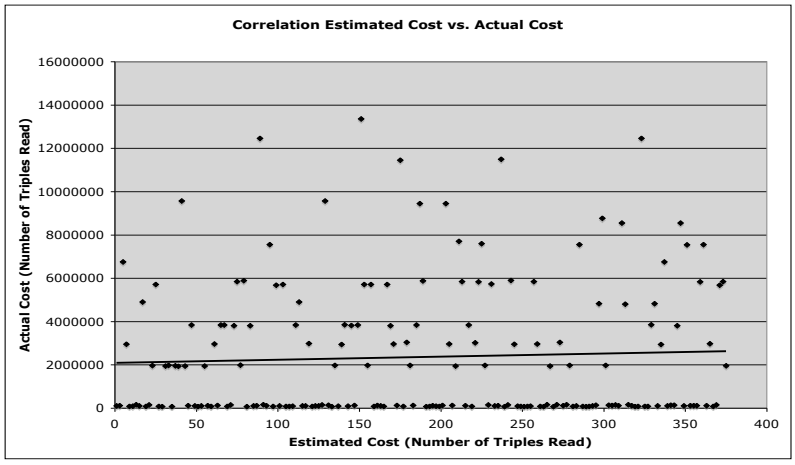


Fig. 6. Correlation actual cost vs. estimated cost

6.2 Effectiveness of the OneQL Optimization Techniques

We studied the effectiveness of the OneQL optimization techniques by empirically analyzing the quality of the optimized plans w.r.t. the rest of the plans of the corresponding query, and the runtime performance of the optimized plans.

To analyze the quality of the optimized plans, we generated all the plans for queries in benchmark one with three and four patterns, and computed the percentile in which the optimal plan falls. The average percentile is 97 and the lowest is 92. These results

indicate that the optimizer is able to identify execution plans that are at least better than 92% of the execution plans of the query.

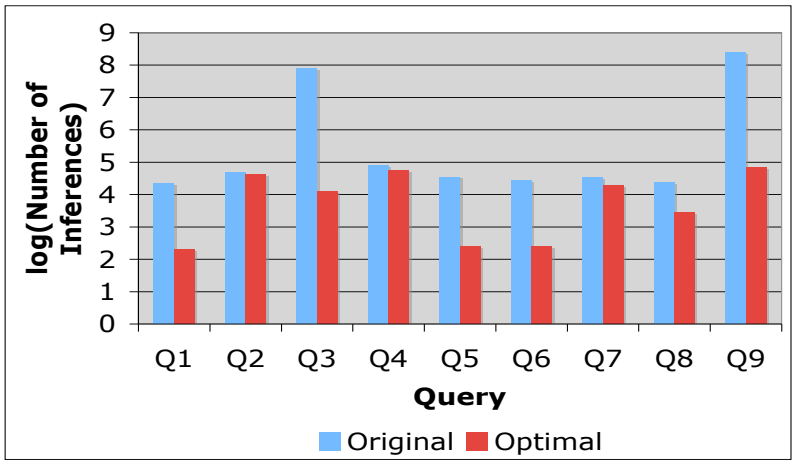


Fig. 7. Cost of Original vs. Optimal Queries (Inferences-logarithmic scale)

We also report on the runtime performance of the optimized queries. Figure 7 compares the number of inferred triples of the non-optimized and optimized versions of benchmark one in logarithmic scale. In general, we can observe that the optimized query has a significantly lower cost than the original query, speeding up the evaluation time in some cases by more than one order of magnitude. Plans with the most significant performance improvements correspond to bushy trees, and they are comprised of *Group* joins with small size groups.

6.3 Effectiveness of the OneQL Physical Operators

We have conducted an empirical analysis on the benefits of the evaluation techniques implemented on OneQL, and have executed 60 queries of benchmark two. Figure 8 compares the evaluation time (logarithmic scale) of the queries comprised of Group Joins (GJoin) against queries with Index Nested Loop Joins (Njoin). We can observe that the plans composed of GJoins overcome the Njoin plans by at least one order of magnitude when the GJoins are comprised of small size groups and low join selectivity.

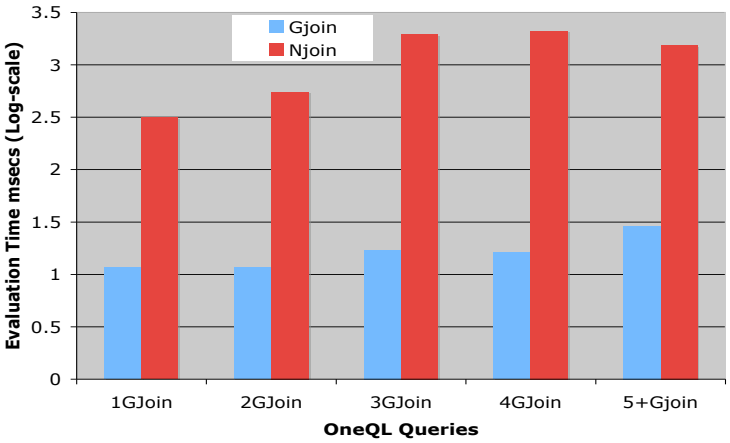


Fig. 8. Performance of the OneQL Physical Operators

6.4 Effectiveness of the OneQL Bhyper-based representation

Figure 9 compares the evaluation cost of twenty queries of benchmark two in logarithmic scale, when the *Bhyper* structures are used to index the RDF data, and when the *Bhyper* structures are not used to index the RDF data. The results indicate that the indices improve the performance of the physical operators, and the evaluation time is reduced by up to two orders of magnitude in queries comprised of a large number of GJoins composed of groups of instantiated triples.

6.5 Effectiveness of the OneQL Optimization and Evaluation Techniques

Finally, we studied the benefits of the optimization and evaluation techniques implemented by OneQL by empirically analyzing the quality of the OneQL optimized plans w.r.t. the plans optimized by the RDF-3X query optimizer. Queries of benchmark one were optimized by OneQL and RDF-3X and the generated plans were run in OneQL with and without *Bhyper* indices. Each RDF-3X optimized plan was run using the *GJoin* and *NJoin* operators to evaluate the groups in the bushy plans. Figure 10 reports the evaluation time (logarithmic scale) of these combinations of queries. We can observe that *Bhyper*-based representation is able to speed up the evaluation time of all the versions of the queries comprised of instantiated triples. Second, the evaluation time of the OneQL and RDF-3X optimized plans are competitive, except for queries *q1* and *q6* where OneQL was able to identify plans where all the triples are instantiated, and the most selective ones are evaluated first. These results indicate that the OneQL optimization and evaluation techniques may be used in conjunction with the state-of-the-art

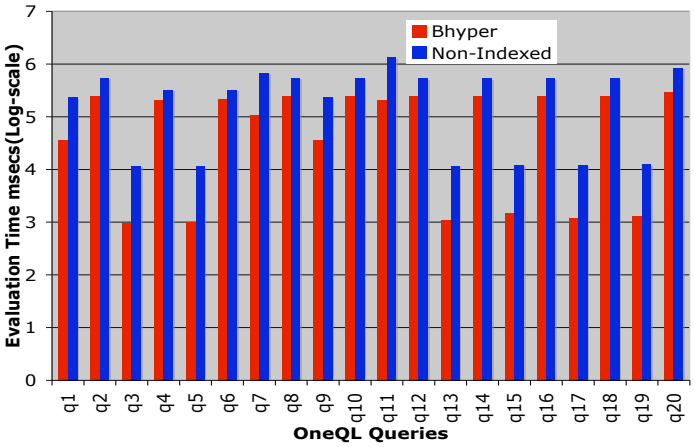


Fig. 9. Performance of the OneQL Bhyper-based index representation

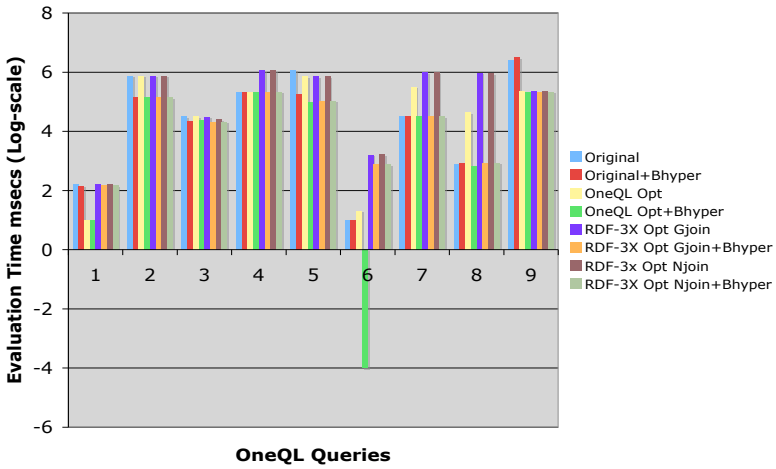


Fig. 10. Performance of the OneQL Optimization and Evaluation Techniques

techniques to provide more efficient query engines; they have encouraged us to develop our physical operators in existing RDF engines. So far, we have implemented the *GJoin* operator in the Jena engine, and we have observed in initial experiments that our *GJoin* implementation outperforms the evaluation time by up to three orders of magnitude. In the future, we also plan to implement these techniques in RDF-3X and conduct a more exhaustive empirical study to corroborate the effects of the developed techniques.

7 Conclusions

We have presented the OneQL system for efficiently evaluating SPARQL queries. We have addressed the challenges of scaling up to large RDF documents and complex SPARQL queries. We report on the results of our optimization and evaluation techniques for SPARQL queries. Then, we describe a *Bhyper*-based representation for RDF documents that reduces the space and time complexity of the tasks of storing and querying RDF documents. In the future, we plan to enhance the hybrid cost model with Bayesian inference capabilities to consider correlations between the different patterns that can appear in a SPARQL query; implement our operators in existing SPARQL query engines; and finally, extend the set of physical operators to better exploit the properties of the *Bhyper*-based representation.

8 Acknowledgments

This research has been partially supported by the DID-USB and the Proyecto ALMA Mater-OPSU. The authors are very grateful to Eduardo Ruiz for his programming support.

References

1. D. J. Abadi, A. M. 0002, S. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB J.*, 18(2):385–406, 2009.
2. D. J. Abadi, A. M. 0002, S. Madden, and K. J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*, pages 411–422, 2007.
3. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.
4. AllegroGraph. http://www.franz.com/agraph/allegrograph/.
5. P. Auillans, P. O. de Mendez, P. Rosenstiehl, and B. Vatant. A Formal Model for Topic Maps. In *Proceedings of the Third International Semantic Web Conference (ISWC 2004)*, 2002.
6. G. Benson. Editorial. *Nucleic Acids Research*, 35(Web-Server-Issue):1, 2007.
7. F. Dau. RDF as Graph-Based, Diagrammatic Logic. In *Proceedings of the 16th International Symposium on Methodologies for Intelligent Systems (ISMIS 2006)*, 2006.
8. G. Fletcher and P. Beck. Scalable Indexing of RDF Graph for Efficient Join Processing. In *CIKM*, 2009.
9. G. Gallo, G. Longo, S. Pallottino, and S. V. Nguyen. Directed Hypergraphs and Applications. In *Discrete Applied Mathematics*, 2003.
10. G. Gallo and M. G. Scutella. Directed Hypergraphs as a Modelling Paradigm. In *Tech. Rep. TR-99-02, Universita di Pisa*, 1999.

11. M. Y. Galperin. The Molecular Biology Database Collection: 2007 update. *Nucleic Acids Res*, 35(Database issue), January 2007.
12. M. Y. Galperin. The Molecular Biology Database Collection: 2008 update. *Nucleic Acids Res*, 36(Database issue):D2–D4, Jan 2008.
13. A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *ISWC/ASWC*, pages 211–224, 2007.
14. G. Ianni, T. Krennwallner, A. Martello, and A. Polleres. A Rule System for Querying Persistent RDFS Data. In *Proceedings of the 6th European Semantic Web Conference (ESWC2009)*, Heraklion, Greece, May 2009. Springer. Demo Paper.
15. The JenaOntology Api. http://jena.sourceforge.net/ontology/index.html.
16. Jena TDB. http://jena.hpl.hp.com/wiki/TDB.
17. R. Lipton and J. Naughton. Query Size estimation by adaptive sampling (extended abstract). In *Proceedings of SIGMOD*, 1990.
18. A. Martinez and M. Vidal. A Directed Hypergraph Model for RDF. In *KWEPSY*, 2007.
19. J. McGlothlin and L. Khan. RDFJoin: A Scalable of Data Model for Persistence and Efficient Querying of RDF Dataasets. In *VLDB*, 2009.
20. T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.
21. R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
22. E. Ruckhaus, E. Ruiz, and M. Vidal. Query Evaluation and Optimization in the Semantic Web. In *Proceedings ALPSWS2006: 2nd International Workshop on Applications of Logic Programming to the Semantic Web and Semantic Web Services*, 2006.
23. E. Ruckhaus, E. Ruiz, and M. Vidal. OnEQL: An Ontology Efficient Query Language Engine for the Semantic Web. In *Proceedings ALPSWS2007*, 2007.
24. E. Ruckhaus, E. Ruiz, and M. Vidal. Query Evaluation and Optimization in the Semantic Web. *TPLP*, 2008.
25. R. Sacks-Davis, T. D. J. A. Thom, and J. Zobel. Indexing documents for queries on structure, content, and attributes. In *Proceedings of the International Conference on Digital Media Information Bases*, 1997.
26. P. Selingerl, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. *Proceedings of ACM Sigmod*, 1979.
27. L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.
28. M. Stoker, A. Seaborne, A. Bernstein, C. Keifer, and D. Reynolds. SPARQL Basic Graph Pattern Optimizatin Using Selectivity Estimation. In *WWW*, 2008.
29. G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory Pract. Log. Program.*, 8(2):129–165, 2008.
30. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.
31. J. Wielemaker. An Optimised Semantic Web Query Language Implementation in Prolog. In *ICLP*, pages 128–142, 2005.
32. K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds, and J. Database. Efficient RDF Storage and Retrieval in Jena2. In *EXPLOITING HYPERLINKS 349*, pages 35–43, 2003.

Scalable RDF query processing on clusters and supercomputers

Jesse Weaver and Gregory Todd Williams

Rensselaer Polytechnic Institute, Troy, NY, USA
{weavej3,willig4}@cs.rpi.edu

Abstract. The proliferation of RDF data on the web has increased the need for systems that can query these data while scaling with their growing size and number. We present an application of parallel hash-joins for basic graph pattern matching over large amounts of RDF designed for shared nothing architectures including high-performance clusters and the Blue Gene/L. Our approach does not require any pre-processing of the RDF data or costly index building. Rather, we rely on a cluster's high bandwidth and fast memory to load and query data in parallel and in near-real time. We present an initial evaluation of our algorithm showing competitive results on clusters of up to 1,024 processors.

1 Introduction

The web has recently seen a proliferation of structured data. RDF data is now available from many sources across the web relating to a huge variety of topics. Examples of these RDF datasets include the Billion Triples Challenge[1] dataset (collected by a webcrawler from RDF documents available on the web), the Linking Open Data project[2] (in which a number of independent datasets are linked together using common URIs), and the recent conversion[3] of the data.gov[4] dataset to RDF.

With such a large and growing availability of RDF data, new and more efficient ways of querying these data are needed. While most existing systems rely on common database indexing techniques to allow fast retrieval of RDF data, the time required to load and index the data can be prohibitive. In this paper, we present a system for RDF query answering on clusters that does not require any pre-processing, global indexing, or particular assignment of RDF triples to processors. Our system is designed for use on shared-nothing clusters that can range from simple Beowulf clusters to the IBM Blue Gene/L supercomputer.

By utilizing a cluster's parallelism, our system is able to load and query a large dataset much more quickly than traditional approaches. After data is loaded, our system makes use of a parallel hash-join to answer basic graph

[1] http://challenge.semanticweb.org/
[2] http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData/
[3] http://data-gov.tw.rpi.edu/
[4] http://www.data.gov/

pattern queries. Hash-join is a join algorithm that derives its efficiency from partitioning data based on a hash value. We base our work on an existing parallelization of the hash-join algorithm [1]. Our join implementation utilizes an on-the-fly conversion between RDF node values and locally-unique node identifiers to allow efficient join processing without requiring global node identifiers. We differ from most previous work with parallel hash-joins by assuming the presence of a high performance cluster with enough system memory (between all processors) to keep the entire RDF dataset and all intermediate results in memory.

The architecture of our system allows for very fast querying of a dataset. Since our system never pre-processes input RDF data, this speed enables ad-hoc querying with the ability to add and remove arbitrary amounts of data in subsequent queries with little to no cost. We evaluate our system with several existing datasets on a Linux-based AMD Opteron cluster ranging from 2 to 128 processors, and on a Blue Gene/L from 32 to 1,024 processors.

The rest of this paper is organized as follows. Section 2 reviews related work on parallel hash-join algorithms and other approaches to processing RDF data in distributed and parallel environments. Section 3 presents specific details of our parallel hash-join implementation including parallel loading and indexing of RDF data, hash-based distribution and joining of intermediate results. Section 4 presents an evaluation of our system using several existing RDF datasets and queries. Finally, Section 5 concludes the paper and discusses possible future work in extending our system for reasoning and support for more complex queries.

2 Related Work

Other works on parallel and/or distributed RDF query processing include RDFPeers [2], Continuous RDF Query Processing over DHTs [3], YARS2 [4], Virtuoso[5] [5, 6], GridVine [7], Clustered TDB [8], and 4store[6]. While works like Marvin [9, 10], parallel OWL inferencing [11], and parallel RDFS inferencing [12] use parallelism for semantic web reasoning, they are not directly comparable to the system we present in this paper since we focus on RDF query and not inferencing.

RDFPeers creates a distributed RDF repository over a multi-attribute addressable network (MAAN) [13]. Triples are stored as three attribute-value pairs (subject=..., predicate=..., object=...) on three nodes based on hash values generated from the subject, predicate, and object. RDFPeers provides a query language which maps to MAAN's multi-attribute range queries allowing for distributed querying. [3] focuses on "continuous evaluation of conjunctive triple pattern queries over RDF data stored in distributed hash tables," and GridVine is also a DHT approach. These approaches do not address parallelism and thus differ from our work.

[5] http://virtuoso.openlinksw.com/
[6] http://4store.org/

YARS2, Clustered TDB, Virtuoso (cluster edition), and 4store provide support for RDF stores on clusters. The Clustered TDB work discusses several forms of parallelism: inter-query (running more than one query in parallel), intra-query (running subqueries in parallel and pipelining operators), and intra-operation (distributing single operations for concurrent execution). YARS2 provides fine-grained intra-operation parallelism in triple-pattern matching. The details of Virtuoso and 4store are less certain to us since the finer details of these stores' query evaluation techniques are not published to our knowledge. We differ from these approaches in that we address parallel query processing as a process involving both the loading and computation over RDF data rather than a process occurring over a persistent storage system. For clarity, the system presented herein is not interactive. Queries are queued for evaluation, and our system executes once in its entirety for each query.

Finally, in [1], DeWitt and Gerber show an extension of the hash-join to a multiprocessor environment, and demonstrate its effectiveness in parallel join execution. Our work is based heavily on this extension with two notable exceptions. We restrict our work to all in-memory environments, avoiding the need for variants of the hash-join such as Grace and Hybrid hash-joins that address optimizations in the presence of limited memory. Moreover, every node in our system acts as both a partitioning processor and a joining processor, allowing a join to utilize all available processing power.

3 Methodology

We implement our system in C using the Message Passing Interface[7] (MPI) for interprocessor communication. Each processor maintains an in-memory triple store consisting of three indexes that can directly answer any triple pattern.

In this section we assume the reader is familiar with the following notation from SPARQL[8]. A solution mapping μ is a partial function from variables to RDF terms, and a set of results from a query is a multiset, Ω, of solution mappings. An example solution mapping with two variable bindings might look like {name="Alice", email=<mailto:alice@work.example>}.

3.1 Parallel Hash-Join

In our parallel hash-join implementation, two subqueries are executed independently on each processor i, regardless of the dataset held locally on each processor. The results of these subqueries ($\Omega_{1,i}$ and $\Omega_{2,i}$) are then redistributed among the processors in such a way as to ensure that the appropriate results for the join are colocated. This is done by hashing on the values of the variables shared between the two result sets. For example, if the results of the two subqueries join on variables $?a$ and $?b$, then for each solution mapping μ in the results,

[7] http://www.mpi-forum.org
[8] http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/#initDefinitions

we hash on the values of $?a$ and $?b$ in μ, and based on that hash value, μ is sent to the appropriate processor. Therefore, solution mappings with the same terms bound to $?a$ and $?b$ will have the same hash value and will get sent to the same processor. After distributing the results, each processor performs the join locally on the received results ($\Omega'_{1,i}$ and $\Omega'_{2,i}$). The redistribution is illustrated in Algorithm 1, while the overall parallel hash-join is illustrated in Algorithm 2 (assuming $\Omega_{1,i}$ and $\Omega_{2,i}$ are available as input after executing the two subqueries). In Algorithm 2, we use "**pardo**" to mean "do in parallel." (For clarity, note that it is allowed for a processor to "send" a solution mapping to itself on line 5 of Algorithm 1. This is a logical description of the algorithm; the implementation of this algorithm may handle such sends as a special case.)

For the query evaluation of a basic graph pattern containing n triple patterns, the parallel hash-join algorithm is run $n - 1$ times, joining the triple patterns in a so-called left-deep query execution plan. The union on line 6 of Algorithm 2 represents the logical, complete results of the join. During basic graph pattern evaluation, however, instead of performing this union, each $\Omega'_{1\bowtie 2,i}$ simply becomes the input $\Omega_{1,i}$ for the subsequent join with the results from the next triple pattern in the query execution plan.

Algorithm 1: Distribute Solution Mappings (distmu)

Input: A (local) multiset of solution mappings Ω_i, a set of join variables V, and a number of processors p.
Output: A multiset of solution mappings Ω'_i from redistribution.

```
1  Ω'_i = ∅
2  foreach μ ∈ Ω_i do
3      μ' = project(V, μ)
4      recvr = hash(μ') % p
       // Send μ to recvr.
5      send(recvr, μ)
       // Receive solution mappings from any processor.
6      while recv(∗, μ_r) do
7          add μ_r to Ω'_i
8      end
9  end
10 return Ω'_i
```

3.2 Parallel RDF I/O

We utilize the same approach to loading RDF data in parallel as in [12]. Our only requirement on the input data is that it be in syntax similar to N-Triples[9]. We say similar to the N-Triples syntax because we do not require the data to

[9] http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/#ntriples

Algorithm 2: Parallel Hash Join

Input: Two multisets of solution mappings $\Omega_1 = \bigcup_{i=0}^{p-1} \Omega_{1,i}$ and $\Omega_2 = \bigcup_{i=0}^{p-1} \Omega_{2,i}$, the set of join variables V, and a number of processors p.

Output: A multiset of solution mappings $\Omega_{1 \bowtie 2} = join(\Omega_1, \Omega_2)$.

```
   // Loop indicates parallelism where i is the rank of the processor.
1  for i = 0 to p − 1 pardo
       // Ensure solution mappings that can join meet on same processor.
2      Ω'_{1,i} = distmu(Ω_{1,i}, V, p)
3      Ω'_{2,i} = distmu(Ω_{2,i}, V, p)
       // Join local solution mappings.
4      Ω'_{1⋈2,i} = join(Ω'_{1,i}, Ω'_{2,i})
5  end
6  return ⋃_{i=0}^{p−1} Ω'_{1⋈2,i}
```

be encoded in 7-bit US-ASCII. The simple format of these N-Triples-like files make parallel reading of the data trivial. Each processor is assigned—in rank-order—a chunk of the input file to read; that is, the i^{th} processor reads the i^{th} consecutive chunk of data. The chunk of data may begin and/or end in the middle of a triple. To handle this, each processor of rank i simply sends the fragment at the beginning of its chunk to processor with rank $i-1$. Processor i then receives such a fragment from processor $i+1$ and concatenates the triple fragment to the end of its chunk of data. Then, every processor has a set of complete triples which it loads locally into an indexed, in-memory store, converting the serialized RDF nodes into 64-bit identifiers and holding in memory a map for converting between the two (which we will refer to as the nodemap). Unlike most traditional databases and much like many RDF stores, the indexes themselves are the data; there are no data tables holding additional information. Note that while we index the local data on each processor, there is no global index for the entirety of the data (that is, an index over all the data distributed across all processors). This is discussed in the following subsection.

At the end of the query, each processor writes out its local set of solutions (the last $\Omega_{1 \bowtie 2,i}$) to its own file using RDF node values (as opposed to the local identifiers). Therefore, our entire query process starts with N-Triples-like files and ends with results containing full RDF node values.

3.3 Communicating Solution Mappings

As mentioned in the previous section, no global indexes are created at any point of the query evaluation. Each processor holds its local triples as 64-bit identifiers and a nodemap. From lines 6 and 7 of Algorithm 1, the solution mappings must be communicated in a way that is meaningful to all processors. This is done by converting the 64-bit identifiers back into string representations before sending the solution mapping to another processor. This allows the receiving processor

to assign its own local identifier to the RDF node. While this may incur a higher communication cost, it saves greatly on loading time. Generating, distributing, managing, and performing lookups on global 64-bit identifiers is an extremely time-consuming process, one which we found to be prohibitive.

We note that this approach makes loading data inexpensive enough that we can afford to load data for every query evaluation. This allows our system to take advantage of the up-to-date state of the data without costly index maintenance.

While assigning local identifiers, we take advantage of a simple optimization. During line 2 of Algorithm 2, as results are received from the left-hand side of the join (from $\Omega_{1,i}$ of the sending processors into $\Omega'_{1,i}$ of the receiving processors), a processor assigns new local identifiers to RDF nodes from received solution mappings, placing the new identifiers in a new nodemap. Then, during line 3, as results are received from the right-hand side of the join (from $\Omega_{2,i}$ of the sending processors into $\Omega'_{2,i}$ of the receiving processors), the RDF terms bound to the join variables in the solution mappings are checked for local identifiers in the nodemap generated from the left-hand side of the query. For each solution mapping, if there is no local identifier assigned to one of its join variables' RDF terms, then we can be certain that there are no results from the left-hand side to which the solution mapping can join. In this case, we can eliminate the solution mapping immediately. Otherwise, if local identifiers exist for all the RDF terms bound to join variables, then the remaining (non-join) variables' RDF terms in the solution mappings are also added to the nodemap. In essence, this simply allows us to eliminate results from the right-hand side without actually attempting the join. This is similar to the effect of the use of bit vector filtering in [1].

4 Evaluation

We evaluated our system on a high performance cluster and a Blue Gene/L supercomputer at Rensselaer Polytechnic Institute's Computational Center for Nanotechnology Innovations[10] (CCNI). Each node of the CCNI high performance Opteron cluster is an IBM LS21 blade server running RedHat Workstation 4 Update 5 with two dual-core 2.6 GHz AMD Opteron processors with gigabit ethernet and InfiniBand interconnects. We ran tests on up to 128 processors on medium-memory nodes, each of which has 12GB of system memory. Our testing on the CCNI Blue Gene/L was performed on up to 1,024 nodes, each of which has two 700-MHz PowerPC 440 processors and 512–1024MB of system memory. We utilize three of the Blue Gene/L's specialized hardware networks: a 175MBps 3-dimensional torus for point-to-point communication, a 350MBps global-collective network, and a global barrier network.

We read and write files to/from the large General Parallel File System[11] (GPFS) which has a block size of 1024 KB, scatter block allocation policy, and 256 KB RAID device segment size using a RAID5 storage system.

[10] http://www.rpi.edu/research/ccni/
[11] http://www-03.ibm.com/systems/clusters/software/gpfs/index.html

We evaluate query performance using the Lehigh University Benchmark [14] 20-university dataset (LUBM(20,0)) and on the Barton dataset[12] using queries introduced in [15]. LUBM datasets are synthetically generated datasets containing information about universities. Since LUBM is well-known and widely evaluated against, we provide an evaluation on a LUBM dataset to allow for comparisons with other systems. After generating LUBM(20,0), we used the work from [12] to produce the RDFS closure so as to make the standard LUBM queries meaningful. (For example, for LUBM query 6, no results will be returned unless inferencing is performed to derive that, e.g., all graduate students are students.) The RDFS closure of LUBM(20,0) has 5,159,292 triples. The Barton dataset is an RDF formatted version of the MIT Libraries Barton catalog, and contains 51,598,374 triples.

Much performance tuning can be done by tweaking parameters that affect how Algorithm 1 sends and receives solution mappings. Such parameters include the ratio of transient send messages to transient receive messages and also the frequency at which the processors collaborate to determine whether they have finished distributing solution mappings (a costly operation). The Blue Gene/L has a more sensitive network in that a high number of transient messages can cause the system to effectively fail (ultimately due to memory limitations), and so we set the number of allowable transient messages on the Blue Gene/L lower than on the Opteron cluster. The Blue Gene/L also has an optimized collective network allowing for processors to collaborate to determine termination of Algorithm 1 at a lower cost, and thus we allow the Blue Gene/L to check more frequently for termination than on the Opteron cluster. In our experience, these tuning parameters greatly affect performance and scaling characteristics, and for this evaluation we have tuned them according to personal experience. However, we have yet to optimize these parameters, and so better performance may be possible.

All figures discussed below use logarithmic axes for both time and number of processors.

Figures 1 through 4 show performance of four of the LUBM queries on the Opteron cluster scaling from 2 to 16 processors, and in general, they show scaling of loading time and total time with respect to the number of processors. Only query two in Figure 1 shows an increase in query time from 8 to 16 processors. This is likely because query two is the only query of the four that requires a high number of joins and does not restrict results to data from a specific university. Queries three and four have a bound term that—by the nature of LUBM data—restricts results to data from "University0", and query six has only a single triple pattern (no joins).

The Barton dataset is roughly ten times the size of the LUBM(20,0) RDFS closure, and so more memory is needed to perform the evaluation. Figure 5 shows the query execution of Barton query 7 on 32 to 128 processors. The loading time decreases greatly as the number of processors increases, but the query time increases after 64 processors.

[12] http://simile.mit.edu/rdf-test-data/barton/

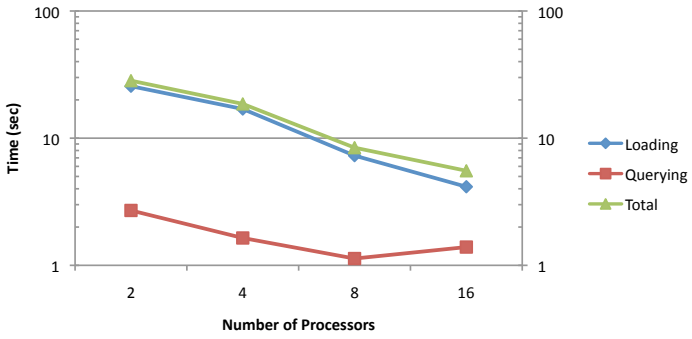


Fig. 1. LUBM(20,0) Query 2 evaluation on Opteron cluster

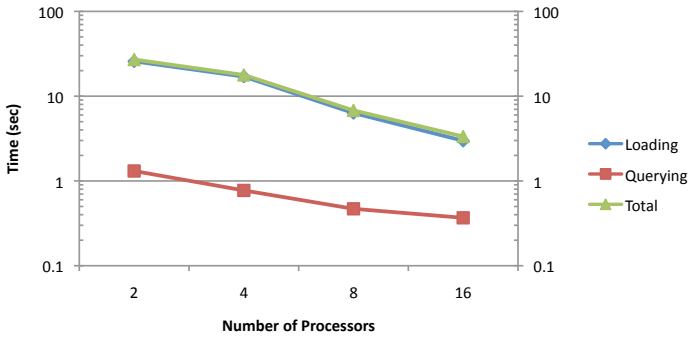


Fig. 2. LUBM(20,0) Query 3 evaluation on Opteron cluster

In Figure 6, we also show query execution of LUBM query 3 on the LUBM(20,0) RDFS closure using the Blue Gene/L ranging from 32 to 1024 processors. Clearly, loading time scales linearly, but the query time increases after 128 processors.

We notice from Figures 1, 4, and 6 that there seems to be a "sweet spot" for query time only (excluding loading time). Further tweaking of the aforementioned parameters have shown that we can adjust the characteristics of the "sweet spot," but often at a cost. We believe that tuning the parameters based on the number of processors will provide better scaling, and such is left as future work.

Our system competes well with state-of-the-art RDF query systems in loading and query times. Anecdotal evidence indicates that Virtuoso provides the fastest RDF loading time of any RDF store at 110,532 triples-per-second on eight processors[13]. We report in Figure 3 a loading rate of 820,117 triples per second on eight processors, and we achieve a maximum loading rate of 3,088,172

[13] http://www.openlinksw.com/weblog/oerling/index.vspx?page=&id=1562

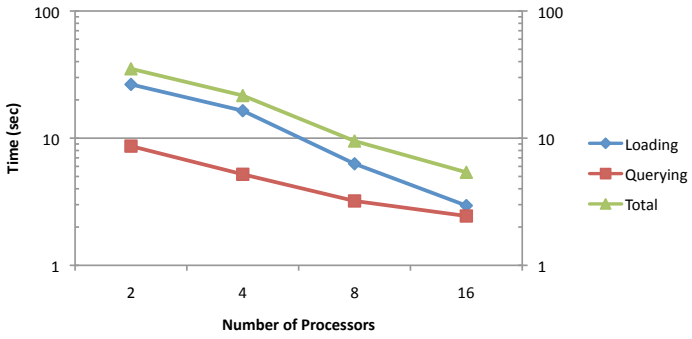


Fig. 3. LUBM(20,0) Query 4 evaluation on Opteron cluster

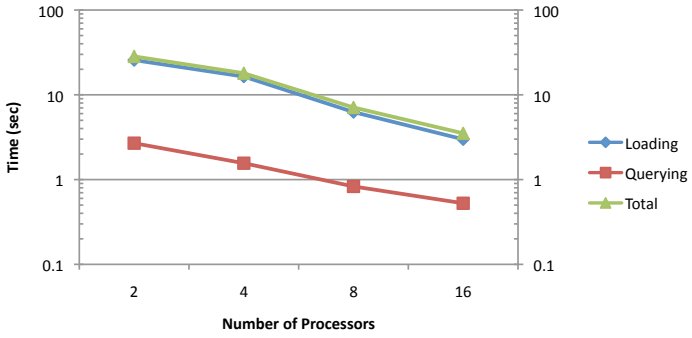


Fig. 4. LUBM(20,0) Query 6 evaluation on Opteron cluster

triples per second on 1024 processors on the Blue Gene/L. RDF-3X [16] seems to be the state-of-the-art in query times. It is difficult to compare our query times to theirs since most of the Barton queries that they use for evaluation use non-standard SPARQL features (e.g., aggregation, "duplicates" keyword, "in" operator) and filters, features which we do not currently support. Therefore, we compare only Barton query 7, the single query that we both support. RDF-3X evaluates Barton query 7 in 32.61 seconds with cold caches (dropping to 1.26 seconds after five runs), whereas we perform the same query in 23.75 seconds on 64 processors. We emphasize, though, that RDF-3X requires 13 minutes to load the Barton dataset after an unreported amount of pre-processing time, whereas our total time (loading and querying) is at lowest 49.92 seconds on 64 processors and 47.37 seconds on 128 processors.

Our system is capable of loading roughly 1.25 million triples from the tested datasets per 1GB of RAM. This total includes storing the full nodemap as well as the three covering indexes. We note that we have spent no time attempting to improve this storage density, but our system should be able to take advantage of

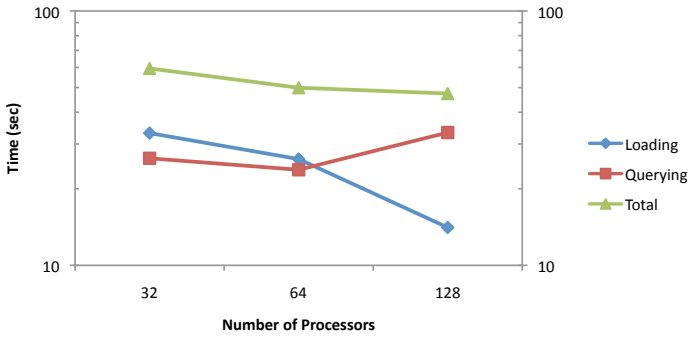


Fig. 5. Barton Query 7 evaluation on Opteron cluster

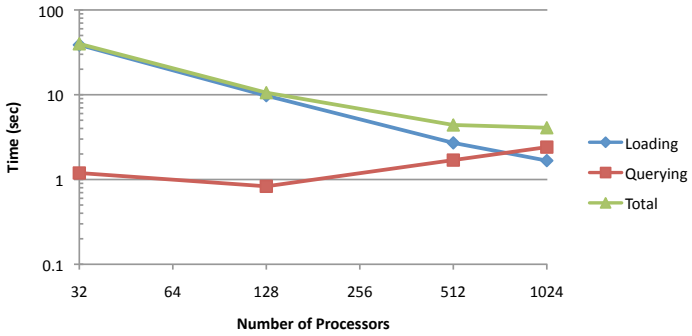


Fig. 6. LUBM(20,0) Query 3 evaluation on Blue Gene/L

compression techniques such as those discussed in [16], significantly improving storage density. While storage density is obviously a concern for an in-memory system like ours, we also note that the primary limitation we faced was not available memory but job queuing time on both the Opteron cluster and Blue Gene/L (both heavily used systems).

5 Conclusion and Future Work

In this paper we have presented a system for answering basic graph pattern queries over large RDF datasets on clusters. Our evaluation has shown our system to be competitive with more traditional indexed, persistent triple stores without the need for expensive pre-processing, loading, or global indexing of the data. Our results show that some datasets and queries exhibit a "sweet spot" for optimal execution dependent on the number of processors and tuning parameters while others show total time of loading data and query evaluation speed can scale with a constant factor as the number of processors increases.

There are many areas where our system can be improved. Beyond further evaluation and tuning on both the Opteron cluster and Blue Gene/L, we hope to pursue some of the following areas in future work. Currently our system only handles basic graph patterns, but a natural extension would include optional patterns, named graphs, and filters. In addition, the ability to distribute results in our system is ideally suited to answering aggregate queries, a feature we hope to implement.

Our current hash-join implementation seems to perform well on selective queries, but can have trouble with unselective queries or triple patterns. We are currently investigating a second parallel join algorithm to address queries with unselective triple patterns. We are also pursuing evaluation on larger datasets such as the RDFS closure of LUBM(10000,0) (containing roughly 2.4 billion triples) and the Billion Triples Challenge 2009 dataset. Finally, we hope to integrate the work presented in [12] with our system to allow parallel inferencing to occur during query evaluation.

Acknowledgements. We thank Gunnar AAstrand Grimnes for his insightful comments on this work.

References

1. DeWitt, D.J., Gerber, R.H.: Multiprocessor Hash-Based Join Algorithms. In: Proceedings of the 11th International Conference on Very Large Data Bases. (1985) 151–164
2. Cai, M., Frank, M.R.: RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In: Proceedings of the 13th International World Wide Web Conference. (2004) 650–657
3. Liarou, E., Idreos, S., Koubarakis, M.: Continuous RDF Query Processing over DHTs. In: Proceedings of the 6th International Semantic Web Conference and the 2nd Asian Semantic Web Conference. (2007) 324–339
4. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: Proceedings of the 6th International Semantic Web Conference and the 2nd Asian Semantic Web Conference. (2007) 211–224
5. Erling, O., Mikhailov, I.: RDF Support in the Virtuoso DBMS. In Auer, S., Bizer, C., Müller, C., Zhdanova, A.V., eds.: Proceedings of the 1st Conference on Social Semantic Web. Volume 113 of LNI., GI (2007) 59–68
6. Erling, O.: Toward web scale RDF. In: Proceedings of the 4th International Workshop on Scalable Semantic Web Knowledge Base Systems. (2008)
7. Cudré-Mauroux, P., Agarwal, S., Aberer, K.: GridVine: An Infrastructure for Peer Information Management. IEEE Internet Computing **11**(5) (2007) 36–44
8. Owens, A., Seaborne, A., Gibbins, N., mc schraefel: Clustered TDB: A Clustered Triple Store for Jena. http://eprints.ecs.soton.ac.uk/16974/1/www2009fixedref.pdf (2008)
9. Anadiotis, G., Kotoulas, S., Oren, E., Siebes, R., van Harmelen, F., Drost, N., Kemp, R., Maassen, J., Seinstra, F.J., Bal, H.E.: MaRVIN: a distributed platform for massive RDF inference. http://www.larkc.eu/marvin/btc2008.pdf (2008)

10. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: MaRVIN: A platform for large-scale analysis of Semantic Web data. In: Proceeding of the WebSci'09: Society On-Line. (March 2009)
11. Soma, R., Prasanna, V.K.: Parallel Inferencing for OWL Knowledge Bases. In: ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing, Washington DC, USA, IEEE Computer Society (2008) 75–82
12. Weaver, J., Hendler, J.A.: Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In: Proceedings of the 8th International Semantic Web Conference. (2009)
13. Cai, M., Frank, M.R., Chen, J., Szekely, P.A.: MAAN: A Multi-Attribute Addressable Network for Grid Information Services. Journal of Grid Computing **2**(1) (2004) 3–14
14. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics **3**(2-3) (2005) 158–182
15. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB Endowment (2007) 411–422
16. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proceedings of the VLDB Endowment **1**(1) (2008) 647–659

Appendix

Below we list the four LUBM queries and one Barton query (defined in [14] and [15], respectively) used in our evaluation. We chose these four LUBM queries as representative and ranging from a single triple pattern (query 6) to a six-way join (query 2). Out of seven original Barton queries, only two can be represented in SPARQL (the others cannot due to their use of aggregates). Of the remaining two queries, we chose query 7 because it is the only query for which we can directly compare results with RDF-3X.

LUBM Query 2

```
PREFIX : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT DISTINCT * WHERE {
  ?z a :Department .
  ?z :subOrganizationOf ?y .
  ?y a :University .
  ?x :undergraduateDegreeFrom ?y .
  ?x a :GraduateStudent .
  ?x :memberOf ?z .
}
```

LUBM Query 3

```
PREFIX : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT DISTINCT * WHERE {
```

```
  ?x a :Publication .
  ?x :publicationAuthor
    <http://www.Department0.University0.edu/AssistantProfessor0> .
}
```

LUBM Query 4

```
PREFIX : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT DISTINCT * WHERE {
  ?x a :Professor .
  ?x :worksFor <http://www.Department0.University0.edu> .
  ?x :name ?y1 .
  ?x :emailAddress ?y2 .
  ?x :telephone ?y3 .
}
```

LUBM Query 6

```
PREFIX : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT DISTINCT * WHERE {
  ?x a :Student .
}
```

Barton Query 7

```
PREFIX : <http://simile.mit.edu/2006/01/ontologies/mods3#>
SELECT ?s ?bo ?co
WHERE {
  ?s :point "end" .
  ?s :encoding ?bo .
  ?s a ?co .
}
```

4store: The Design and Implementation of a Clustered RDF Store

Steve Harris, Nick Lamb, and Nigel Shadbolt

Garlik Ltd.
{steve.harris, nick.lamb, nigel.shadbolt}@garlik.com

Abstract. This paper describes the design and implementation of the 4store RDF storage and SPARQL query system with respect to its cluster and query processing design. 4store was originally designed to meet the data needs of Garlik, a UK-based semantic web company. This paper describes the design and performance characteristics of 4store, as well as discussing some of the trade-offs and design decisions. These arose both from immediate business requirements and a desire to engineer a scalable system capable of reuse in a range of experimental contexts where we were looking to explore new business opportunities.

1 Introduction

The need for 4store originated from a fundamental business requirement in Garlik. 4store was designed primarily to provide the backend storage for Garlik's DataPatrol[1], a consumer-facing personal information protection product. This product and its variants now have established user bases of many tens of thousands of individuals.

4store was implemented on a low-cost networked cluster with many tens of servers supporting a 24x7 operation. In addition Garlik has built semantically informed search and harvesting software and used industrial strength language engineering technologies across many millions of people-centric Web pages. Methods have been developed for extracting information from structured and semi structured databases. This information is organised against a lightweight people-centric ontology which, when imported comprises many billions of RDF triples.

Since its initial development 4store has been replaced within Garlik by a new clustered store with even greater scalability and efficiency. The 4store source code has been made available under the GNU General Public Licence version 3 [1]. The ANSI C99 source code and documentation can be found at `http://4store.org/`.

[1] `http://www.garlik.com/`. DataPatrol is an online application that checks databases and internet data sources for indications that personal information has "leaked" into the public domain.

1.1 Original Requirements

Due to the expected data volume that would be stored Garlik decided to aim for the storage of 10^9 quads in a cluster of nine machines, each machine having two processor cores, 4GB of RAM, and two SATA disks – a typical configuration for a commodity server at the time.

Average response time to SPARQL [2] queries was required to be in the low milliseconds range – for the typical queries that would be required to provide the service.

One of the features of the application was that a wide range of heterogeneous data would need to be incorporated into the system. Moreover, new data sets were likely to become available whose form and structure we could not anticipate. Since the exact nature of the data could not be known it was decided that the storage system should not be specialised to any particular RDF schema.

Data updates were intended to be applied in bulk. The RDF store refreshing a weeks worth of data at a time. Time allowed for this import was around eight hours. Due to the volume of updates taking place it was felt that transactions would be required to ensure the integrity of results.

1.2 Current Requirements

As the application grew, the requirements evolved. For the last version of 4store that was used to support DataPatrol, the requirement was to hold 15×10^9 triples in a cluster of nine machines with 8GB of RAM each.

The volume of updates averaged 4×10^9 triples per week, updated in a twelve hour period. However, it was found that explicit transactions were not necessary, so this requirement was dropped.

Garlik had also developed other applications backed by 4store during this period, including QDOS[2] and the QDOS FOAF Index[3], which brought their own requirements. The requirement that had the most impact on the design was for live updates, completing in a predictable time, proportional to the size of the RDF file imported. This required moving from an earlier quad index structure to the one described in section 5.2. It was important that we were able to adapt the design of the store to new business requirements.

2 Related Work

There are a number of other RDF storage systems which use cluster based storage, or share some design principles with 4store. These include:

3store Although 3store [3] is not a cluster-based RDF engine many of the design principles originated in 3store. In particular the method of mapping RDF Resources to integers is inherited more or less directly from 3store.

[2] `http://qdos.com/`, an online impact measuring application.

[3] `http://foaf.qdos.com/`, an index of around ten million FOAF files, stored in an instance of 4store.

Bigdata Bigdata [4] is another clustered RDF store. It has very high import performance, but little information about its design is available at this time.

Jena Clustered TDB Jena's Clustered TDB backend [5] has similar design goals to 4store. The paper describes an early prototype, rather than a production environment, but the segmentation and storage are substantially different to those in 4store.

Virtuoso Cluster Edition The Clustered Edition of Virtuoso [6] uses yet another clustering model based on the Map-Reduce algorithm and bitmap quad indices.

YARS2 YARS2 [7] uses a very different approach to 4store to achieve heavy utilisation of the cluster following a more conventional clustering model to spread the load across multiple nodes. It is known to scale to 9×10^9 triples.

3 Architecture

At the time of the initial design it was uneconomic to purchase computers with sufficient main memory to hold an adequate proportion of an RDF index for the projected data size. It was estimated that a reasonable average memory footprint for a quad was in the region of 100 bytes, implying that 93GB of RAM would be required to hold the complete index. As a result it was decided to pursue a clustered storage methodology.

For reasons of cost efficiency it was decided to base the cluster on commodity 64-bit, multicore x86 hardware, running the Linux operating system. At the time this was felt to offer the best price/performance ratio, and offers access to a large number of skilled administrators and systems programmers. The communications were to be provided by Gigabit Ethernet network interface controllers and switches. The choice of this hardware platform suggested the "Shared Nothing" architecture [8] as the most practical design.

3.1 Cluster Topology

The data is divided among a number of segments (non-overlapping slices of data), with one or more segments on every storage node, as shown in figure 1. These nodes are divided into *Processing* and *Storage* nodes.

It is also possible to run 4store on a single node, running the Processing front-end and one or more Storage back-ends on a single machine. 4store draws little advantage from the proximity, and the overhead of TCP communications between the Processing and Storage components is still incurred.

3.2 Segmentation

The segmentation model used in 4store is extremely simple. A RID integer (see section 5.1) is calculated for the subject of any given triple. The segment number is then computed such that

$$segment = \mathrm{rid}(subject) \bmod segments$$

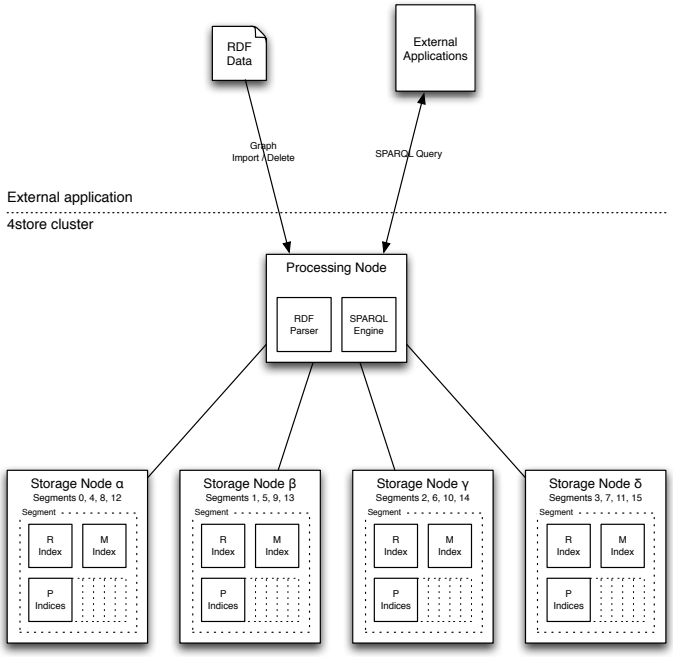


Fig. 1. 4store's cluster topology

To segment resources the same function is applied to the RID of the resource. This extremely simplistic segmentation schema has some benefits, but also a number of drawbacks, as illustrated below.

Benefits For commonly encountered data this segmentation scheme produces remarkably even distribution of data amongst the segments. If s_n is the population of segment n then the coefficient of variation (c_v) for a given system is given by $\frac{\sigma(s)}{\bar{s}}$. The values of c_v for the twenty five million triple BSBM [9] dataset, a sample of FOAF data[4], and the USGS TIGER/Line dataset [5] are shown in table 1.

Due to the relatively value low of c_v there is rarely any need to migrate segments between nodes, and there is little need to gather the statistics required for re-segmenting the data during import operations.

[4] Taken from a population of ten million FOAF files crawled in 2008 as part of the QDOS FOAF Index project

[5] The USGS TIGER/Line dataset, converted into RDF. This dataset was regularly used as test data in the development of 4store.

Dataset	c_v	$\frac{F_r}{F_q}$
BSBM 25MT	2.83×10^{-3}	14.99
FOAF	1.70×10^{-2}	9.09
TIGER/Line	1.59×10^{-3}	5.27

Table 1. Characteristics of various datasets

Drawbacks Synthetic datasets, and potentially real-world ones could skew the distribution of subjects in such as way as to increase the value of c_v, this would have a deleterious effect on the performance and efficiency of the cluster. For example, a large number of triples of the following form would increase c_v substantially:

```
country:US :citizen _:us1 .
country:US :citizen _:us2 .
...
```

Given a triple pattern where the subject is not known, the segmentation algorithm used cannot determine in which segment the matching quad or quads will be found. Because of this it is necessary for the querying process to contact every node in the cluster in order to find matches for this pattern. In practice this limits the application of this algorithm to relatively small clusters. However, in such small clusters it potentially offers an advantage in that the query optimiser is frequently given the choice between a broad shallow query across many nodes, or a narrow deep one against a single node.

When there are two or more potential query operations with similar specificity, but some have constant or known subject values, and some have constant or known object values then the query engine can make the choice between querying all nodes specifying one or more objects in the *bind* (see section 6.1), or it can specify the subjects and target on the required segments.

The trade-off is that broad bindings consume more resources overall, but complete in a shorter wall-clock time, the IO load being spread across many nodes in the cluster.

3.3 Segment Distribution and Replication

Given a set of nodes N, $\{\alpha, \beta, ...\}$ and a set of segments S, $\{0, 1, ...\}$ the nodes are assigned non-negative integer identifiers, starting from zero. The segments are the divided amongst the nodes, such that the set of segments assigned to node n with r replicas A^{nr} is as below.

$$A^{nr} = \bigcup_{m=0}^{r} R^{nm}$$

$$R^{nm} = \begin{cases} \{s \in S : \ s \bmod |N| = n\} & \text{for } m = 0 \\ \{s \in S : \left(\left(s + \left\lfloor \frac{s}{|N|} \right\rfloor\right) \bmod (|N| - m) + m\right) \bmod |N| = n\} & \text{for } m > 0 \end{cases}$$

For a cluster of 8 nodes, consisting of 32 segments with 2 way replication, the allocations would be as seen in table 2. The aim of this replication method is to ensure that should up to r nodes fail, the increased load is distributed evenly across the remaining nodes.

Node	R^{n0}	R^{n1}	R^{n2}
α	{0, 8, 16, 24}	{7, 14, 21, 28}	{6, 13, 20, 27}
β	{1, 9, 17, 25}	{0, 15, 22, 29}	{7, 14, 21, 28}
γ	{2, 10, 18, 26}	{1, 8, 23, 30}	{0, 15, 22, 29}
δ	{3, 11, 19, 27}	{2, 9, 16, 31}	{1, 8, 23, 30}
ϵ	{4, 12, 20, 28}	{3, 10, 17, 24}	{2, 9, 16, 31}
ζ	{5, 13, 21, 29}	{4, 11, 18, 25}	{3, 10, 17, 24}
η	{6, 14, 22, 30}	{5, 12, 19, 26}	{4, 11, 18, 25}
θ	{7, 15, 23, 31}	{6, 13, 20, 27}	{5, 12, 19, 26}

Table 2. Segment distribution across an eight node cluster with two way replication

4 Inter-Node Communications

Processing nodes communicate with storage nodes via TCP/IP. There is no direct communication between storage nodes. Having discovered the addresses of the storage nodes (see section 4.1) at startup a processing node asks each node which segments are stored there, and makes one connection per segment on that node. At this point the storage nodes may also optionally (configured at setup time) require an authentication step using a shared secret password to provide some degree of assurance that the processing node is authorised to access the data. The connection is not encrypted because of the likely impact on performance.

Connections between processing nodes and storage nodes are used to send variable sized messages using a type-length-value scheme, each message is either a request or a reply. Communication is always initiated by the processing node sending a message with a request. For most types of request the storage node replies with a message of its own, a few types do not require any reply. In place of the expected reply a storage node can send an error reply, including human readable text if there is a fatal error performing the requested action.

Only one message is sent at a time on any particular connection, but since there is a separate connection for each segment, a request can be sent to all the segments and then all the replies aggregated, meaning the total time to fulfill the request over the whole cluster is limited by the slowest response, rather than the sum of time taken to fulfill the request for each individual segment.

In order to provide replication requests which write new data to a segment are sent to all replicas of the segment, while to improve performance requests which only read data (e.g. the requests used for the *bind* functions described

in section 6.1) make requests to a single replica, and try to choose a replica on a node with least outstanding requests. If a storage node fails while in use, attempts to write data will report errors until it is repaired, but attempts to read data will continue to work normally (but potentially with reduced performance) if at least one replica of each segment is still accessible.

4.1 Discovery

From the outset we wanted processing nodes to be able to discover the storage nodes containing the relevant data without any specific configuration. The processing node needs to identify the complete address (IP address and port number) of a listening TCP socket on each storage node. A "well known port" was not desired, as this would impose a limit of only one instance of the storage node software per node. DNS Service Discovery [10] seemed well suited to this purpose and, since the storage nodes are on the local network, we used Multicast DNS [11] to enable this without needing a DNS server to be installed or specifically configured for this purpose.

Each storage node advertises a service with the 4store DNS service type (_4store._tcp). To distinguish multiple datasets stored on the same physical nodes, or on different nodes connected to the same network, each dataset has a unique name, which is included in a DNS TXT record, and the total number of segments is also included in the advertisement.

The processing nodes solicit advertisements for the 4store service type and then listen for advertisements. Received advertisements are checked to see that the name matches the desired dataset, and if so the processing node attempts to connect to the advertised address. If the connection fails, other addresses are tried. If after a reasonable time the processing node has not been able to identify and connect to all the storage nodes (or for a processing node which performs only queries, enough nodes to access all the distinct segments) it gives up and reports an error.

5 RDF Representation

5.1 Resources

RIDs RIDs (Resource IDentifiers) are used as a symbol encoding [12] for resource values. RIDs are 64-bit integers which represent URIs, Literals, and Blank Nodes using a disjoint value space. The one or two most significant bits of the RID value determine whether the RID encodes a URI, Literals or Blank Node:

MSB1	MSB2	Encodes
0		Literal
1	0	Blank Node
1	1	URI

In the case of URIs and Literals the remainder of the RID is made up of the least significant bits of a UMAC-64 [13] hash of the UTF-8 encoded lexical value of the resource. The cryptographic features of a strongly universal hash are of little relevance, however the collision resistance is desirable. In the case of literals with either a language tag or a datatype, an attribute RID is calculated from the language tag or datatype, stored as the *attr* and additionally exclusive-or'd with UMAC hash of the lexical value. For URIs, Blank Nodes, and Plain Literals the value of *attr* is zero.

Blank Nodes are encoded differently. An integer representing the highest blank node identifier is maintained on the storage node(s) holding segment zero. When an importing process wishes to allocate some Blank Node RIDs it requests a block of IDs, represented as (min, max) from segment zero. These IDs are bit-wise permuted in such a way as to keep both the MSBs and LSBs of the resulting ID varying frequently, whilst ensuring that distinct input IDs in the range $[0, 2^{62}]$ produce unique output IDs. The variability at both ends of the identifier ensures an even distribution of Blank Nodes across both segments, and in the trie used to store quads (see section 5.2).

Once a collision is detected on any given segment, all future translations from lexical forms to RIDs on that segment must be confirmed by the resource index in the appropriate segment, in order to prevent incorrect results from being returned.

As the RDF Literals and URIs are stored in separate value spaces the point at which collisions are likely depends on the number of unique literals and URIs in a given dataset. The probability of a collision occurring for n values in a space of d possible hash values is given by

$$1 - \prod_{k=1}^{n-1} \left(1 - \frac{k}{d}\right)$$

So, if the number of literals is f_l and the number of URIs is f_u then the overall probability of a collision in a particular segment, where there are s segments is:

$$1 - \left(\prod_{k=1}^{\frac{f_l}{s}-1} \left(1 - \frac{k}{2^{63}s}\right) \prod_{k=1}^{\frac{f_u}{s}-1} \left(1 - \frac{k}{2^{62}s}\right) \right)$$

Assuming that the number of resources in each segment is approximately equal, which is likely given the strong universality of the UMAC function [13].

The number of values that can be hashed before we expect to encounter a collision is given by $\sqrt{N}$ for a hash of N values. If we make the assumption[6] that $f_l \approx f_u$, and define f_r as the sum of f_l and f_u then the approximate number of resources that can be imported before we expect to encounter a collision is given by $2\sqrt{2^{62}}$.

[6] Although this situation is not especially likely the statistics for the breakdown into URIs and Literals are not available at this time. Nevertheless, this approximation should still provide a reasonable order-of-magnitude estimate

To know the expected number of quads that can be imported before encountering a collision (e_q) it is necessary to know the ratio of unique quads (f_q) to f_r.

$$e_q = 2^{32} \frac{f_q}{f_r}$$

Some values for $\frac{f_q}{f_r}$ are given in table 1. From this we can estimate that typically 3.9×10^{10} quads of FOAF data could be imported before collision handling would be required. Consequently, it is a worthwhile optimisation to delay handling of collisions until it becomes necessary.

Lexical Value Storage RDF Resources, (URIs, Literals, and Blank Nodes) are represented as a 3-tuple of $(rid, attr, lexical\ value)$, and stored in a bucketed, power-of-two sized hash table, shown as the "R Index" in figure 1. The *rid* and *attr* are both RIDs, and the *lexical value* is a text string, encoded in one of a number of ways.

5.2 Quad Storage

In 4store, RDF triples are represented as quads of $(model, subject, predicate, object)$, where a model is somewhat analogous to a SPARQL Graph. The chief differences between a SPARQL Graph and a model are in the handling of empty graphs and the behaviour of the default graph. In 4store, triples assigned to the default graph are placed in a particular model, which is used in query execution against the default graph – when the SPARQL default graph behaviour is enabled.

Although the quads are queried using a flat pattern structure (see section 6.1), the internal structure more closely resembles property tables [12].

Each quad in a particular segment is stored in three indexes. The first two will be described here and the third will be described in section 5.3. The indexes described here are shown as "P Indices" in figure 1.

The P Indices consist of a set of radix tries [14], two for each predicate, using a 4-bit radix. Ordinarily radix tries are at a disadvantage compared to balanced trees as their worst case lookup performance is $\mathcal{O}(k)$, where k is the key length, compared to $\mathcal{O}(\log n)$ for a B-Tree [15]. However the keys in this case have already been mapped to 64-bit integers, so are of finite, short length. Additionally, the integers are already evenly distributed across the space due to the combination of hashing and Blank Node identifier distribution, making the worst case lookup conditions uncommon.

The key for the per-predicate radix tries is the subject or object of the quads to be indexed. The graph and subject/object are stored in a list of rows, pointed to by leaf entry in the radix trie.

Queries with known subjects or objects, but unknown predicates are relatively expensive to execute, as all tries must be consulted to determine if matching subjects or objects appear with that predicate.

5.3 Model Storage

Graphs are indexed using a hash table that points to a list of rows of triples. This index is shown as the "M Index" in figure 1. The function of the graph index is twofold. Primarily it allows queries of the following form to be executed efficiently:

```
SELECT * WHERE { GRAPH <some-graph> { ?s ?p ?o } }
```

A side effect of this is that it allows graph-level deletes to be performed more efficiently, clearing out the Graph index entry and removing matching quads from the appropriate radix trees, this can be performed on each segment independently as all the pertinent information is held locally to the segment.

6 Query Operations

Often the focus on performance of clustered systems is on delegating work to cluster members in order to distribute the work. In large clusters with many parallel task to complete this is a significant efficiency gain, but on the relatively small clusters that 4store is designed to run on this is not always the case.

Running a test on a cluster of five machines, connected by Gigabit Ethernet on an isolated network the mean time over one thousand requests for one node to issue an empty request and receive an empty response from one cluster member is 175μs, given an established TCP/IP connection.

For comparison, a join on two 2,000 row binding tables, with one common variable can be completed on the same hardware in 520μs. Consequently, given a pure response-time consideration it's only advantageous to push the join down to cluster members if the tables can be split, sent, joined and returned by the remote cluster members in 520μs. Given the situation that CPU manufacturers are offering increasing numbers of cores, there is a potential to perform multiple joins simultaneously without incurring network IO overhead. As many such parallel operation can be performed locally, this optimisation looks increasingly unattractive for small operations.

There are however still situations in which performing the joins on cluster members has an overall time advantage. For instance, when the data for both sides of the join is already present on one member, one such situation will be discussed in section 8.2.

Equally a cluster where a very large volume of queries is being performed, saturating the CPU cores on the front-end machines, would benefit from pushing more joins down into the cluster. Another situation would be when the joins are typically performed across very large tables. A join between two binding tables has a complexity around $\mathcal{O}(n \log n)$, so breaking this down into m $\mathcal{O}(\frac{n}{m} \log \frac{n}{m})$ operations is a win in net processor time for sufficiently large n, regardless of the parallelism.

4store has two fundamental distributed query operations, *bind* and *resolve*, these are used to perform the underlying operations for all SPARQL expression evaluation and are described below.

6.1 The Bind Functions

The *bind* functions are used by the query engine when it wishes to produce a binding set for some SPARQL graph pattern. One function takes four multisets of RIDs, and the other a set of quads of RIDs that describe the match to be performed (see below). These arguments are presented to the network distribution algorithm. The network distribution algorithm decides what segment or segments should be consulted to return the complete set of bindings and divides the sets into one set for each segment that is to be consulted.

Once the results have been obtained from the segments of interest the network distributor performs a multiset union on the results and returns this to the query engine.

The primary form of the *bind* function takes four multisets of RIDs, M, S, P and O and a set of flags indicating which columns should be projected. These multisets are matched against the quads held by a particular segment (Q_s), containing a set of quads of RIDs such that quads of the form (m, s, p, o) are selected where:

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

The resulting multiset of quads is then projected to produce a multiset of n-tuples where n is the cardinality of the projection set.

There is also a second *bind* function, called the *reverse-bind*, for historical reasons. This *reverse-bind* function takes a set of quads R, and returns a multiset of quads such that:

$$\{(m, s, p, o) \in Q_s : (m', s', p', o') \in R, m = m' \vee m' = \omega, s = s' \vee s' = \omega, \\ p = p' \vee p' = \omega, o = o' \vee o' = \omega\}$$

This multiset is then projected as per the main *bind* form. ω in this expression is the "null" value, some nominated RID value which cannot appear in real data.

6.2 The Resolve Function

The *resolve* function is used to map RIDs to attribute RIDs and the lexical value of the input RID.

It takes a set of RIDs and returns a set of tuples of the form (*rid, attr, lexical value*), for a given segment. As in the *bind* case there is a network distributor that is responsible for sending *resolve* requests to the appropriate segments and the result is the union of the returned values from each segment.

7 Query Execution

The 4store query engine is largely based on Relational Algebra. This is due to the fact that 4store's query engine predates the finished SPARQL algebra. Moreover, there is a large body of literature around optimisation that can be

applied to relational algebra. Implementations that started after the publication of the SPARQL specification are more likely to use the SPARQL algebra. However many of the observations that follow are likely to be relevant to SPARQL algebra implementations.

4store uses the Rasqal SPARQL parser [16]. Rasqal produces a parse tree representing the underlying structure of the SPARQL expression. 4store walks this tree looking for occurrences of variables, which it records as metadata, and labels blocks of binding patterns with IDs. For example, consider the following query, where the block IDs are show in parentheses:

```
SELECT * WHERE {
 ?x a <Foo> .          (0)
 ?x <has> ?y .         (0)
 OPTIONAL {
  ?y <factor> ?z .     (1)
 }
 { ?y <value> ?v .     (2)
   ?v <label> ?l .     (2)
 } UNION {
   ?y <label> ?l . }   (3)
}
```

Additionally it records which blocks are joined to which other blocks, and by what operation. So, in this case we have:

child	parent	operation
1	0	$\rtimes$
2	0	$\bowtie$
3	2	$\cup$

The query executor descends the expression tree, internally joining the expressions to produce a table for each block. Although the projection is performed internally by the *bind* function, in the expressions below it will be shown separately, for clarity:

$$b_0 \leftarrow \pi_x \rho_{x/subject}(\text{bind}(\emptyset, \emptyset, \{\texttt{rdf:type}\}, \{\texttt{<Foo>}\})) \bowtie \pi_{x,y} \rho_{x/subject} \rho_{y/object}(\text{bind}(\emptyset, x, \{\texttt{<has>}\}, \emptyset))$$

$$b_1 \leftarrow \pi_{y,z} \rho_{y/subject} \rho_{z/object}(\text{bind}(\emptyset, b_0.y, \{\texttt{<factor>}\}, \emptyset))$$

$$b_2 \leftarrow \pi_{y,v} \rho_{y/subject} \rho_{v/object}(\text{bind}(\emptyset, b_0.y, \{\texttt{<value>}\}, \emptyset)) \bowtie \pi_{v,l} \rho_{v/subject} \rho_{l/object}(\text{bind}(\emptyset, v, \{\texttt{<label>}\}, \emptyset))$$

$$b_3 \leftarrow \pi_{y,l} \rho_{y/subject} \rho_{l/object}(\text{bind}(b_0.y, \emptyset, \{\texttt{<label>}\}, \emptyset))$$

The next phase collapses all the UNION expressions. Relational algebra has no equivalent to SPARQL's UNION, but we will use the $\cup$ symbol to represent SPARQL's UNION. UNION blocks are collapsed bottom-up (from highest block ID to lowest), first any FILTER expressions are evaluated, and non-satisfying rows are removed, then the binding tables for co-UNIONs (any blocks related by the $\cup$ operation in the operations table) are concatenated:

$$b_2 \leftarrow b_2 \cup b_3$$

Next the joins across the remaining blocks are performed:

$$b_0 \leftarrow b_0 \bowtie b_2 \rtimes b_1$$

Finally, any remaining FILTERs, ORDER BY, and DISTINCT are applied. FILTERs are left to as late as possible to avoid having to *resolve* more RIDs than required. The presence of LIMIT without ORDER BY, internal complexity limiting and other factors may indicate that not all lexical values for bindings are required. Calls to the *resolve* operation are relatively expensive, as they are more likely to require random access IO in the storage nodes, and can transfer large volumes of data.

It has been our experience that when dealing with queries over large volumes of data using the SPARQL protocol it is often necessary to use the LIMIT keyword, or enable some form of effort limiting, or soft limit to reduce the volume of answers that will be returned. Few HTTP client libraries expose sufficient support for flow control to indicate to the SPARQL server that enough answers have been obtained, or else that the query has been running for too long.

Where possible FILTER expressions are evaluated as results are streamed to the client, with blocks of RIDs from b_0 being resolved at once, based on an heuristic estimation of how many rows of values will be required to satisfy the query.

8 Notable Optimisations

8.1 Join Ordering Optimisation

The primary source of optimisation is the conventional ordering on the joins internal to a block join. We attempt to predict which *bind* will be the most specific, perform that one first, then successively apply the same specificity estimate, given the values from the binding table at that point.

Earlier versions of 4store had access to comprehensive quad histogram data. The current version only has access to predicate frequency information in order to perform this heuristic evaluation. This is due to the structure of the radix trie indexes, an earlier index form providing a highly efficient way to obtain occurrence histograms as a side-effect of its design.

8.2 Common Subject Optimisation

Where two or more binds of the form

$$\text{bind}(M^1, S, \{p_1\}, \{o_1\}), \text{bind}(M^2, S, \{p_2\}, \{o_2\}), \ldots \text{bind}(M^b, S, \{p_b\}, \{o_b\})$$

are encountered, where $|M^n| \leq 1$ this pair of binds can be transformed to a single *reverse-bind*:

$$\text{reverse-bind}\left(\bigcup_{n=1}^{b} \{(m, s, p_n, o_n) : s \in S, m \in M^n\}\right)$$

Where M^n is treated as $\{\omega\}$ if $|M^n| = 0$.

This has a twofold advantage. Firstly it reduces the number of network operations, and secondly (and more importantly) it breaks the join operation across the storage nodes, due to the way quads are segmented.

In the example of the following query:

```
SELECT ?x WHERE {
  ?x <givenName> "John" ;
     <familyName> "Smith" .
}
```

Any pair of triples matching this pattern will fall into the same segment, as they must share a subject RID, so when the storage node performs the join it will only have to consider one segment at a time, eliminating unnecessary bindings for ?x before they reach the front-end, and thus reducing the search space.

8.3 Cardinality Reduction

If the REDUCED or DISTINCT keywords are used then the cardinality of *bind* functions need not be preserved. Because of this, the presence of one of these keywords is passed down to the storage node, and it takes any time-efficient measures that are available to reduce the cardinality of the result set. For example, by removing adjacent identical rows, and also by use of index structures. Given a *bind* of the form $\pi_{predicate}(\text{bind}(\emptyset, \emptyset, x, \emptyset))$ it is sufficient to to consult the list of predicate indices, to return the RID values for the matching predicates present in the segment. If DISTINCT is specified then the front-end still needs to perform the DISTINCT operation, but typically the size of the result set returned will be greatly reduced.

Similar optimisations are available for bindings to all subjects, objects, models, and resources.

8.4 Unreferenced Variables

It is often necessary to place variables in graph patterns where the value of the variable is not required. Consider a query to find all the people that have some employer:

```
SELECT DISTINCT ?x WHERE {
  ?x a <Person> ;
     <has> ?employer .
}
```

By inspection it is possible to ascertain that bindings for ?employer are not required, we simply have to ensure that such a binding exists. Given this, the bind call for the second triple pattern above can be reduced to:

$$\pi_x \rho_{x/subject}(\text{bind}(\emptyset, x, \{\texttt{<has>}\}, \emptyset))$$

Without the DISTINCT or REDUCED keywords the engine is still required to preserve the cardinality of ?x, but in either case we can avoid holding a column of bindings in the binding table.

9 Future Work

9.1 Updates

As of writing the only update operations that are supported in the front-end are deleting an entire model, and adding triples to a model. The nascent SPARQL/Update specification will require fine-grained updates.

9.2 Full Text Indexing

Currently 4store has no index that can efficiently address full text searches. This is supported in SPARQL via the *regex* function. More sophisticated full-text searching is commonly offered as a non-standard extension. This is an area for future work.

10 Conclusion

In this paper we have described in detail the architectural design principals and methods of implementation for key aspects of our clustered RDF store. We discussed the merits and demerits of a number of fundamental features of the store such as its segmentation model. We have detailed a number of optimisation strategies and we have also reviewed the performance characteristics and trade-offs that informed our design.

We believe that there is much to be gained by sharing effective design patterns, best practice, and hard won insights amongst our emerging community.

References

1. Free Software Foundation: GNU General Public License. `http://www.gnu.org/licenses/gpl.txt` (June 2007)

2. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/ (2005)
3. Harris, S., Shadbolt, N.: SPARQL query processing with conventional relational database systems. In Dean, M., Guo, Y., Jun, W., Kaschek, R., Krishnaswamy, S., Pan, Z., Sheng, Q.Z., eds.: WISE Workshops. Volume 3807 of Lecture Notes in Computer Science., Springer (2005) 235–244 http://eprints.ecs.soton.ac.uk/11126/1/harris-ssws05.pdf.
4. Personick, M.: Bigdata: Approaching web scale for the semantic web. http://www.bigdata.com/whitepapers/bigdata_whitepaper_07-08-2009.pdf (2009)
5. Owens, A., Seaborne, A., Gibbins, N., mc schraefel: Clustered TDB: A clustered triple store for Jena. In: WWW2009. (November 2008) http://eprints.ecs.soton.ac.uk/16974/.
6. Erling, O., Mikhailov, I.: Towards web scale RDF. In: Proceedings of the 4th International Workshop on Scalable Semantic Web Knowledge. (October 2008) http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VOSArticles/VOSArticleWebScaleRDF.pdf.
7. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A federated repository for querying graph structured data from the web. In Aberer, K., Choi, K.S., Noy, N.F., Allemang, D., Lee, K.I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P., eds.: ISWC/ASWC. Volume 4825 of Lecture Notes in Computer Science., Springer (2007) 211–224 http://www.deri.ie/fileadmin/documents/DERI-TR-2007-04-20.pdf.
8. Stonebraker, M.: The case for shared nothing. IEEE Database Eng. Bull. **9**(1) (1986) 4–9 http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf.
9. Bizer, C., Schultz, A.: Berlin SPARQL benchmark (BSBM) specification - v2.0. http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/
10. Cheshire, S.: DNS service discovery (DNS-SD). http://www.dns-sd.org/ (2009)
11. Cheshire, S.: Multicast DNS. http://www.multicastdns.org/ (2006)
12. Wilkinson, K.: Jena property table implementation. Technical report, Hewlett-Packard Labs (October 2006) http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.530.
13. Krovetz, T.: UMAC: Message authentication code using universal hashing. IETF **RFC 4418** (March 2006) http://tools.ietf.org/html/rfc4418.
14. Morrison, D.R.: PATRICIA - practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM **15(4)** (October 1968) 514–534 http://portal.acm.org/citation.cfm?doid=321479.321481.
15. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indexes. Technical report, Boeing Scientific Research Laboratories (July 1970) http://www.minet.uni-jena.de/dbis/lehre/ws2005/dbs1/Bayer-McCreight.pdf.
16. Beckett, D.: Rasqal RDF Query Library. http://librdf.org/rasqal/ (2005)

Efficient reasoning on large SHIN Aboxes in relational databases

Julian Dolby[1], Achille Fokoue[1], Aditya Kalyanpur[1], Li Ma[2], Chintan Patel[3], Edith Schonberg[1], Kavitha Srinivas[1], and Xingzhi Sun[2]

[1] IBM Watson Research Center,P.O.Box 704, Yorktown Heights, NY 10598, USA
dolby, achille, adityakal, ediths, ksrinivs@us.ibm.com
[2] IBM China Research Lab, Beijing 100094, China
malli, sunxingz@cn.ibm.com
[3] Columbia University Medical Center
chintan.patel@dbmi.columbia.edu

Abstract. As applications based on semantic web technologies enter the mainstream, there is a need to provide highly efficient ontology reasoning over large Aboxes. However, achieving sufficient scalability is still a challenge, especially for expressive ontologies. In this paper, we present a hybrid approach which combines a fast, incomplete reasoning algorithm with a slower complete reasoning algorithm to handle the more expressive features of DL. Our approach works for $\mathcal{SHIN}$. We demonstrate the effectiveness of this approach on large datasets (30-60 million assertions), including a clinical-trial patient matching application, where we show significant performance gains (an average of 15 mins per query compared to 100 mins) without sacrificing completeness or expressivity.
keywords: Reasoning, Description Logic, Ontology.

1 Introduction

As applications based on semantic web technologies enter the mainstream, there is a need to provide highly efficient ontology reasoning over large Aboxes. However, achieving sufficient scalability is still a challenge. DL reasoning is intractable in the worst case. In [1], we reported on the use of expressive reasoning for matching patient records to clinical trial criteria. While the system was able to successfully reason on 240,269 patient records, a knowledge base with 59 million Abox and 33,561 Tbox assertions, the execution time was prohibitive. In some cases, the system took hours to respond.

The expressivity of the patient knowledge base was $\mathcal{ALCH}$, so expensive reasoning was needed to be complete. However, most typical queries were simple, and could have been answered faster with a less expensive reasoner. A high cost was paid by all queries to support rarer complex queries. In this paper, we present a hybrid approach, that combines a fast, incomplete reasoning algorithm with a slower complete reasoning algorithm to handle the more expressive features of DL. In this way, we were able to dramatically lower the cost of typical simple queries, without losing the ability to answer more complex queries.

An interesting feature of our technique is that *any sound and incomplete algorithm* may be used in the first phase to quickly find as many solutions as possible to the query. The key novelty in the approach is a mechanism to incorporate these solutions into a slower, complete reasoning algorithm for $\mathcal{SHIN}$, providing much better performance characteristics overall, without sacrificing completeness or expressivity. This approach can be described as self-adjusting, since the reasoner dynamically defaults to the expensive complete algorithm only when deeper inferencing is actually required. On large datasets (30-60 million assertions), this hybrid approach provides significant performance gains (an average of 15 mins per query on the 60 million dataset compared to 100 mins) without sacrificing completeness or expressivity.

At its core, this hybrid approach builds on the summarization and refinement techniques we described earlier to perform sound and complete reasoning on large Aboxes in relational databases [2] [3]. Briefly, this technique applies a standard tableaux algorithm on a *summary Abox* $\mathcal{A}'$ rather than the original Abox $\mathcal{A}$ to answer queries. A summary Abox is created by aggregating individuals which are members of the same concepts, so when any given individual is tested in the summary Abox, all individuals mapped to the summary individual are effectively tested at the same time. For a tested individual s in $\mathcal{A}'$, if the summary is found to be consistent, then we know that all individuals mapped to that summary individual s are not solutions. But if the summary is found to be inconsistent, it is possible that either (a) a subset of individuals mapped to the summarized individual s are instances of the query or (b) the inconsistency is a spurious effect of the summarization. We determine the answer through *refinement*, which selectively expands the summary Abox to make it more precise. Refinement is an iterative process that partitions the set of individuals mapped to a single summary individual based on the common edges they have in the original Abox, and remaps each partition to a new summary individual. The iteration ends when either the expanded summary is consistent, or it can be shown that all individuals mapped to the tested summary individual are solutions. Significantly, convergence on the solution is based only on the structure of the refined summary, without testing individuals in $\mathcal{A}$. In practice, the scalability of this algorithm is limited by the number of refinement steps that are needed. Refinement is performed by database join operations, which become expensive when the database is large.

The key insight of our hybrid approach is that the solutions from the sound and incomplete reasoner can be used as a partitioning function for refinement instead of partitioning based on common edges, as described in our earlier work. This effectively removes the obvious solutions from the summary Abox. If the sound and incomplete reasoning algorithm finds all solutions, there will be no solutions left in the summary Abox after this first refinement, so the algorithm will converge very quickly. Any remaining inconsistencies are spurious, and can be resolved in one or a few refinement steps. If the sound and incomplete algorithm finds only some of the solutions, then the refinement process will find the rest of the solutions with fewer refinement steps.

Our key contributions in this paper are as follows: (a) we develop a fast, sound but incomplete algorithm based on query expansion, and describe how to incorporate solutions from this and other such techniques into a sound and complete hybrid algorithm for reasoning over large expressive Aboxes, and (b) we demonstrate its effectiveness in providing performance gains (from 100 minutes per query to 15 minutes per query) on expressive Aboxes with 60 million assertions.

2 Related Work

There have been efforts in the semantic web community to define less expressive subsets of OWL-DL for which reasoning is tractable. The EL-family of languages [4] is one such example, for which classification can be done in polynomial time. To take advantage of this fact, various query answering algorithms for EL have been proposed (e.g. [5]). Another example is the DL-Lite family [6], for which conjunctive query answering is expressible as a first-order logic formula (and hence an SQL query) over the Abox stored in a relational database. The QuOnto algorithm [6] is a sound and complete query expansion algorithm for DL-Lite.

Our query expansion algorithm described in Section 6 is not significantly novel. It is similar in spirit to the EL and DL-Lite query expansion approaches, with some differences, namely: (i) instead of using an EL reasoner to compute additional subclasses during the normalization process (as in [5]), we use a sound and complete OWL-DL reasoner (Pellet) which enables us to discover more entailments outside of EL; (ii) we use a datalog reasoner to compute *same-as*-individual inferences (considering functional properties) and transitive closure for transitive properties that exist in the ABox.

Furthermore, a key point is that *any* query answering algorithm for a subset of OWL can be plugged into our sound and complete hybrid OWL-DL reasoning system. When it is known that the optimization is complete based on the underlying logic of the KB[4] and the manner in which it is implemented, fallback to our refinement strategy is not necessary. Otherwise, the refinement process will find any remaining solutions.

3 Background

Query answering in expressive DLs can be reduced to consistency detection. For instance, assume that we want to find all instances of the concept C. To answer this query, each individual a is tested by adding the assertion $a : \neg C$ to the Abox, and checking the new Abox for consistency. If the Abox is inconsistent, then a is an instance of C. For large Aboxes, this approach will clearly not scale. Therefore, in our previous work [3], [7], we modify this approach to perform tableau reasoning on a summarized version of the Abox rather than the original

[4] Checking whether the logic falls in EL or DL-Lite is a matter of syntactic checking of the KB axioms which can be done easily

Abox. Formally, an Abox $\mathcal{A}'$ is a summary Abox of a $\mathcal{SHIN}$ Abox $\mathcal{A}$ if there is a mapping function $\mathbf{f}$ that satisfies the following constraints[5]:

(1) if $a : C \in \mathcal{A}$ then $\mathbf{f}(a) : C \in \mathcal{A}'$
(2) if $R(a, b) \in \mathcal{A}$ then $R(\mathbf{f}(a), \mathbf{f}(b)) \in \mathcal{A}'$
(3) if $a \neq b \in \mathcal{A}$ then $\mathbf{f}(a) \neq \mathbf{f}(b) \in \mathcal{A}'$

If the summary Abox $\mathcal{A}'$ obtained by applying the mapping function $\mathbf{f}$ to $\mathcal{A}$ is consistent w.r.t. a given Tbox $\mathcal{T}$ and a Rbox $\mathcal{R}$, then $\mathcal{A}$ is consistent w.r.t. $\mathcal{T}$ and $\mathcal{R}$. However, the converse does not hold. In the case of an inconsistent summary, we use a process of iterative refinement to make the summary more precise, to the point where we can conclude that an inconsistent summary $\mathcal{A}'$ reflects a real inconsistency in the actual Abox $\mathcal{A}$. Refinement is a process by which only the part of the summary that gives rise to the inconsistency is made more precise, while preserving the summary Abox properties (1)-(3). To pinpoint the portion of the summary that gives rise to the inconsistency, we focus on the *justification* for the inconsistency, where a justification is a minimal set of assertions which, when taken together, imply a logical contradiction.

We define refinement for a summary individual s in a justification $\mathcal{J}$ as a partition where individuals mapped to s are partitioned based on which edges in $\mathcal{J}$ each individual actually has. More specifically:

$$key(a, \mathcal{J}) \equiv \left\{ R(t, s) \;\middle|\; \begin{array}{l} \mathbf{f}(a) = s \wedge \\ R(t, s) \in \mathcal{J} \wedge \\ \exists b \text{ in } \mathcal{A} \text{ s.t.} \\ R(b, a) \in \mathcal{A} \wedge \\ \mathbf{f}(b) = t \end{array} \right\} \cup \left\{ R(s, t) \;\middle|\; \begin{array}{l} \mathbf{f}(a) = s \wedge \\ R(s, t) \in \mathcal{J} \wedge \\ \exists b \text{ in } \mathcal{A} \text{ s.t.} \\ R(a, b) \in \mathcal{A} \wedge \\ \mathbf{f}(b) = t \end{array} \right\}$$

Since an individual may be mapped to a summary individual that is in multiple overlapping justifications, we define:

$$key^*(a) = \bigcup_{\{\mathcal{J} | a \in \mathcal{J}\}} key(a, \mathcal{J})$$

In a *refinement step* that refines s in $\mathcal{A}'$, new individuals $s_1 ... s_k$ replace s in $\mathcal{A}'$, where there are k unique key sets $key^*(a)$, for all a in $\mathcal{A}$ such that $\mathbf{f}(\mathrm{a}) =$ s. Individuals a and b in $\mathcal{A}$ mapped to s in $\mathcal{A}'$ are partitioned correspondingly, that is, $\mathbf{f}(\mathrm{a}) = \mathbf{f}(\mathrm{b})$ after the refinement step iff $key^*(a) = key^*(b)$ before the refinement step.

In principle, in the presence of many justifications involving overlapping sets of nodes, the union of the keys could become very large. In practice, we have not observed this across the various knowledge bases we have evaluated, even for ones that do contain overlapping justifications.

If all individuals in $\mathcal{A}$ mapped to a summary individual s have the same key w.r.t. $\mathcal{J}$, then it must be the case that they have all the edges in the justification

[5] We assume without loss of generality that $\mathcal{A}$ does not contain an assertion of the form $a \doteq b$

and hence s is precise w.r.t. $\mathcal{J}$. If a justification is precise, we can conclude that all individuals in $\mathcal{A}$ mapped to the tested individual in the justification are solutions to the query. In the worst case, iterative refinement can expand a summary Abox into the original Abox, but in practice, we conclude on precise justifications with many individuals mapped to each summary node in the justification.

Our implementation of summarization and refinement in a system called SHER is in terms of RDBMS operations to allow the system to scale to large data sets. However, the iterative process of summarization and refinement is expensive, because (a) it requires expensive join operations on all role assertions in the Abox $\mathcal{A}$ to define the $key(a)$, as well as expensive join operations of role assertions with type assertions to rebuild the summary, and (b) it requires several consistency checks to find the many sources of inconsistencies for each summary that gets built. For large knowledge bases with multiple ways in which one can derive a solution to the query, this becomes a serious performance bottleneck.

4 A Sample Knowledge Base

We illustrate our techniques with the sample knowledge base (Tbox $\mathcal{T}$, the Rbox $\mathcal{R}$ and the Abox $\mathcal{A}$) in Figures 1 and 2. This example is a small subset of the UOBM [8] benchmark that we use in our evaluation. To form the summary Abox for Figure 2, the individuals a and b are mapped to a single summary individual w with a concept set of $Woman$, and the individuals f, g and j are mapped to another summary individual p with a concept set of $Person$. The summary Abox is shown in the Figure 3.

$\mathcal{T}$ assertions:

(1) $WomanCollege \sqsubseteq \forall hasStudent.Woman$
(2) $\top \sqsubseteq \leq 1 isTaughtBy$

$\mathcal{R}$ assertions:

(1) $loves \sqsubseteq likes$
(2) $isStudentOf$ is inverse of $hasStudent$
(3) $teacherOf$ is inverse of $isTaughtBy$

Fig. 1. Example $\mathcal{T}$, $\mathcal{R}$

Consider the query $WomanWithHobby$, which is defined as $Woman \sqcap \geq 1 likes$. There are three solutions. The individual b is a solution because $loves \sqsubseteq likes$. The individual f is a solution because the course d can be taught by only one $Person$, and so f and b will be identified with each other during reasoning. Finally, g is a solution, since $isStudentOf(g, WomenCollege)$ implies that g is a $Woman$.

Figure 3 shows the entire refinement process for answering this query:

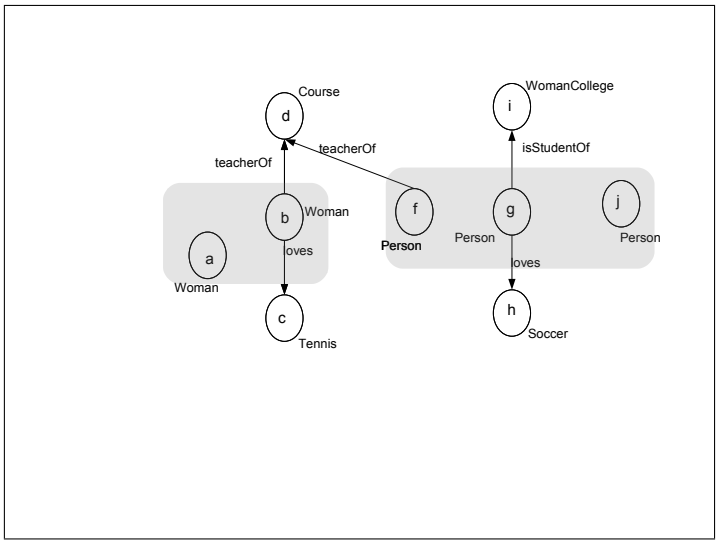


Fig. 2. Example $\mathcal{A}$

(1) Refine w by splitting it into two nodes w' which has a mapped to it, and w'' which has b mapped to it.
(2) Refine p by splitting it into two nodes p' which has g mapped to it, and p'' which has f and j mapped to it.
(3) Refine p'' further, by splitting it into nodes p_1 which has f mapped to it, and p_2 which has j mapped to it.

We explain these steps in more detail. First, $\neg WomanWithHobby$ is added to a tested summary individual w. The resulting Abox is inconsistent, and a justification $\mathcal{J}$ contains the assertions: $w : Woman$, $loves \sqsubseteq likes$, and $loves(w, c)$. For refinement, we target the summary individuals in $\mathcal{J}$, which are w and c. Refinement makes a justification $\mathcal{J}$ *precise*, that is, it partitions the individuals mapped to the summary node w into a new set of summary nodes to reflect the fact that not all individuals in $\mathcal{A}$ mapped to w have the $loves(w, c)$ in $\mathcal{J}$. The summary individual w is therefore split into two new summary nodes, w' that has individuals with no $loves(w, c)$ mapped to it (e.g., a), and w'' that has individuals with $loves(w, c)$ mapped to it (e.g., b). This new refined Abox is still inconsistent, with a new justification $\mathcal{J}$ which contains the individuals w'' and c. Refinement of w'' or c however is no longer possible, because every individual in $\mathcal{A}$ that is mapped to w'' also has the $loves(c, ,)$ and every individual mapped to c has the same edge (here c is the same as the summary node c). At this point, the justification $\mathcal{J}$ is precise, in that it cannot be refined further, and we conclude that all individuals in $\mathcal{A}$ mapped to w'' are solutions to the query.

For the second step, $\neg WomanWithHobby$ is added to a tested summary individual p. The resulting Abox is inconsistent, and this time there is the jus-

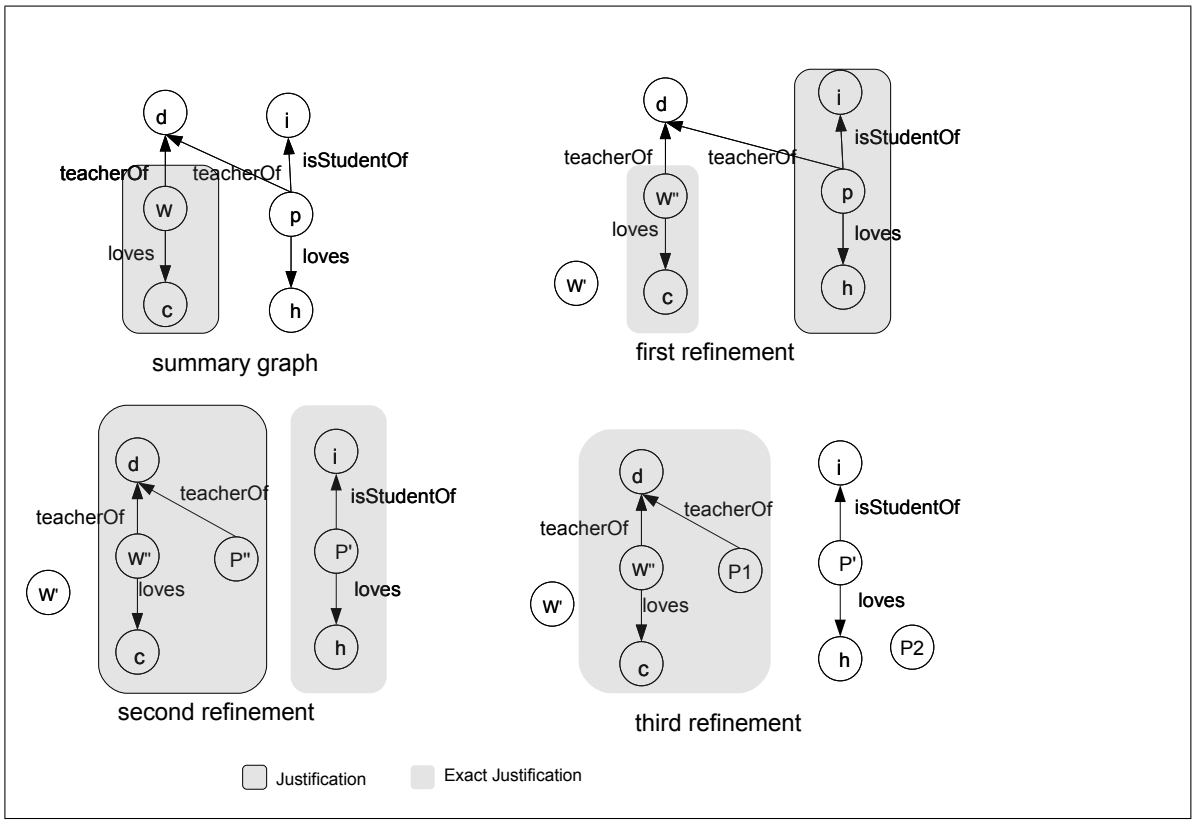


Fig. 3. Refinement Steps for Example

tification: $isStudentOf(p, i)$, $loves \sqsubseteq likes$, and $loves(p, h)$, combined with the axiom $WomanCollege \sqsubseteq \forall hasStudent.Woman$. The result of the second refinement is shown in Figure 3. After this refinement, the subgraph containing p' is still inconsistent, and p' is not refinable. Therefore, all individuals in $\mathcal{A}$ mapped to p', namely g, are solutions.

There is one final justification which is refinable: $teacherOf(p'', d)$, $teacherOf(w'', d)$, $w'' : Woman$, $loves \sqsubseteq likes$, $loves(w'', c)$, and $\top \sqsubseteq \leq 1 isTaughtBy$. After the third refinement step, we conclude that f mapped to $P1$ is a solution.

On large knowledge bases, the cost of each additional refinement is significant, so it is critical to reduce the number of refinements. We show in the next sections how our hybrid reasoning approach can reduce the number of refinements for this example.

5 Hybrid Algorithm

The key idea to reducing refinement iterations is to (a) quickly find solutions to the query, (b) refine the summary to isolate these solutions into new summary individuals, and (c) ignore these individuals for the rest of the refinement process. We find solutions quickly by using a sound and incomplete reasoning algorithm which does a form of query expansion described in Section 6. We point out that other reasoner implementations (such as QuOnto) for less expressive logics may also be plugged into this technique.

To illustrate the overall idea in terms of our example in Figure 2, we expand our query $WomanWithHobby$ into the query $WomanWithHobby(x) \sqcup (Woman(x) \sqcap likes(x, y)) \sqcup (Woman(x) \sqcap loves(x, y))$. This query matches all pairs of individuals in the Abox bound to both x and y, namely the pair (b, c), and this constitutes our set of known bindings. Our next step is to refine the summary Abox, so that the individuals in the solution, namely b and c, are mapped to distinct new summary individuals. We do this by a refining the summary Abox in a manner similar to that described in Section 3; the only difference is that we now partition the Abox individuals according to whether they were bound to any variable in the query or not, rather than according to key sets. That is, $\mathbf{f}(a) = \mathbf{f}(b)$ after the refinement step iff a and b are mapped to the same summary node before the refinement step and either both or neither a and b are individuals in the set of known bindings. Our algorithm keeps track of the subset of known bindings that actually are answers to the query, which is just b in this case. Next, consistency checking is applied to this refined summary, and any remaining inconsistencies are resolved using the standard iterative refinement and summarization process described in [3].

This approach has a nice property: in cases where the incomplete step actually does find all solutions and the summary itself is consistent, the complete reasoning step may simply be a single consistency check on the refined summary. Since there are no more solutions to be found, the only possible causes of inconsistency are spurious inconsistencies, which are the result of our summarization technique. In practice, we find that the incomplete step captures all solutions on most complex queries on most realistic datasets. This optimization therefore significantly reduces the number of refinements and makes query answering practical for large Aboxes.

One non-obvious part of the hybrid algorithm is that it is important to partition out *all* individuals that are bound to any variable in the query, and not just the individuals that are are actual solutions to the query. To illustrate why this is the case, consider a simple Abox shown in Figure 4 with 3 patients (q, r, s) who each have an associated lab event (l, m, n), and each event indicates a presence of organisms of different types, where x, y, and z indicate individuals with organisms of type X, Y and Z, respectively. The summary Abox, as shown in the Figure will contain one patient individual p, which has q, r and s mapped to it, one lab event individual e which has l, m and n mapped to it, and 3 individual nodes for organisms x, y, and z. Consider a realistic query, which is to find all patients who have a laboratory event which shows the presence of

the organism X. As shown in the Figure 4, if a summary is built with only the solution individual q partitioned out, then it will contain spurious inconsistencies which will cause unnecessary refinement. To avoid this issue, we should not only partition out the solution individual q from p, but also other individuals bound to other variables in the query, which in our example would be l and x.

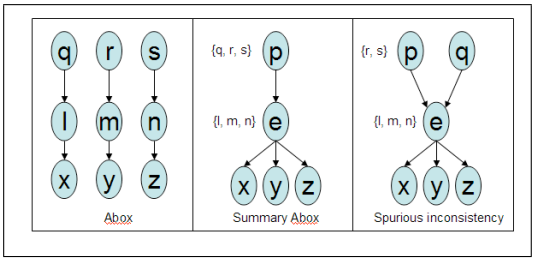


Fig. 4. Partitioning Complexity

The pseudo-code for our overall algorithm is shown in the function ConjunctiveQuery in Figure 5.

6 Query Expansion

Our sound but incomplete reasoning algorithm is based on the well-known recursive query expansion technique suggested in the EL [5] and DL-Lite [6] solutions. As discussed earlier, our approach differs in the following ways: (a) we refer to an OWL-DL reasoner (Pellet) for computing subclasses of a concept when performing the expansion, (b) we have an ABox pre-processing step that uses a datalog reasoner to compute transitive relations in the Abox and *same-as* inferences between ABox individuals due to functional property assertions. The *same* individuals are used to expand query solutions, i.e, if individual a is found to be a solution to the SQL query generated by query expansion, and $sameAs(a,b)$ is inferred by the datalog reasoner, we add b to the solution set.

For any given query $a : C$, we recursively traverse the definitions and subclasses of the concept C. For our sample query $x : WomanWithHobby$, we first generate a union of SQL select statements which signify all the possible ways in which this query can be expanded. The first disjunct in the union matches individuals of $WomanWithHobby$ directly, $rdf : type(x, WomanWithHobby)$. In this case, however, the $WomanWithHobby$ type does not appear in the Abox, and so we drop this disjunct. Next we would generate disjuncts to match individuals that are in subclasses of $WomanWithHobby$, but in this case there are no subclasses (checked by calling a standard DL reasoner). We then add any complex subclasses of $WomanWithHobby$ which can be inferred syntactically. In our example, we have one such obvious subclass because $WomanWithHobby$

Function: ConjunctiveQuery

Input: Conjunctive Query CQ: $C_i(x) \wedge ..R_j(x, y)$

```
/* Get incomplete answers from sound but incomplete algorithm, which
   can be translated to SQL                                          */
```

sqlQuery ← BuildQuery(CQ);

```
/* Get the bindings for all variables in the expanded query, both
   distinguished and non-distinguished variables                     */
```

result ← execute(sqlQuery);

```
/* Build filtered summary for query answering, which is the basic
   summary Abox                                                      */
```

sum ← BuildSummary($\mathcal{A}$, CQ);

```
/* Separate the bindings for distinguished variables x_dist from
   bindings for existentially quantified variables                   */
```

sqlsolutions ← getBindings(result, x_{dist});

others ← $\bigcup_{v \in vars(result) - x_{dist}}$ getBindings(result, v);

```
/* Refine summary based on solutions found from SQL                  */
```

sum ← refineSummaryFromSolutions(sum, sqlsolutions ∪ others) ;

```
/* Find all summary nodes in new summary which have sqlSolutions
   mapped to them                                                    */
```

sumSolutions ← getSummaryNodesForSQLSolutions(sum, sqlSolutions);

```
/* complete query answering, using refined summary                   */
```

restsolutions ← solveQuery(sum, allnodes - sumSolutions);

return *sqlsolutions* ∪ *restsolutions*

Fig. 5. Overall optimized complete query algorithm

is defined as equivalent to $Woman \sqcap \geq 1likes$. The expansion process now recursively continues and we expand this complex concept into a select statement which is a disjunction of conjuncts; i.e., the selection must satisfy the two conditions $rdf : type(x, Woman)$ and $likes(x, y)$, or alternatively, satisfy the two conditions $rdf : type(x, Woman)$ and $loves(x, y)$, since $likes$ has a subproperty $loves$. These queries are applied against an Abox that has been processed to include all edges materialized from the application of all deterministic merger and transitivity rules.

One technical challenge in query expansion in general is keeping the query relatively simple, especially when given very large Tboxes with deep subclass and subproperty hierarchies. Our approach to this problem was to eliminate forms of query expansion if the concept or role did not appear in the ABox. We therefore maintained a simple cache of all roles and concepts that appeared in the ABox, and limited our expansion to only these concepts and roles.

7 Evaluation

We evaluated our technique on two knowledge bases: the first is a real-world knowledge base, and real queries of clinical data that we had used in previ-

ous work[1], and the second is the UOBM benchmark[8]. Our experiments were conducted on a 2-way 2.4GHz AMD Dual Core Opteron system with 16GB of memory running Linux, and we used IBM DB2 V9.1 as our database. Our Java processes were given a maximum heap size of 8GB for clinical data, and 4GB for UOBM.

7.1 Clinical trials dataset

In prior work [1], we reported on the use of expressive reasoning for matching of patient records on clinical trials. The 1 year anonymized patient dataset we used contained electronic medical records from Columbia University for 240,269 patients with 22,561 Tbox subclass assertions, 26 million type assertions, and 33 million role assertions. The 22,561 Tbox subclass assertions are a subset of the a larger Tbox which combines SNOMED with Columbia's local taxonomy called MED for a total of 523,368 concepts. For details of the partitioning algorithm used to define the subset see [1]. Although the expressivity of the SNOMED version we used falls in the EL fragment of DL, the expressivity needed to reason on the knowledge base is $\mathcal{ALCH}$. This is because we have type assertions in the Abox which includes assertions of the type $\forall R.\neg C$, where the concept C is itself defined in terms of a subclass or equivalence axiom. As a concrete example, for a given patient, and a specific radiology episode for the patient, the presence of $ColonNeoplasm$ may be ruled out. $ColonNeoplasm$ has complex definitions in SNOMED (e.g., $ColonNeoplasm \equiv \exists AssociatedMorphology.Neoplasm \sqcap \exists FindingSite.Colon \sqcap ColonDisorder$). We selected the 9 clinical trials we evaluated in our earlier work which are shown Table 1. Table 2 shows the DL version of the queries, in the order shown in Table 1. For query $NCT00001162$, the results shown are for the union of 7 different disorders, only 4 of which are illustrated in Table 2.

ClinicalTrials.gov ID	Description
$NCT00084266$	Patients with MRSA
$NCT00288808$	Patients on warfarin
$NCT00393341$	Patients with breast neoplasm
$NCT00419978$	Patients with colon neoplasm
$NCT00304382$	Patients with pneumococcal pneumonia where source specimen is blood or sputum
$NCT00304889$	Patients on metronidazole
$NCT00001162$	Patients with acute amebiasis, giardisis, cyclosporiasis or strongloides...
$NCT00298870$	Patients on steroids or cyclosporine
$NCT00419068$	Patients on corticosteroid or cytotoxic agent

Table 1. Clinical Trial Requirements Evaluated

Table 3 shows the queries, the number of patients matched to the queries, the time to process the queries in minutes, the time in minutes for our hybrid

DL Query
$\exists associatedObservation.MRSA$
$\exists associatedObservation.$ $\exists roleGroup.$ $\exists administeredSubstance.$ $\exists roleGroup.\exists hasActiveIngredient.Warfarin$
$\exists associatedObservation.BreastNeoplasm$
$\exists associatedObservation.ColonNeoplasm$
$\exists associatedObservation.$ $\begin{pmatrix} PneumococcalPneumonia \\ \sqcap \\ \exists hasSpecimenSource.Blood \sqcup Sputum \end{pmatrix}$
$\exists associatedObservation.$ $\exists roleGroup.$ $\exists administeredSubstance.$ $\exists roleGroup.\exists hasActiveIngredient.Metronidazole$
$\exists associatedObservation.$ $\begin{pmatrix} acuteamebiasis\sqcup \\ giardisis\sqcup \\ cyclosporiasis\sqcup \\ strongloides\sqcup \\ \ldots \end{pmatrix}$
$\exists associatedObservation.$ $\exists roleGroup.$ $\exists administeredSubstance.$ $\exists roleGroup.\exists hasActiveIngredient.cyclosporine \sqcup steroids$
$\exists associatedObservation.$ $\exists roleGroup.$ $\exists administeredSubstance.$ $\exists roleGroup.\exists hasActiveIngredient.corticosteroid \sqcup cytotoxicAgent$

Table 2. DL Queries for Evaluated Clinical Trials

approach (HTime), the time in minutes for our previous approach (Time), the number of refinements with our hybrid approach (HRefinements) and the number of refinements with our previous approach (Refinements). As can be seen from the table, the hybrid approach reduced the number of refinements to 1 in all cases, which reflects the refinement needed to check that there are no additional solutions after the incomplete algorithm has completed (The one case where 0 refinements occurred was because for that specific query, our expressivity checker decided that no refinement was needed given the specific filtered Abox that was built for the query and the Tbox.) The hybrid approach improved our overall query times from 100.4 mins on average with a standard deviation of 113.7, to 15.6, with a standard deviation of 3.5. This is not surprising, given that the entire variability in query answering in our previous approach was due to the number of refinements.

Query	Matched Patients	Time (m)	HTime (m)	Refinements	HRefinements
$NCT00084266$	1052	68.9	17.8	6	1
$NCT00288808$	3127	63.8	11.6	5	0
$NCT00393341$	74	26.4	12.1	2	1
$NCT00419978$	164	31.8	12.4	3	1
$NCT00304382$	107	56.4	15.1	8	1
$NCT00304889$	2	61.4	20.7	3	1
$NCT00001162$	1357	370.8	13.5	58	1
$NCT00298870$	5555	145.5	19.3	8	1
$NCT00419068$	4794	78.8	17.5	5	1

Table 3. Patient Matches for Trial DL Queries for 240,269 Patients

7.2 UOBM

We evaluated our approach on the UOBM benchmark, modified to $\mathcal{SHIN}$ expressivity. This was done by adding a new concept to correspond to each of the nominals in the dataset (e.g. SwimmingClass for Swimming), adding a type assertion for each nominal (e.g., $Swimming : SwimmingClass$), and changing any of the references to nominals in the Tbox to point to the class. Currently, we have evaluated membership query answering, and we tested one membership query for each concept in the benchmark[6], comparing the hybrid approach with our prior techniques. We report results for UOBM size 100—with roughly 7.8 million type assertions and 22.4 million role assertions—and UOBM size 150—with about 11.7 million type assertions and 33.5 million role assertions. The queries naturally fall into three categories:

empty Concepts that have no instances in the Abox.

simple Concepts that have only simple solutions (i.e. reasoning does not require iterative refinement because the justification viewed as a graph does not have path lengths greater than 1).

complex Concepts that have complex solutions (i.e. reasoning requires iterative refinement because the justification viewed as a graph has path lengths greater than 1).

We expect the hybrid approach to benefit only the third category of queries. One complication is that the summary Abox for the UOBM benchmark has a spurious inconsistency induced by the summarization process, so all membership query answering require 2 passes of refinement in order to make the summary consistent.

Table 4 shows results for the 3 query categories for UOBM sizes 100 and 150. The first three columns list the UOBM dataset size, the category of query, and how many such queries there are. For both sizes and each query category, we report the average and standard deviation for the query time and the number of passes of refinement. For both datasets, we timed out queries that took

[6] That is, all classes in the original benchmark. The extra classes introduced by our transformation to SHIN are ignored.

			Time (seconds)				Refinement			
			Original		Hybrid		Original		Hybrid	
Size	Category	Count	Average	Stdev	Average	Stdev	Average	Stdev	Average	Stdev
100	empty	11	214	37	214	19	2	0	2	0
100	simple	43	255	83	265	47	2	0	2	0
100	complex	14	891*	386*	377	105	14*	11*	3	.3
150	empty	11	301	35	347	45	2	0	2	0
150	simple	43	340	88	416	85	2	0	2	0
150	complex	14	1368*	508*	647	198	14*	11*	3	.3

Table 4. Results for UOBM Membership Queries for sizes 100 and 150

longer than 30 minutes to complete; the timeouts occured on both the 100 size (1 timeout) and the 150 size (6 timeouts) for the original approach. Hence, those averages and standard deviations are significant underestimates, and so are marked with a $*$ in the table.

As one might expect, there is some overhead for executing the incomplete query, and so the simpler queries actually show some slowdown in the hybrid approach. However, the results do indicate that our hybrid approach greatly reduces the time for the complex queries, which were the most expensive ones with our previous approach. In fact, for all but one query, the incomplete reasoning algorithm found all the solutions. The one query which was the outlier, `GraduateCourse`, required propagation from a universal restriction for reasoning, which was not accounted for by our incomplete algorithm. In this case, we proceeded to find the answer through our prior complete reasoning algorithm.

8 Conclusion and Future Work

We have developed an efficient, scalable query answering system for large expressive ABoxes. The hybrid approach proposed in this paper combines our novel summarization and refinement technology to do sound and complete OWL-DL reasoning with any incomplete reasoning implementation (possibly for a subset of OWL).

We have used our hybrid solution to build a web-based semantic search engine for biomedical literature, known as *Anatomy Lens*, details of which can be found in [9]. Anatomy Lens has indexed 300 million RDF triples dealing with PubMed data, and utilizes ontological information from three large biomedical ontologies (Gene ontology, Foundational Model of Anatomy, and MeSH), doing query answering in a few seconds. Performing web-time reasoning for such a large expressive dataset would not have been possible without our approach.

We plan to further optimize our query expansion algorithm by pruning irrelevant queries considering the summary ABox, and to continue to explore the use of SHER in real world semantic web applications.

References

1. C.Patel, J.Cimino, J.Dolby, A.Fokoue, A.Kershenbaum, L.Ma, E.Schonberg, K.Srinivas: Matching patient records to clinical trials. Proc. of the Int. Semantic Web Conf. (ISWC 2007) (2007)
2. A.Fokoue, A.Kershenbaum, L.Ma, E.Schonberg, K.Srinivas: The summary abox: Cutting ontologies down to size. Proc. of the Int. Semantic Web Conf. (ISWC 2006) (2006) 136–145
3. Dolby, J., A.Fokoue, Kalyanpur, A., A.Kershenbaum, L.Ma, E.Schonberg, K.Srinivas: Scalable semantic retrieval through summarization and refinement. Proc. of the 22nd Conf. on Artificial Intelligence (AAAI 2007) (2007)
4. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence IJCAI-05, Edinburgh, UK, Morgan-Kaufmann Publishers (2005)
5. Rosati, R.: On conjunctive query answering in EL, CEUR Electronic Workshop Proceedings (2007)
6. Calvanese, D., Giacomo, G.D., Lembo, D., Lenzerini, M., Rosati, R.: Dl-lite: Tractable description logics for ontologies. Proc. of AAAI (2005)
7. Dolby, J., Fokoue, A., Kalyanpur, A., Ma, L., Schonberg, E., Srinivas, K., Sun, X.: Scalable grounded conjunctive query evaluation over large and expressive knowledge bases. In: ISWC '08: Proceedings of the 7th International Conference on The Semantic Web, Berlin, Heidelberg, Springer-Verlag (2008) 403–418
8. Ma, L., Yang, Y., Qiu, Z., Xie, G., Pan, Y.: Towards a complete owl ontology benchmark. In: Proc. of the third European Semantic Web Conf.(ESWC 2006). (2006) 124–139
9. Dolby, J., Fokoue, A., Kalyanpur, A., Schonberg, E., Srinivas, K.: Scalable highly expressive reasoner (sher). In: Journal of Web Semantics, (accepted), http://dx.doi.org/10.1016/j.websem.2009.05.002 (2009)

A Semantic Web Knowledge Base System that Supports Large Scale Data Integration

Zhengxiang Pan, Yingjie Li and Jeff Heflin

Department of Computer Science and Engineering, Lehigh University
19 Memorial Dr. West, Bethlehem, PA 18015, U.S.A.
{zhp2, yil308, heflin}@cse.lehigh.edu

Abstract. A true Semantic Web knowledge base system must scale both in terms of number of ontologies and quantity of data. It should also support reasoning using different points of view about the meanings and relationships of concepts and roles. We present our DLDB3 system that supports large scale data integration, and is provably sound and complete on a fragment of OWL DL when answering extensional conjunctive queries. By delegating TBox reasoning to a DL reasoner, we focus on the design of the table schema, database views, and algorithms that achieve essential ABox reasoning over an RDBMS. The ABox inferences from cyclic axioms are materialized at load time, while other inferences are computed at query time. Instance data are directly loaded into the database tables. We evaluate the system using synthetic benchmarks and compare performances with other systems. We also validate our approach on data integration using multiple ontologies and data sources.

1 Introduction

The Semantic Web is growing and clearly scalability is an important requirement for Semantic Web systems. Furthermore, the Semantic Web is an open and decentralized system where different parties can and will, in general, adopt different ontologies. Thus, merely using ontologies, does not reduce heterogeneity: it just raises heterogeneity problems to a different level. Without some form of alignment, the data that is described in terms of one ontology will be inaccessible to users that ask questions in terms of another ontology. Our ontology, perspective semantics provides a framework to integrate data sources using different ontologies. This framework uses only standard OWL axioms and hence would not add any additional reasoning complexity to the knowledge base system. Another unique feature of this framework is that different viewpoints regarding the integration (or mapping) could coexist in one knowledge base system, even when they are contradictory. We think this is important for the Semantic Web, which is inherently distributed and inconsistent.

Based upon this framework, we built DLDB3, a scalable Semantic Web knowledge base system that allows queries from different points of view. DLDB3 has major improvements over DLDB2 [16], including a novel approach to handle cyclic (recursive) axioms in relational databases. Although relational databases are optimized for scalable query answering, they usually require special purpose algorithms to handle recursive queries [17]. Our approach identifies cyclic axioms that cannot be directly handled by

precomputed database views and materializes the inferences entailed by these axioms. Our resulting system is complete for 10 of 14 queries in UOBM DL benchmark and all of the LUBM [8] queries, and can load 130 million triples with 24 hours.

The kinds of cycles that lead to incompleteness in a database view approach like DLDB include transitive axioms and property restrictions where the same class appears on both sides of a general concept inclusion axioms (e.g. $\exists P.C \sqsubseteq C$). Wang et al. [20] surveyed 1275 ontologies and only found 39 (3%) that contained a transitive property. In our own analysis of Swoogle's [5] Semantic Web data collection, we could not find any cycles that involves an existential restriction. Only 1.6% out of 16285 ontologies define transitive properties. The number of transitive properties defined in all ontologies is 459, merely 1.4% of the total number of properties. Given that such cycles are rare, we believe that they should be handled as special cases, and our RDBMS-based architecture should be preserved. We hypothesize that selectively materializing these axioms will only lead to minor increases in load time and overall repository size using our approach. In prior work, we added materialization of transitive properties to DLDB2 [16]. In this paper we generalize this to materialize all other cyclic axioms. A key element to this approach is ensuring that the materialized data only appears in the appropriate perspectives.

In what follows, we first introduce ontology perspectives. We then present the DLDB3 system's reasoning algorithms, architecture, design and implementation with a focus on how the reasoning is achieved and perspectives are supported. Finally we evaluate the system using benchmarks as well as multi-ontology data sets and queries.

2 Ontology Perspectives

In prior work, we have defined ontology perspectives which allows the same set of data sources to be viewed from different contexts, using different assumptions and background information [9]. That work also presents a model theoretic description of perspectives. In this section, we set aside the versioning issues of that paper and introduce some essential definitions.

Each perspective is based on an ontology, hereafter called the basis ontology or base of the perspective. By providing a set of terms and a standard set of axioms, an ontology provides a shared context. Thus, data sources that commit to the same ontology have implicitly agreed to share a context. When it makes sense, we also want to maximize integration by including data sources that commit to different ontologies.

We now provide informal definitions to describe our model of the Semantic Web. A Semantic Web space $\mathcal{W}$ is a pair $\langle \mathcal{O}, \mathcal{S} \rangle$, where $\mathcal{O}$ is a set of ontologies and $\mathcal{S}$ is a set of data sources. An ontology O in $\mathcal{O}$ is a four-tuple $\langle \mathcal{C}, \mathcal{R}, \mathcal{T}, \mathcal{E} \rangle$, where $\mathcal{C}$ is a set of concepts; $\mathcal{R}$ is a set of roles; $\mathcal{T}$ is a TBox that consists of a set of axioms; $\mathcal{E} \subset \mathcal{O}$ is the set of ontologies that are extended by O. Note extension is sometimes referred to as inclusion or importing.

An ancestor of an ontology is an ontology extended either directly or indirectly by it. If O_2 is an ancestor of O_1, we write $O_2 \in anc(O_1)$. Note the ancestor function returns the extension closure of an ontology, which does not include the ontology itself.

For ontology extension to have its intuitive meaning, all models of an ontology should also be models of every ontology extended by it. Here we assume that the models of $\mathcal{T}$ are described by the semantics of OWL, for example see [10]. We now define the semantics of a data source.

Definition 1 *A data source s is a pair $\langle \mathcal{A}, O \rangle$, where $\mathcal{A}$ is an ABox that consists of a set of formulas and O is the ontology that s commits to. A model of s is a model of both $\mathcal{A}$ and O.*

When a data source commits to an ontology, it has agreed to the terminology and definitions of the ontology. It means that for data source $s = \langle \mathcal{A}_s, O_s \rangle$, all the concepts and roles that are referenced in $\mathcal{A}_s$ should be either from the ontology O_s or $anc(O_s)$.

We now define an ontology perspective model of a Semantic Web space. This definition presumes that each ontology can be used to provide a different viewpoint of the Semantic Web.

Definition 2 (Ontology Perspective Model) *An interpretation $\mathcal{I}$ is an ontology perspective model of a semantic web space $\mathcal{W} = \langle \mathcal{O}, \mathcal{S} \rangle$ based on $O \in \mathcal{O}$ (written $\mathcal{I} \models_O W$) iff: 1) $\mathcal{I}$ is a model of O and 2) for each $s = \langle \mathcal{A}_s, O_s \rangle \in \mathcal{S}$ such that $O_s = O$ or $O_s = anc(O)$, $\mathcal{I}$ is a model of s.*

Based on this definition, entailment is defined in the usual way, where $\mathcal{W} \models_O \phi$ is read as "$\mathcal{W}$ O-entails ϕ".

Theoretically, each O-entailment relation (perspective) represents a set of beliefs about the state of the world, and could be considered a knowledge base. Thus, the answer to a Semantic Web query must be relative to a specific perspective. We now define a Semantic Web query.

Definition 3 *Given a Semantic Web Space $\mathcal{W} = \langle \mathcal{O}, \mathcal{S} \rangle$, a Semantic Web query is a pair $\langle O, \rho \rangle$ where $O \in \mathcal{O}$ is the base ontology of the perspective and ρ is a conjunction of query terms q_1,....., q_n. Each query term q_i is of the form x:c or $\langle x, y \rangle$:r, where c is an atomic concept and r is an atomic role from O or ancestor of O and x, y are either individual names or existentially quantified variables.*

An answer to the query $\langle O, \rho \rangle$ is θ iff for each q_i, $\mathcal{W} \models_O \theta q_i$ where θ is a substitution for the variables in ρ.

We argue that our perspectives have at least two advantages over traditional knowledge representation languages. First, the occurrence of inconsistency is reduced compared to using a global view, since only a relevant subset of the Semantic Web is involved in processing a query. Even if two ontologies have conflicting axioms, the inconsistency would only be propagated to perspectives based on common descendants of the conflicting ontologies. Second, the integration of information resources is flexible, i.e. two data sources can be included in the same perspective as long as the ontologies they commit to are both being extended by a third ontology.

3 A Scalable Algorithm for Semantic Web Query Answering

Our approach was inspired and based on the work of Description Horn Logic (DHL)[7], a fragment of DL that can be translated into logic programming. Although the DHL

work has a general description on how the datalog programs can be implemented on relational databases, details or working algorithms are not present, especially for handling (cyclic) recursive rules. To our best knowledge, none of the publicly available Semantic Web knowledge base systems has took this route of combining datalog programs with relational databases. Although deductive databases directly implement datalog, their techniques are currently not mature enough for handling large scale data.

3.1 Defining the Language

The DHL language and its mapping to other formalisms has been described in detail in [7]. Here we provide a quick summary for the convenience of discussion. Formally, in DHL:

Concepts are defined as	Roles are defined as
$D ::= A \mid D_1 \sqcap D_2 \mid \forall R.D$	$R ::= P \mid P^-$
$C ::= A \mid \exists R.C \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2$	Where P denotes atomic role.
Where A denotes atomic concept.	
The axioms have form:	The assertions have form:
$C \sqsubseteq D$	$a : C$
$R_1 \sqsubseteq R_2$	$(a, b) : R$
$R^+ \sqsubseteq R$ means R is a transitive property	where a, b are named individuals.
$Func(R)$ means R is a functional property [1]	

Translation input	**Translate to**
Trans(A, x)	$A(x)$
Trans($C \sqsubseteq D, x$)	Trans(C, x)→ Trans(D, x)
Trans($C_1 \sqcap C_2$, x)	Trans(C_1, x) ∧ Trans(C_2, x)
Trans($C_1 \sqcup C_2$, x)	Trans(C_1, x) ∨ Trans(C_2, x)
Trans($\exists R.C$, x)	Trans(R, x, y) ∧ Trans(C, y)
Trans($\forall R.D$, x)	Trans(R, x, y) → Trans(D, y)
Trans($R_1 \sqsubseteq R_2$, x, y)	Trans(R_1, x, y) → Trans(R_2, x, y)
Trans($R^+ \sqsubseteq R$)	Trans(R, x, y) ∧ Trans(R, y, z) → Trans(R, x, z)
Trans(P^-, x, y)	Trans(P, y, x)
Trans(P, x, y)	P(x, y)

Table 1. Translation Function from DL axioms to Horn rules

The DL constructors and axioms can be recursively mapped to First Order Logic rules. By applying Lloyd-Topor transformation to rewrite rules with conjunctions in the

[1] This feature will be handled separately in our system, and will not be translated into horn rules.

head and disjunctions in the body, we can ensure the resulting rules are all horn rules. The equivalences can be rewritten into two subsumptions.

3.2 Reasoning

Our approach uses a relational database to store, manage and retrieve instance data. For each predicate P (class or property) that is defined in an Ontology, we create a dedicated table P_{table} to store the explicit facts. In order to reflect the perspective, we use P^O_{table} to represent the explicit or materialized facts of P that committed to ontology O or $anc(O)$. Note in actual implementation, P^O_{table} doesn't have to be a real table in the database, it can be implemented as a filter of ontology commitment on top of P_{table}. When facts are stored as tuples in the database, their "provenance" or "source" information are also preserved in the form of the identifier of the ontology they commit to. We use P^O_{view} to represent all instances of P, explicit or implicit, from the perspective based on O. For convenience, we define extensional disjunctive normal form.

Definition 4 *A logical formula in first order logic is in extensional disjunctive normal form (E-DNF) if and only if it is a disjunction of one or more conjunctions of one or more atoms, where all predicates correspond to extensional tables.*

Algorithm 1 shows how the reasoning is conducted in our approach. First, we convert axioms in a given ontology into a set of horn rules using the translation function defined above. Note the DL axioms can be reasoned and enriched by a DL reasoner, but it is not necessary in our algorithm. Then for a set of horn rules with a common head, we convert them into a single FOL rule, where the left hand side is in E-DNF. This conversion is processed recursively by applying Modus Pones in a reverse direction until all predicates are primitive (correspond to extensional tables). Next, we separate acyclic portion of disjuncts from the cyclic portion. For acyclic rules, we create view for the head's predicate using straightforward translation from rule to SQL, where conjunctions correspond to joins on common variables and disjunctions correspond to UNION in SQL. Each ontology has a distinct view for each predicate defined by it or one of its ancestors. Periodically when data is being loaded, we use Algorithm 2 to handle cyclic rules left from Algorithm 1. During the computation, we materialize new tuples in the tables so that the computation does not need to be invoked at query time. The materialization would also set the ontology commitment information to be the ontology that invokes this round of computation, such that these "derived" facts can be correctly managed using perspectives. When answering extensional conjunctive query $\langle O, \rho \rangle$ as defined in section 2, each predicate P is directly translated into a view P^O_{view} that not only contains the implicit facts through subsumptions, but also the explicit and materialized facts in the underlying table.

Theorem 1 *Given a knowledge base consists of ontologies and data sources in DHL, the reasoning algorithms described above is sound and complete w.r.t any Semantic Web Query $\langle O, p \rangle$.*

PROOF. (Sketch) When we load an ontology O, the axioms of O and $anc(O)$ are used to generate database views for the perspective based on O. This is consistent with the

Algorithm 1 Load an ontology into the knowledge base

LOADONTOLOGY(O)

1: Translate axioms in O and $anc(O)$ into a set of horn rules H
2: Initialize sets F, F*, each holds a set of FOL rules
3: **for** each predicate P in H **do**
4: Convert the set of horn rules whose heads' predicate is P to L, where the body of L is a E-DNF FOL formula
5: **for** each disjunct B in L such that one of the predicates is P but with different arguments **do**
6: remove B from L and add B as a disjunct into the body of L^* where the head of L^* is also P
7: $F = F \sqcup L$, $F^* = F^* \sqcup L^*$
8: **end for**
9: **end for**
10: **for** each FOL rule $L \in F$ **do**
11: create view P^O_{view} as a SQL expression that joins the conjuncts and unions the disjuncts. Each predicate A in the body of L is translated into A^O_{table}.
12: **end for**

Algorithm 2 Fix point computation of cyclic rules

FIXPOINTCOMPUTE(F^*, O)

1: **repeat**
2: **for** each FOL rule L^* in F^*, where the head's predicate is P **do**
3: Translate the body of L^* into SQL query q, where each predicate A are replaced by views A^O_{view}
4: Execute q, add new tuples into P^O_{table}
5: **end for**
6: **until** A fix point has reached, which means none of the iterations generates new tuples

Ontology Perspective Model in 2. It has been shown in [7] that the translation from DL axioms in DHL to horn rules preserves semantic equivalence. Further on, the conversion to extensional disjunctive normal form in essence is backward chaining and syntactical rewriting of rules by applying Modus Pones in a reverse direction. This kind of conversion does not alter their semantics and logical consequences in FOL. The separation of acyclic rules from cyclic rules is a variation of Lloyd-Topor transformation for disjunctions in the body. Thus again, the changes are only syntactic. For the acyclic rules, their correspondence in SQL has been shown in previous work such as [19]. For the cyclic rules, the correctness of fix point algorithms has also been proved in [19]. To summarize, the query on each database view of a predicate A would get the exact same set of facts as a sound and complete reasoning algorithm would infer for $A(x)$ since the algorithms behind the view exercise all and only the axioms in the knowledge base that the perspective represents.

4 Implementation of DLDB3 System

4.1 Architecture

DLDB3 is a knowledge base system that combines a relational database management system with additional capabilities for partial OWL reasoning. It is freely available as an open source project under the HAWK framework [2].

The DLDB3 core consists of a Load API and a Query API implemented in Java. Any DL Implementation Group (DIG) compliant DL reasoner and any SQL compliant RDBMS with a JDBC driver can be plugged into DLDB3. This flexible architecture maximizes its customizability and allows reasoners and RDBMSs to run as services or even clustered on multiple machines.

It is known that the complexity of complete OWL DL reasoning is NEXPTime-complete. Our pragmatic approach is to trade some completeness for performance. The overall strategy of DLDB3 is to find the ideal balance of precomputation of inference and run-time query execution via standard database operations. The consideration behind this approach is that DL reasoners are optimized for reasoning over ontologies, as opposed to instance data.

Following our algorithm described in the last section, creating tables corresponds to the definition of classes or properties in ontology. Each class and property has a table named using its URI.

Normally, the 'sub' and 'obj' fields are foreign keys from the 'ID' field of the class tables that are the domain or range of the property, respectively. However, if the property's range is a declared data type, then the 'object' field is of the corresponding data type (RDF and OWL use XML Schema data types). Currently DLDB3 supports integer, float and date in addition to string.

DLDB3's table schema is different from the vertical (also called "schema-oblivious") approach [18], which uses a single table for storing both RDF/S schemata and resource descriptions under the form of triples (subject-predicate-object). Note, when new ontologies are loaded, the correspondent tables are created for their classes and properties.

We think the table design of DLDB3 has two advantages over the vertical approach. First, it will contain smaller tables than the vertical scheme. Second, it preserves the data types of the properties and hence can directly and efficiently answer queries involving numeric comparison, such as finding individuals whose ages are under 21. Compared to the traditional relational model, where properties correspond to columns, multi-valued properties (attributes) in DLDB3 don't need to be identified at ontology design time.

In addition to the basic table design, some details should be taken into account when implementing the database schemas for the system. First, we need a scheme for naming the class and property tables. Note, using the full URI will not work, because these URIs often exceed the RDBMS's limits on the length of table names. However, the local name is also insufficient because many ontologies may include the same local name. So we assign a unique sequence number to each loaded ontology. Then each table's name is a class or property's local name plus its ontology's sequence number. This is supported by an extra table:

[2] http://swat.cse.lehigh.edu/downloads/hawk.html

`ONTOLOGIES_INDEX(Url, SeqNum)`
that is used to register and manage all the ontologies in the knowledge base. The sequence number will be assigned by the system when an ontology is first loaded into the database.

Since each row in the class tables corresponds to an instance of the class, an 'ID' field is needed here to record the ID of the instances. The rest of the data about an instance is recorded using the table per property (a.k.a. decompositional) approach. Each instance of a property must have a subject and an object, which together identify the instance itself. Thus the 'sub' and 'obj' fields are set in each table for property.

Sometimes it is important to know which document a particular statement came from, or how many documents contain a particular statement. We support this capability by including a 'Src' field in each class and property table. Together with other fields, it serves as the multiple-field primary key of the table. In other words, the combination of all the information of one record identifies each instance stored in the knowledge base. In order to support perspectives described in section 2, the ontology to which the individual data commits is also needed. Thus an 'Onto' field is include in the tables. This field is used to record the committed ontology from Algorithm 1 and 2. An example of class and property tables might be:
`STUDENT:1(Id, Src, Onto)`
`TAKESCOURSE:1(Sub, Obj, Src, Onto)`

In order to shrink the size of the database and hence reduce the average query time, DLDB assigns each URI a unique numeric ID in the system. We use a table:
`URI_INDEX(Uri, Id)`
to record the URI-ID pairs. Thus, for a particular resource, its URI is only stored once; its corresponding ID number will be supplied to any other tables. Since the typical URI is often 20-50 characters long and the size of an integer is only 4 bytes in the database, this leads to a significant savings in disk space. Furthermore, query performance is also improved due to the faster joins on integers instead of strings and the reduced table size. By discriminating the DataType properties and ObjectType properties, the literals are kept in their original form without being substituted by ID numbers. The reason why we don't assign IDs to literals is 1) literals are less likely to be repeated in the data; and 2) literals are less likely to be joined with other tables because they are never used as database keys.

Unsurprisingly, the tradeoff of doing the URI-ID translation is an increase in load time. We use a hash table to cache URI-ID pairs found recently during the current loading process. Since URIs are likely to repeat in one document or neighboring documents, this cache saves a lot of time by avoiding lookup queries when possible.

When DLDB3 loads an ontology, it uses Algorithm 1 to do reasoning. Note if there are instance data in the ontology, they are processed in the same way as if they come from a data source which commits to this ontology.

It is worth noting that DLDB3 uses a DL reasoner to make inferences on DL axioms before these axioms are translated into horn rules. In result, although the horn rules implemented in the DLDB3's relational database system correspond to a subset of DHL, DLDB3 does support reasoning on DL ontologies richer than DHL. For example, the axioms $A \sqsubseteq B \sqcup C$ and $A \sqsubseteq \neg B$ are both beyond the expressiveness of DHL. However,

the DL reasoner will compute and return $A \sqsubseteq C$ assuming A and C are both atomic classes. Unfortunately, it is difficulty to characterize this expressivity formally since some other axioms involving disjunction or negation are not supported.

In general, data loading in DLDB3 is straight-forward. Each *rdf:type* triple inserts a row into a class table, while each triple of other predicates inserts a row into a role table corresponding to the predicate. If a data document imports multiple ontologies, the value of the 'Onto' field is decided by the ontology that introduces the term that the table corresponds to. However, DLDB3 materializes certain inferences at data load time, as discussed in the next sections.

4.2 Inference on Individual Equality

This subsection focuses on precomputations that simplify reasoning at query time. These ABox reasoning routines along with the rule-based reasoning algorithm make the system complete on a significant subset of OWL DL.

OWL does not make the unique names assumption, which means that different names do not necessarily imply different objects. Given that many individuals contribute to the Web, it is highly likely that different IDs will be used to refer to the same object. Such IDs are said to be equal. A Semantic Web knowledge base system thus needs an inference mechanism that actually treats them as one object. Usually, equality is encoded in OWL as (a *owl:sameAs* b), where a and b are URIs.

In DLDB3, each unique URI is assigned a unique integer id in order to save storage space and improve the query performance (via faster joins on integers than strings). Our approach to equality is to designate one id as the canonical id and globally substitute the other id with this canonical id in the knowledge base. The advantage of this approach is that there is effectively only one system identifier for the (known) individual, nevertheless that identifier could be translated into multiple URIs. Since reasoning in DLDB3 is based on these identifiers instead of URIs, the existing inference and query algorithms do not need to be changed to support equalities.

However, in many cases, the equality information is found much later than the data that it "merges". Thus, each URI is likely to have been already used in multiple assertions. Finding those assertions is especially difficult given the table design of DLDB3, where assertions are scattered into a number of tables. It is extremely expensive to scan all the tables in the knowledge base to find all the rows that use a particular id, especially if you consider that the number of tables is equal to the number of classes plus the number of properties. For this reason, We use auxiliary tables to keep track of the tables that each id appears in.

Often times, the knowledge on equality is not given explicitly. Equality could result from inferences across documents: *owl:FunctionalProperty*, *owl:maxCardinality* and *owl:InverseFunctionalProperty* can all be used to infer equalities. DLDB3 is able to discover equality on individuals using a simple approach. If two URIs have the same value for an *owl:InverseFunctionalProperty*, they are regarded as representing the same individual. A naive approach is to test for this event every time a value is inserted into an inverse functional property table. However, this requires a large number of queries and potentially a large number of update operations. In order to improve the throughput

of loading, we developed a more sophisticated approach which queries the inverse functional property table periodically during the load. This happens after a certain number of triples have been loaded into the system. The specific interval is specified by users based upon their application requirements and hardware configurations (we used 1.5 million in our experiment). This approach not only reduces the number of database operations, but also speeds up the executions by bundling a number of database operations as a stored procedure. DLDB3 also supports the same approach on *owl:FunctionalProperty*.

4.3 Handling Different Kinds of Cyclic Axioms

Although Algorithm 1 deals with cyclic axioms in general, our implementation handles two categories differently. Class and Property Equalities is a form of cyclic axioms. However, they do not need fix point computation in our algorithm since they are solved during the atom expansion process. The fundamental difference between these equalities and other forms of cyclic axioms such as the transitive property is that they do not involve self-joinings or iterative procedures. They simply require that we synchronize the subsumptions between two equivalent terms. For named classes and properties, this synchronization has been taken care of by the DL reasoner.

In actual implementation, transitive properties are obvious cyclic axioms and hence they do not need to be identified by translating into FOL rules. Our solution is to periodically run an algorithm that self-joins the view (not the table) on the transitive property iteratively until a fixed point is reached. This algorithm also takes care of the perspectives, which allows different ontologies to independently describe a property as transitive or not. The other forms of cyclic axioms are handled by repeating iterations until no new tuples are generated. For new tuples generated during the iterations, their 'onto' field is set to the value of the ontology that invokes this round of fix point computation (as shown in Algorithm 2).

4.4 Special Handling on Domain and Range

Normally, DHL allows universal restrictions on the right hand side. Domain and range axioms are both special cases of universal restriction ($\top \sqsubseteq \forall P.C$ and $\top \sqsubseteq \forall P^{-}.C$, respectively). The reasoning algorithm would include such rules and their corresponding SQL expressions in the view definition of classes. Our initial implementation experience shows this kind of axioms can lead to efficiency issues at query time. In particular, when the property involved has many triples, this leads to inference of class membership for a large number of instances. However, since OWL-DL requires that every instance has a type, these inferences are possibly redundant with explicit information on the knowledge base. Note, explicit universal restrictions that involve specific classes on the left hand side usually are not as bad as domain and range, simply because the join would reduce the number of facts that need to be compared with existing facts.

In order to improve the query efficiency, DLDB3 handles domain and range axioms at load time. Following the same translation method that translate the Horn rules into SQL expressions, we execute these expressions periodically during load time and effectively materialize the new facts into the corresponding tables. Then at query time, there is no need to invoke the inferences for domain and range axioms. Our initial analysis

has shown that this special treatment for domain and range axioms can improve the overall performance of the system, considering the same domain or range class could be queried over and over. Note this special handling would not alter the soundness and completeness of the reasoning algorithm.

4.5 Query Answering

The query API of DLDB3 currently supports SPARQL encodings of conjunctive queries as defined in section 2. During execution, predicates and variables in the query are substituted by table names and field names through translation. Depending on the perspective being selected, the table names are further substituted by corresponding database view names. Finally, a standard SQL query sentence is formed and sent to the database via JDBC. Then the RDBMS processes the SQL query and returns appropriate results.

Since we build the class and property hierarchy when loading the ontology, there is no need to call the DL reasoner at query time. The results returned by the RDBMS can be directly served as the answer to the original query. We think this approach makes the query answering system much more efficient than conducting DL reasoning at query time. To improve the query performance. DLDB3 system independently maintains indexes without the intervention from database administrators.

5 Related Work

The C-OWL work [2] proposed that ontologies could be contextualized to represent local models from a view of a local domain. The authors suggested that each ontology is an independent local model. If some other ontologies' vocabularies need to be shared, some bridge rules should be appended to the ontology which extends those vocabularies. Compared to C-OWL, our perspective approach also provides multiple models from different views without modifying the current Semantic Web languages.

Our work differs from deciding the conservative extensions [14] in that we do not study the characteristics of the logical consequences of integrating two ontologies. Instead, we focus on how to efficiently integrate data that commits to those ontologies assuming that the logical consequences have been explored and approved by the user.

In order to improve the scalability of ABox reasoning, a number of "tractable fragments" of OWL have been proposed and studied. Compare to DHL, DL-Lite [3] supports a limited form of existential restrictions on both sides of class inclusion. It also supports negation on classes and properties. However, it does not support transitive properties. EL++ [1], on the other hand, supports qualified existential restrictions and negation on classes but does not support inverse properties. Both DL-lite and EL++ do not support universal restrictions. In terms of expressiveness, DHL is more or less close to those subsets of OWL DL. It is noteworthy that in the upcoming W3C recommendation OWL 2, EL++ and DL-Lite are the bases of EL profile and QL profile respectively.

In recent years there has been a growing interest in the development of systems that will store and process large amount of Semantic Web data. The general design goal of these systems is often similar to ours, in the sense that they use some database systems

to gain scalability while supporting as much inference as possible by processing and storing entailments. However, most of these systems emphasize RDF and RDF(S) data at the expense of OWL reasoning. Some systems resemble the capabilities of DLDB3, such as KAON2 [11], which uses a novel algorithm to reduce OWL DL into disjunctive datalog programs. SwiftOWLIM [12] uses a rule engine to support a limited OWL-Lite reasoning. SOR [13] uses DL reasoner to do TBox reasoning and a rule engine to do ABox reasoning. SHER [6] aims at scalable ABox reasoning using a summary of ABox. To the best of our knowledge, none of the systems above supports queries from different perspectives.

6 Evaluation

6.1 Performance on Benchmarks

We evaluated DLDB3 using LUBM [8] and UOBM (DL)[15]. UOBM extends LUBM with additional reasoning requirements and links between data. The evaluation is done on a desktop with P4 3.2G CPU and 3G memory running Windows XP professional. We configured DLDB3 to use Pellet 2.0 as its DL reasoner and MySQL 5.0 community version as its backend RDBMS. For comparison, we also tested SHER, SOR (version 1.5), KAON2 and SwiftOWLim (v2.9) on the same machine. IBM DB2 Express-C V9.5 was used as SHER and SOR's backend database.

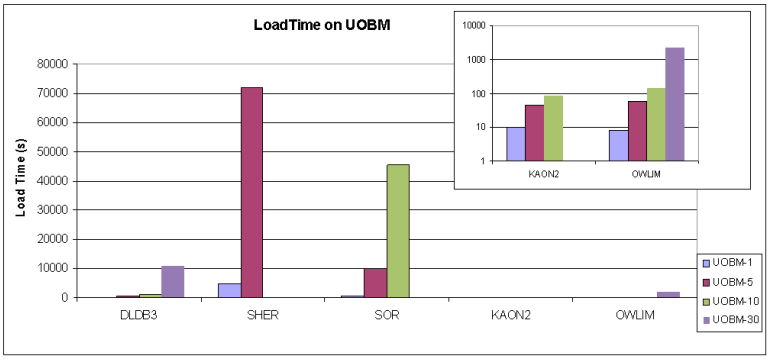


Fig. 1. Load time on Benchmarks

Note UOBM only provides datasets in four sizes and has no publicly available data generator. The largest dataset, UOBM-30 has 6.6 million triples. In our experiment, DLDB3 can load 130 million triples from LUBM(1000,0) with 24 hours and be complete on all the queries in LUBM.

The smaller diagram on Figure 1 shows the load time of KAON2 and SwiftOWLim on UOBM. These two systems are memory based so that they are fast at loading. However, their scalability are limited by the memory size. KAON2 could not finish reasoning

on UOBM-10 in our experiment. SHER, SOR and DLDB3 all use a backend RDBMS and hence would scale better than memory based systems. DLDB3 is faster on loading than SOR and SHER. It is reasonable for SOR since it materializes all inferred facts at load time. For reasons that we cannot explain, SHER failed to load datasets larger than UOBM-5. SOR did not finish the loading of UOBM-30 within a 72 hours period in our experiment.

Figure 2 shows the average query response time (average on 14 queries) for all systems on UOBM. DLDB3 is faster than KAON2 and SHER on all datasets, and keeps up with SwiftOWLim as the size of the knowledge base increases. The standard deviation on query response times gives no surprise: DLDB3 has higher variation than SwiftOWLim. For DLDB3, all the queries can be finished under 4 minutes across the datasets.

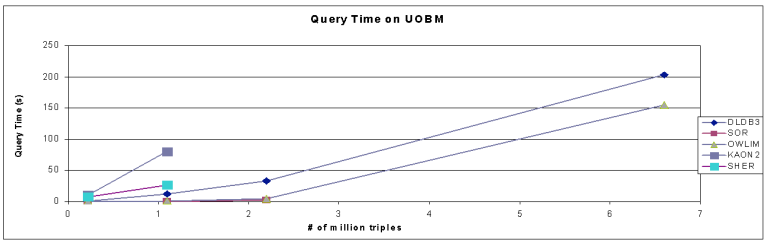


Fig. 2. Query Response Time on UOBM

As shown in Table 2, DLDB3 is complete on 11 out of the 14 queries in UOBM. Two of the queries (Q12 and Q14) are not complete due to cardinalities. Another query (Q13) involves inference using negation. SwiftOWLim is incomplete on only one query (Q13); SOR is incomplete on two queries (Q12 and Q14) and SHER is only complete on 6 queries.

	Q1 - Q11	Q12	Q13	Q14
DLDB3	100%	80%	96%	0%
SOR	100%	63%	100%	63%
SwiftOWLim	100%	100%	96%	100%

Table 2. Completeness on UOBM-1

6.2 Multi-ontology Evaluation

Both LUBM and UOBM only have a single ontology in their test dataset, which means they cannot be used to test the system's capability on data integration. In addition,

both benchmark ontologies contain no cycles besides transitive properties. In order to empirically validate our implementation on perspectives and cyclic axiom handling, we used a synthetic data generator to generate a multi-ontology test dataset. The details about the data generator is described in [4]. The dataset we chose features 10 domain ontologies and 10 mapping ontologies that map between the domain ontologies, the expressivity of the ontologies was limited to DHL. There are 20,000 data files and about 1 million triples in total. 10 random generated queries associated with one particular ontology were picked as test query. Note the ontologies in this dataset have a number of cyclic axioms, some of them form cycles across ontologies.

The experiment set-up was the same as the UOBM benchmark described above. KAON2 and DLDB3 were tested using this multi-ontology dataset. Since KAON2 is proved to be sound and complete on DHL, the results of KAON2 is used as reference. In order to verify the correctness of DLDB3, each test queries was issued using different perspectives. For a query $\langle O, \rho \rangle$, the reference result sets were collected by loading the ontologies and data sources that are included in the perspective model based on O (see definition 2) into KAON2, and then issue the conjunctive query ρ to KAON2.

All the query results from DLDB3 match the references from KAON2. This experiment has shown that DLDB3 is sound and complete w.r.t the logical consequences defined in section 2, and correctly implements the algorithm that handles cyclic axioms.

We also did some initial analysis on the scalability of the perspective approach. Figure 3 shows that as the number of ontologies included (though mappping) in the perspective increases, the query response time would increase. However, more ontologies bring more results to the query, which at large extent justifies the increase of response time.

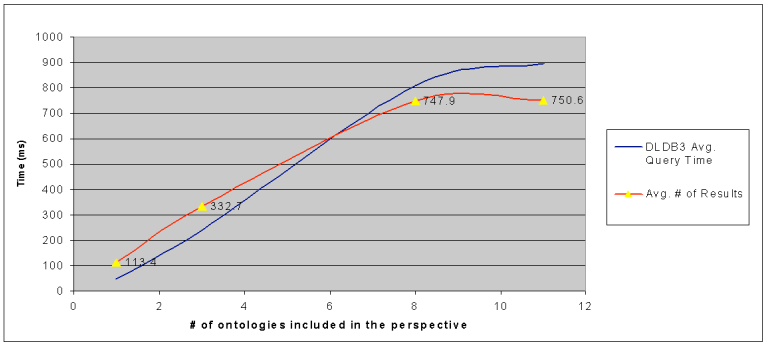


Fig. 3. Number of Ontologies V.S. Avg. Number of Query Results

On the other hand, as shown in Figure 4 the depth of the mapping (the maximum length of the mapping chain from the query term to the term that data commits to) only has small impact on the query response time. Again, when compared with the number of results, the increase of query response time is justified. Overall, we have seen that the

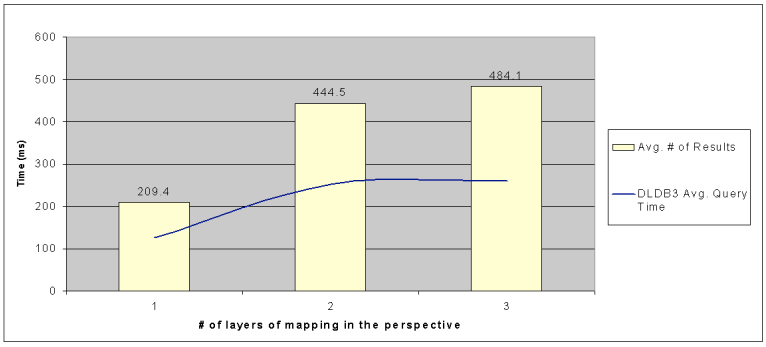


Fig. 4. Depth of Mapping V.S. Avg. Number of Query Results

query performance of DLDB3 would degrade gracefully as the depth or breadth of the perspective increases, but largely due to the gaining of new results through integration.

7 Conclusion and Future Work

In this paper we present our DLDB3 system, which takes advantage of the scalability of relational databases and the inference capability of description logic reasoners. Our scalable querying answering algorithm is guaranteed to be sound and complete on DHL. Our evaluation shows that DLDB3 scales well both in load and query comparing to other systems. It has achieved a good balance between scalability and query completeness. Based on ontology perspectives which use existing language constructs, DLDB3 can support queries from different view points. Real-world data using multiple ontologies and realistic queries show that DLDB3 has achieved this capability without any significant performance degradation.

Although we believe our work is a first step in the right direction, we have discovered many issues that remain unsolved. First, to ensure the freshness of data, we plan to support efficient updates on documents. Second, we will investigate query optimization techniques that can improve the query response time. Third, we will evaluate our system's capability on perspectives more extensively and comprehensively using real-world datasets.

8 Acknowledgment

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. IIS-0346963. The authors would like to thank Tim Finnin of UMBC for providing access to the Swoogle's index of URLs. Graduate students Abir Qasem also contributed to the evaluations in this paper.

References

1. F. Baader, S. Brand, and C. Lutz. Pushing the el envelope. In *In Proc. of IJCAI 2005*, pages 364–369. Morgan-Kaufmann Publishers, 2005.
2. P. Bouquet, F. Giunchiglia, F. van Harmelen, L. Serafini, and H. Stuckenschmidt. C-OWL: Contextualizing ontologies. In *Proc. of the 2003 Int'l Semantic Web Conf. (ISWC 2003)*, LNCS 2870, pages 164–179. Springer, 2003.
3. D. Calvanese, D. Lembo, M. Lenzerini, and R. Rosati. Tailoring owl for data intensive ontologies. In *In Proc. of the Workshop on OWL: Experiences and Directions*, 2005.
4. A. Chitnis, A. Qasem, and J. Heflin. Benchmarking reasoners for multi-ontology applications. In *In Proc. of Workshop on Evaluation of Ontologies and Ontology-Based Tools*. ISWC 07, 2007.
5. L. Ding, T. Finin, A. Joshi, Y. Peng, R. Pan, and P. Reddivari. Search on the semantic web. *IEEE Computer*, 10(38):62–69, October 2005.
6. J. Dolby, A. Fokoue, A. Kalyanpur, L. Ma, E. Schonberg, K. Srinivas, and X. Sun. Scalable grounded conjunctive query evaluation over large and expressive knowledge bases. In *International Semantic Web Conference*, pages 403–418, 2008.
7. B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *Proceedings of WWW2003*, Budapest, Hungary, May 2003. World Wide Web Consortium.
8. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for owl knowledge base systems. *Journal of Web Semantics*, 3(2):158–182, 2005.
9. J. Heflin and Z. Pan. A model theoretic semantics for ontology versioning. In *Proc. of the 3rd International Semantic Web Conference*, pages 62–76, 2004.
10. I. Horrocks and P. F. Patel-Schneider. Reducing OWL entailment to description logics satisfiability. In *Proceedings of the Second International Semantic Web Conference*, pages 17–29, 2003.
11. U. Hustadt, B. Motik, and U. Sattler. Reducing SHIQ description logic to disjunctive datalog programs. In *Proc. of the 9th International Conference on Knowledge Representation and Reasoning*, pages 152–162, 2004.
12. A. Kiryakov. OWLIM: balancing between scalable repository and light-weight reasoner. In *Developer's Track of WWW2006*, 2006.
13. J. Lu, L. Ma, L. Zhang, J.-S. Brunner, C. Wang, Y. Pan, and Y. Yu. Sor: a practical system for ontology storage, reasoning and search. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1402–1405. VLDB Endowment, 2007.
14. C. Lutz, D. Walther, and F. Wolter. Conservative extensions in expressive description logics. In *In Proc. of IJCAI-2007*, pages 453–459. AAAI Press, 2007.
15. L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu. Towards a complete OWL ontology benchmark. In *ESWC*, pages 125–139, 2006.
16. Z. Pan, X. Zhang, and J. Heflin. Dldb2: A scalable multi-perspective semantic web repository. In *Web Intelligence*, pages 489–495, 2008.
17. G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory Pract. Log. Program.*, 8(2):129–165, 2008.
18. Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of rdf/s stores. In *Proc. of the 4th International Semantic Web Conference*, pages 685–701, 2005.
19. J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Rockville, MD, 1988.
20. T. D. Wang, B. Parsia, and J. Hendler. A survey of the web ontology landscape. In *Proc. of the 5th Int. Semantic Web Conference (ISWC 2006), Athens, Georgia*, 2006.

Representing and Integrating Light-weight Semantic Web Models in the Large

Matteo Palmonari, Carlo Batini

Department of Computer Science, Systems and Communication (DISCo)
University of Milan - Bicocca
via Bicocca degli Arcimboldi, 8
20126 - Milan (Italy)
tel +39 02 64487857 - fax +39 02 64487839
{palmonari,batini}@disco.unimib.it

Abstract. Semantic Web model representation and integration can be exploited to provide organizations that deal with a large amount of data sources with an integrated view on the overall information managed. In order to support semantic Web model representation and integration in the large users must be provided with light-weight languages to represent and integrate the models, in particular avoiding the design of complex Tbox axioms. Assuming to adopt at the front-end level graph-based Concept-to-Concept Relationship (CCR) representations, in this paper we question about two semantic issues. First, we inquire whether light-weight semantic Web languages such as RDFS and DL-Lite can be used to provide the semantics of individual CCR models. Second, we inquire whether these languages can be used to provide appropriate semantics for the mappings needed for model integration. Discussing a case study in the eGovernment domain we claim that both the answer are negative. Therefore, we propose a new semantic interpretation for CCR models and we define three main classes of integration and abstraction relations defining their semantics.

1 Introduction

Semantic Web technologies and languages such as RDF, RDFS and OWL provide knowledge sharing and logical modeling capabilities based on ontologies [1], and techniques to achieve data and schema integration [2]. Web ontologies (ontologies represented in a semantic Web language) can support meta-data management, but also different applications targeted to data integration, document management, or service provision [1] by representing Web-compliant conceptual model referring to logical and conceptual schemata. Moreover, Web ontologies map to different data models of information sources, ranging from XML, to RDBMS [2]. The level of expressiveness ranges from light-weight ontologies, with taxonomies as the least expressive one, to heavy-weight ontologies with very-expressive constraints as the most expressive representative [3]. Scalability in Semantic Web ontologies is related to at least two problems. A first problem is related to the expressiveness/complexity trade-off w.r.t. reasoning: roughly speaking, the more the language of an ontology is expressive the more complex is reasoning on the ontology [1]. A second scalability problem is related to an

expressiveness/cost trade-off in ontology management and engineering according w.r.t. a cost/benefit model: roughly speaking, the more a language is expressive the more it costs to manage and engineer it (design, development, maintenance and so on) [4–6, 3].

This paper focuses on the latter scalability problem in semantic Web ontologies, which is gaining more and more attention in the last few years. A number of studies showed that rich Web ontologies represented in languages like OWL-DL are too costly in the large and are difficult to use for people with little formal background [5]; in particular, mastering the complex Tbox-level OWL axioms' semantics can be difficult and impacts on a number of ontology engineering costs [4]. RDFS is simpler and easier than OWL, as proved by the number of RDFS ontologies actually published on the Web [5]; in this paper, we refer to RDFS as the main light-weight semantic Web language. DL-Lite [7] provides the logical foundation for the OWL 2 QL profile of OWL 2, which is another semantic Web language that we consider to a certain extent light-weight (we will refer to this language as DL-Lite throughout the paper); in fact, it is more expressive than RDFS but by far less expressive than OWL-DL. Ontology frameworks such as semantic wikis [8, 9], which are explicitly targeted to support end-users in collaborative ontology design, provide tools to simplify the design process; in these tools a simplified syntax for the specification of simple ontology axioms (domain and range restrictions on properties) prevent users from defining complex axioms using quantifiers and complex concept constructors.

Based on the above considerations, it seems that the languages/tools that are more used in fact by non skilled ontology designers, e.g. RDFS and semantic wikis, tend to present ontologies, at the front-end level, as graphs where nodes represent concepts and arcs represent relationships among these concepts in this paper we will call these models Concept-To-Concept Relationship (CCR) models. Different languages or subsets of them isomorphic to CCR models (e.g. RDFS, and DL-Lite, the Semantic Media Wiki syntax), or visual interfaces based on graphs or quantifiers-free forms (e.g. [9]) can be considered as front-end concrete languages for light-weight ontology design.

Of course the expressiveness/cost trade-off need to be considered w.r.t. a cost/benefit model, where the benefit depends on specific application contexts (e.g. DL-Lite is particularly useful for vertical data integration applications with few information sources because of its good computational properties). In this paper, we focus on contexts where conceptual models of many different sources, semantically heterogeneous and referring to different domains need to be represented and integrated; as an example consider to represent and integrate about 500 models representing the databases of the Italian public administrations. This scenario is typical when large organizations need to be provided with an overall view of the information they manage, and of their semantics, to improve the government of their data. We refer to this context as to *conceptual schema representation and integration in the large*. In this context, we often call conceptual schemata, or *schemata* for short, the semantic Web models to represent and integrate.

Assuming to represent light-weight ontologies as isomorphic to CCR schemata at the front-end level, this paper questions whether the current available semantic Web language are suitable in this context. In particular; a first research question considered in the paper is the following "in the context of conceptual schema representation and integration in the large, are the available light-weight semantic Web languages and their

traditional semantics appropriate for the representation of each CCR schema?" (Q1). A second research question considered in the paper is "in the context of conceptual schema representation and integration in the large, are the current light-weight semantic Web languages, and their traditional semantics, suitable for designing the integration of such schemata?" (Q2).

By discussing a case study in conceptual schema representation and integration in the large in the eGovernment domain, the paper argues that the answer to both the questions is negative. W.r.t. Q1, the interpretation of CCR models based on standard proposal (RDFS, DL-Light) are too restrictive. An alternative semantics for CCR models is proposed to overcome the discussed limitations. W.r.t. Q2, the available languages fails to cover important loose abstraction relations among concepts of different conceptual schemata needed in the integration process. Based on the literature the paper proposes three kinds of integration-abstraction relationships, and discusses their semantics.

The paper is organized as follows: the problem context and the case study are introduced in Section 2; the problems and the proposed solution for individual schema representation and their integration are discussed in Section 3; related works are discussed in Section 4; conclusions end the paper in Section 5.

2 Schema Representation and Integration in the Large

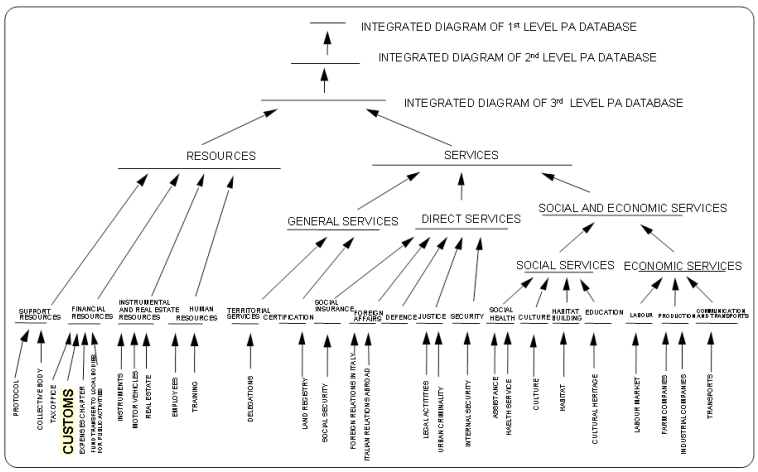


Fig. 1: The multi-layered conceptual schemata' repository of the Italian central public administrations

Schema representation and integration in the large support conceptual meta-data management, in order to provide large organizations, or networked enterprises with an integrated view of the information managed. To make an example we consider a case

study referring to past experiences in the design of repository of conceptual schemata in the eGovernment domain.

Experiences in the design and exploitation of structured repositories of conceptual schemata (Entity Realtionship schemata) related to the most relevant databases of the Italian central public administrations are described in [10]. A Central Public Administration repository of schemata (CPA repository) has been developed to provides public institutions with a conceptual meta-data management framework; the structure of the repository is shown in Figure 1, where each node in the hierarchy represents an ER schema. The bottom level of the CPA repository consists of approximately 500 conceptual schemata representing at the conceptual level logical schemata of the information sources (*basic schemata* - not represented in the figure); these basic schemata are recursively clustered and integrated by exploiting integration and abstraction primitives defined in [11]; the bottom level of the figure shows the conceptual schemata obtained from the basic schemata at the first integration and abstraction step.

The benefits of exploiting structured repositories of schemata at the back-end and at the front-end level, e.g. to improve government-to-citizen and government-to-business relationships, have been discussed in [10]. As for the exploitation of repositories at the application level, the CPA repository has been exploited to support the semi-automatic construction of a Local Public Administration (LPA) repository based on reverse engineering techniques [10]. Building such a repository with current semantic Web languages could bring even more benefits, such as the possibility to exploit the concepts and relationships for semantic annotation and search in SOA, document management or data integration initiatives.

2.1 Conceptual Schema Representation

Focusing on the representation of individual schemata in the repository, the context of conceptual schema integration in the large is characterized by the necessity of representing and integrating many schemata (about 500 in the example), referring to many heterogeneous domains (e.g. financial resources, certifications, justice, security, education, and so on). However, a number of concepts are shared among the different schemata (e.g. the concept of Subject appear in most of the different eGovernment domains); these concepts are the key concepts to integrate the different schemata [10]. Such a scenario requires a lot of effort for ontology design and engineering. It is very difficult to commit such an effort to one skilled and experienced ontology designer. Experts in the domain need to be provided with tools to design their models. Because of the amount of the schemata to design and integrate, and the designers' profiles (domain experts with little formal background), light weight languages are needed to cope with the expressiveness/cost trade-off. Moreover, capturing deep ontological commitments (e.g. with cardinality restrictions) it is often difficult (e.g. at the more abstract levels) and not useful (e.g. to support softer tasks such as navigation, semantic annotation and search rather than more specific reasoning-based tasks such as data integration).

In order to support scalable design of the schamas in the large, it is important to consider the knowledge continuum perspective [3]. According to this perspective, knowledge is represented in a continuum of knowledge artifacts represented in languages with different expressivity. In the context addressed in this paper, focusing on schemata of

semi-structured and structured information sources, this means that light-weight semantic models could be used as sketches to develop more expressive Web ontologies when needed. As an example, suppose that a new eGovernment data integration project targeted to the Human Resources' data sources (see Figure 1) is started: richer ontologies with more expressive axioms might be needed in this case. However, the reuse of the light-weight models defined at the conceptual meta-data management level should be guaranteed: the semantics of such light-weight models need not to be inconsistent with the new axioms. We will refer to this point as to the *knowledge continuum* issue.

Given the amount of schemata to represent, their levels of detail and the need for a collaborative design process carried out by domain experts, the CCR expressivity level can be considered a good compromise between the need of exploiting at the front-end level a language easy to be used, and the need of going beyond taxonomies representing at least the relationships between concepts. Considering the case study discussed, although the CPA repository is based on the ER model, the schemata represented are not far from CCR models (no cardinality restrictions are used and relationships with arity greater than 2 represents less than 5% of the total number of relationships used).

2.2 Conceptual Schema Integration

In the context addressed in the paper (when many schemata are considered) a one-step integration is nearly impossible, and schemata need to be integrated in a recursive way: sets of schemata clustered according to similarity criteria and balanced according to their levels of detail (LOD) are merged together by means of a *schema integration* primitive (see [12] for schema integration mechanisms); the process is iterated on the resulting set of schemata. However, the integration of schemata would easily lead to schema very large in size with difficulties in their management and comprehension. *Schema abstraction* primitives can be applied to obtain a schema at coarser LOD from a given schema. When considering the application of an integration or an abstraction primitive, we will call *source schemata* the schemata to which the primitive is applied, and *target schema* the schema resulting from the application of the primitive; we will call *source concepts* the concepts of the source schemata and *target concepts* the concepts of the target schemata. In practice, integration and abstraction primitives are often applied together to obtain an integrated target schema at a coarser LOD. We will refer to this primitive as *integration-abstraction*; observe that this primitive is more general than abstraction, and the application of an abstraction primitive can be considered a special case of integration-abstraction applied to a single schema. As a result, this iterative integration-abstraction process leads to consider schemata at progressively coarser LOD, that is, that are progressively more abstract.

For the details of the methodology to carry out this iterative integration process we refer to [11]. In order to provide a more detailed example of schema integration-abstraction, we consider the Customs domain highlighted in Figure 1; the conceptual schema of the Custom domain is obtained by means of integration-abstraction mechanisms applied to three basic schemata, namely Custom Agencies, Custom Declarations and Item Categorization. Figure 2 represents a simplified version of this example (the size of the schemata is reduced to illustrate the main points addressed in this paper); broken-lined arrows represent subclass relationships, thick A-arrows represent

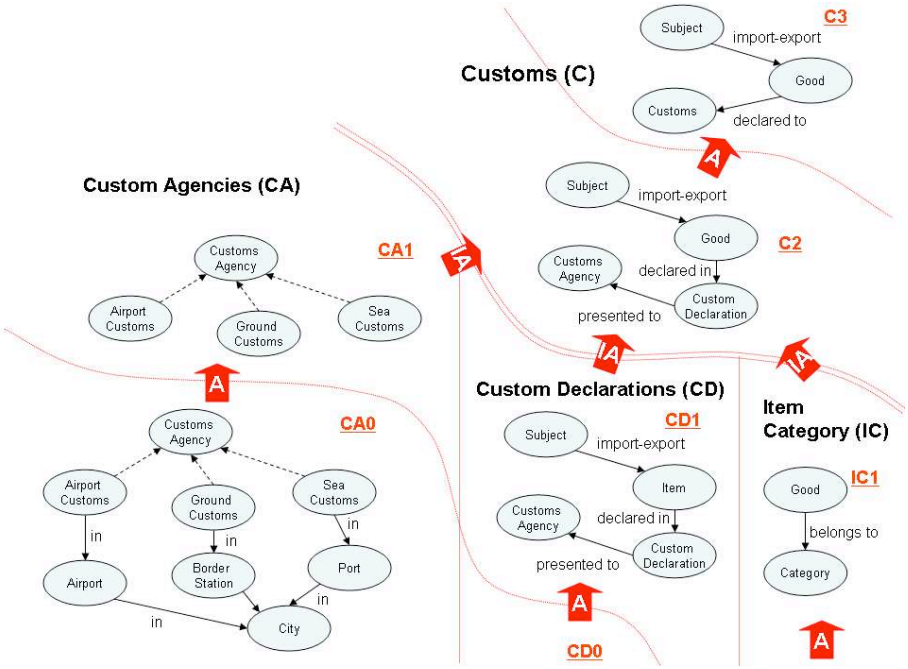


Fig. 2: Examples of integration and abstraction of ER conceptual schemata

schema abstraction primitives, and thick IA-arrows represents integration-abstraction primitives; four LODs are represented (only the schema CA0 - Custom Agency at level 0 - is drawn as representative of the bottom level; the schemata CD0 and IC0 are omitted); the schemata are represented as CCR models. An abstraction primitive is applied to CA0, producing the schema CA1 (where the locations the custom agencies are associated with are discarded). The schemata CA1, CD1 and IC1 are integrated-abstracted in the schema C2. The schema C2 is abstracted into the schema C3, where the two concepts Custom Agency and Custom Declaration are replaced by the concept Customs, in order to express, at a more abstract level, that Goods are declared to Customs.

The question Q2 addressed in the paper concerns the suitability of available lightweight ontology Web languages to represent these kind of integration-abstraction primitives. Assuming to represent each source schema as a Web ontology, standard import and namespace mechanisms in OWL and RDFS easily allows for the integration of schemata at the architectural level (e.g. to define that C2 integrates the three schemata CA1, CD1 and IC1 one could define a Web ontology C2 importing CA1, CD1 and IC1). The question therefore focuses on the representation of the mappings between the schema concepts: are the available language suitable to represent the set of significant relations occurring between the entities of the source and the target schemata? As an example, is it possible to trace out that the specific locations associated with the three types of custom agencies in the schema C0 of Figure 2 are *forgot* in CA1?

3 Semantics for Conceptual Schema Integration Based on CCR Models

In order to better define the problem, we introduce the notion of schema integration-abstraction framework and an abstract CCR formalism.

Given a set of schemata Σ, a *schema integration-abstraction framework* can be formalized by a schema integration-abstraction operation $f^{\Sigma} : \mathcal{P}(\Sigma) \longmapsto \Sigma$; as an example, the schema integration-abstraction framework based on Figure 2 is defined by a f^{Σ} such that $f^{\Sigma}(\{CA0, CD0, IC0\}) = C1$, and $f^{\Sigma}(\{C1\}) = C2$. In the following we assume that the schema integration-abstraction frameworks are *partitive*, that is, each schema is integrated-abstracted in only one schema).

The abstract CCR formalism is defined through a language (CCR language from now on) that represents concepts, binary relationships between concepts and subconcept relationships; moreover, since the notion of inverse property is quite intuitive for binary directed relationships, the notion of inverse property is introduced (scalability w.r.t. reasoning is not addressed in this paper); finally, specific relationships to represent inter-schema integration-abstractions mappings are introduced. In order to avoid misunderstandings w.r.t. to technical notions from standard ontology Web languages (in particular in the next sections when a translation-based semantics is provided), we adopt the following conventions inspired by the Entity Relationship model: concepts in the CCR formalisms will be called *entities* and subconcept relationships will be called *generalization* relationships.

Formally, the CCR syntax is defined as follows.

Definition 1. ***Concept-to-Concept Relationship (CCR) Alphabet***. *An CCR alphabet $\mathcal{A} = (\Sigma, E, R, gen, IA, i^{inv})$, is a tuple where: Σ is a set of schema names, E is a set of entity names, R is a set of binary relationship names, gen is the generalization relation symbol, IA is a set of names of integration-abstraction relations, $i^{inv} = R \rightarrow R$ associates relation names with names of their inverse relations, and the sets Σ, E, R, gen, IA are pairwise disjoint.*

Definition 2. CCR ***language and*** CCR ***schemata.*** *Given a CCR alphabet $\mathcal{A} = (\Sigma, E, R, gen, IA, i^{inv})$, a CCR language $\mathcal{L}^{CCR}$ based on $\mathcal{A}$ is the set of sentences having the form:*

- *intra-schema $\mathcal{L}^{CCR}$ sentences*
  - $S{:}r(S{:}e, S{:}f)$;
  - $S{:}r^{-}(S{:}e, S{:}f)$;
  - $gen(S{:}e, S{:}f)$;
- *inter-schema $\mathcal{L}^{CCR}$ sentences*
  - $ia*(S{:}e, S'{:}f)$

where $S \in \Sigma$, $r \in R$,$\{e, f\} \in E$, and $ia* \in IA$, and (r^{-} is a short for $i^{inv}(r)$). Given a language $\mathcal{L}^{CCR}$ defined over an alphabet $\mathcal{A}$, an CCR schema S is a set of intra-schema sentences $\Phi \subseteq \mathcal{L}^{CCR}$. Statements having the form $S{:}r(S{:}e, S{:}f)$ and $S{:}r^{-}(S{:}e, S{:}f)$ are called **CCR patterns***; a CCR pattern whose relation is r is called* **CCR** *r-pattern. Given a schema integration-abstraction framework defined on Σ by a*

structuring function f^{Σ}, a set of inter-schema $\mathcal{L}^{CCR}$ sentences Ψ is valid iff for every S and S' such that $ia*(S{:}e, S'{:}f) \in \Psi$ there exist a set of schemata $T \subseteq \Sigma$ such that $S \in T$ and $f^{\Sigma}(S) = S'$.

As an example, the schema CD1 of Figure 2) is conceptually represented in the CCR language by means of the following statements:
$CD1{:}import-export(CD1{:}Subject, CD1{:}Good)$,
$CD1{:}declared\_in(CD1{:}Good, CD1{:}Cus.\_Decl.)$,
$CD1{:}presented\_to(CD1{:}Cus.\_Decl., CD1{:}Cus.\_Agency)$ (in order to avoid long names the following abbreviations are used in the paper: "Cus." for Custom, "Decl." for declaration, "Adv." for adventure). In the following, when the schema that intra-schema $\mathcal{L}^{CCR}$ sentences refer to is clear from the context, or not relevant, the schema reference will be avoided for sake of clearness (the more compact notation $r(e, f)$ will be used to denote CCR patterns).

One of the peculiar characteristics of CCR schemata is the *Multiple Use of Relationship Names* (MURN) in a same schema; as an example, more than one *in* relationships are represented in the CA0 schema of Figure 2. Many conceptual modeling languages, e.g. the ER language, formally assume the Single Use of Relationship Names (SURN). SURN means that a schema such as CA0 of Figure 2 cannot be represented and specific different names for each of the involved relationship need to be introduced (e.g. $in\#1$, $in\#2$, etc.). SURN can be defined more formally as follows.

Definition 3. ***SURN condition and SURN assumption.*** *Given a schema $S = \varphi_1, ..., \varphi_n$ defined over a language $\mathcal{L}^{CCR}$, the SURN property holds for S iff there not exist two CCR patterns $r(e_1, e_2)$ and $r'(f_1, f_2)$ in S such that $r = r'$. We call a SURN-schema a schema for which the SURN property holds. Given a set Σ of CCR schamas, the SURN assumption holds for Σ iff every schema $S \in \Sigma$ is a SURN-schema.*

We call MURN-schemata the CCR schemata for which the SURN property is not required to hold, and we call MURN the relaxation of the SURN assumption for a set of schemata. Observe that SURN-schemata are MURN-schemata, while the converse does not hold.

3.1 Representing MURN CCR schemata with sound semantics

In the following we exploit a DL notation defined for OWL ($\mathcal{SHOIQ}^{\mathcal{D}}$) and RDFS (based on $DL-Lite$), as defined respectively in [1] and [7]; the DL-Lite axioms $\exists R \subseteq C$ and $\exists R^{-} \subseteq C$ (equivalent to $\exists R.\top \subseteq C$ and $\exists R^{-}.\top \subseteq C$ in $\mathcal{SHOIQ}^{\mathcal{D}}$) represent that C is respectively the domain or the range of the DL role R, where R^{-} denotes the inverse role of R.

The representation of MURN schemata is a crucial but often overlooked issue in light-weight ontology modeling. First, consider the abstract nature of the models represented in the schemata. SURN forces a proliferation of relationship names for relationships with a unique intuitive meaning (e.g. three relationship names in_1,in_2, and in_3 would be needed - under SURN - in the CA0 schema of Figure 2). In the context addressed in this paper this point is particularly relevant: besides the amount

of schemata to represent, many generic relationships are used (e.g. has, use, is related to, part of, and so on), and particularly in the more abstract schemata. Second, there are many references to MURN schemata in the literature: SURN is not assumed in the early semantic nets and is often violated even for languages such as ER for which it is supposed to hold (e.g. see examples in [13]). Third, in our past experiences in the design of repositories of ER schemata SURN was not adopted by the designers: the SURN assumption was systematically violated in the CPA repository (e.g. the "related to" relationship is used up to 7/8 times in a same schema, which consists of less than 20 entities).

Representing MURN CCR schemata by means of a light-weight language such as RDFS is not possible. RDFS easily maps to CCR under the assumption that CCR patterns are represented by domain and range restrictions; e.g. $in(Aiport\_Cus., Airport)$ of CA0 in Figure 2 is interpreted as the assertion that $Aiport\_Cus.$ and $Airport$ are respectively domain and range of the relationship in. However, according to the semantics of RDFS, multiple domain and range assumptions have a conjunctive interpretation; which means that in the CA0 schema the domain of in consists of the intersection of all the concepts the arcs labelled as in start from ($Aiport\_Cus.$,$Airport$, and so on). As a result, RDFS semantics does not capture the intended semantics of CCR patterns in MURN schemata.

However, there can be other possible DL-Lite and OWL-DL interpretations of CCR patterns that conflict with specific CCR pattern combinations, as shown in Figure 3. In particular, it is remarkable that participation constraints that can be represented in DL-Lite do not allow for (0,n) cardinality restrictions, which when representing abstract schemata can be assumed as default cardinality restrictions (they impose the lighter possible constraints on the underlying data models).

CCR pattern DL formalization			Conflicting CCR pattern
{∃R ⊑ C, ∃R⁻ ⊑ D}	RDFS DL-Lite	**Role-typing** First elements in a relation R are of type C, second elements in a relation R are of type D	C —R→ D; E —R→ F
C ⊑ ∃R	DL-Lite	**Existential Restriction** All instances of C have a filler for the relation R	C —R (0,n)→ D
C ⊑ ∃R.D	OWL-DL	**Qualified Existential Restriction** All instances of C have a filler for the relation R of type D	
C ⊑ ∀R.D	OWL-DL	**Qualified Universal Restriction** Second elements in a relation in R whose first elements are of type C are of type D	C —R→ D; C —R→ E

Fig. 3: Possible CCR pattern interpretations and conflicting CCR patters

To address the above problems we propose a new semantic interpretation for CCR patterns based on an automatic deterministic translation of CCR light-weight schemata into OWL-DL ontolgies. As usual, $\mathcal{L}^{CCR}$ entities are interpreted as OWL-DL concepts and $\mathcal{L}^{CCR}$ relationship names are interpreted as OWL-DL properties. Our interpreta-

tion is characterized by three main assumptions. A very light interpretation of the first entity in a CCR R-pattern as a concept included in the domain of R, and of the second entity in the CCR R-pattern as a concept included in the range of R (*domain/range inclusion*). A first epistemic closure that states that the domain (range) of a relationships R is the disjunction of all the concepts corresponding to the entities occurring as first (second) elements in any CCR R-pattern (*domain/range global union*). A second epistemic closure that captures the conditional constraint represented in a CCR pattern of the form $r(e, f)$, that is, that when the first element of a tuple in r is of type e, then the second element of the tuple is of type f; of course we need to consider MURN and adequately treat multiple conditional range (domain) restrictions (e.g. when two CCR patterns $r(e, f)$ and $r(e, f')$ are considered); the strategy is analogous to domain/range global union but it is conditional to specific domain/range concepts *(domain/range conditional union)*. Observe that domain/range inclusion and domain/range global union can be represented by defining the domain/range to be equivalent to the union of all the concepts occurring in the domain/range global union specifications (*domain/range global equivalence*).

As an example, consider the relation in and the in-related patterns of schema CA0. The concepts $Aiport\_Cus.$, $Sea\_Cus.$, $Ground\_Cus.$, $Airport$, $Border\_Station$, $Port$ are subconcepts of the in domain (domain inclusion), which in CA0 consists of the union of these concepts (domain global union); the concepts $Airport$, $Border\_Station$, $Port$, and $City$ are subconcepts of the in range (range inclusion), which in CA0 consists of the union of these concepts (range global union); moreover, given a tuple $< x, y >\in in$: if $x \in Aiport\_Cus.$, then $y \in Airport$; if $x \in Sea\_Cus.$, then $y \in Port$; if $x \in Ground\_Cus.$, then $y \in Border\_Station$; analogous conditional interpretations occur for the other in-patterns in the schema; as for domain conditional union, let us focus on the three in-patterns $in(Airport, City)$, $in(Border\_Station, City)$, and $in(Port, City)$: in this case the interpretation is that, given a tuple $< x, y >\in in$, if $y \in City$, then $x \in Airport \sqcap Border\_Station \sqcap Port$.

The formal conceptual semantics for $\mathcal{L}^{CCR}$ can be defined translating $\mathcal{L}^{CCR}$ schemata into OWL-DL ontologies, where entities are represented by OWL concepts, relationships by OWL properties and CCR patterns by restrictions on properties, according to the mappings defined in Table 1. In the table we adopt the following compact notation: $r(e, \{f_1, ..., f_k\})$ represents the set of $\mathcal{L}^{CCR}$ assertions where e occurs as a first element in a relation r, $r(\{e_1, ..., e_h\}, f)$ represents the set of $\mathcal{L}^{CCR}$ assertions where f occurs as second element in a relation r, and $r(\{e_1, ..., e_h\}, \{f_1, ..., f_k\})$ represent the set of all the $\mathcal{L}^{CCR}$ assertions about a relation r where one element of the first set occurs as first element and one element in the second set occurs as second element.

Observe that based on the epistemic closures, this semantic provides an interpretation of schemata that is relative to an epistemic state. If the schemata are changed, the semantics should be computed again. This is consistent with the aim of this paper: we do not propose to design CCR models with the OWL-DL language (we would not be consistent with our assumptions). The semantics proposed is aimed at providing formal translations at the back-end level for front-end CCR models, that is, in a transparent way to the designers. In this paper we claim that our proposals allows for the more freedom

in the design of CCR models without conflicts with possible CCR patterns and without giving up a set-theoretic semantics.

Table 1: Translation from binary $\mathcal{L}^{CCR}$ MURN schemata to $OWL-DL$ ontologies

$\mathcal{L}^{CCR}$	$\mathcal{SHOIQ}^{\mathcal{D}}$ (OWL-DL)	Intuitive Semantics
$e \mapsto C^e$		Concepts
$r \mapsto P^r$		Properties
$r(\{e_1, ..., e_h\}, \{f_1, ..., f_k\}) \mapsto$	$\exists R.\top \equiv C^{e_1} \sqcup ... \sqcup C^{e_h}$, $\exists R^-.\top \equiv C^{f_1} \sqcup ... \sqcup C^{f_k}$	domain global equivalence, range global equivalence
$r(\{e_1, ..., e_k\}, f) \mapsto C^f \sqsubseteq \forall R^-.(C^{e_1} \sqcup ... \sqcup C^{e_k})$, $r(e, \{f_1, ..., f_k\}) \mapsto C^e \sqsubseteq \forall R.(C^{f_1} \sqcup ... \sqcup C^{f_k})$; $gen(e, f) \mapsto C^e \sqsubseteq C^f$		domain conditional union, range conditional union

3.2 Loose integration of CCR schemata

A set of source schemata are integrated-abstracted in order to provide a target schema that accounts for the knowledge represented in the source schemata at a coarse LOD. The integration-abstraction primitive is based on the application of a set of abstraction mechanisms that, given a set of source concepts, provide an abstract target concept that represent the source concepts. The issue addressed in this paper is related to the representation of the relationships (or mappings) that might occur between the source concepts and the target concept that abstracts them.

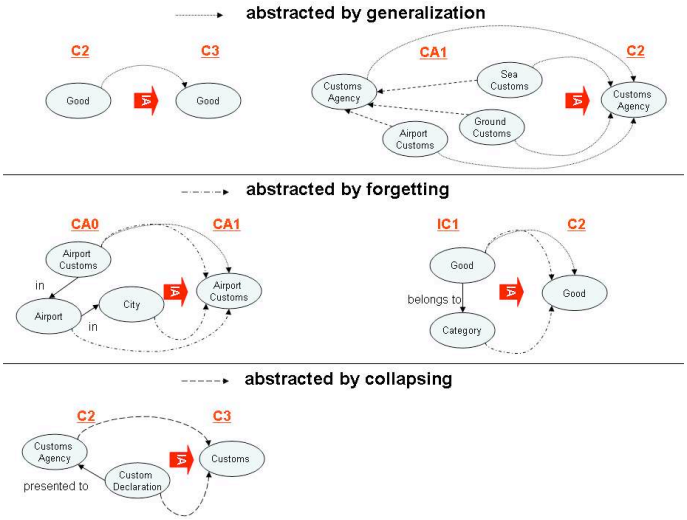


Fig. 4: Examples of the three classes of integration-abstraction relationships

Different kinds of abstraction mechanisms can be applied as to a set of entities, resulting in different kinds of integration-abstraction relations. In this paper we focus on three abstraction mechanisms for CCR schemata, namely *abstraction by generalization*, *abstraction by forgetting* and *abstraction by collapsing*; these abstraction mechanisms are used in the case study described in Figure 2 and have been acknowledged in the literature, althought sometimes under different naming (see Section 4 for details). Figure 4 provides some examples of the application of three mechanisms taken from the case study represented in Figure 2; sets of source concepts are replaced in the target schema by one concept (observe that these sets might trivially consists of one schema, and that the target concept might have the same name of the source schema).

We discuss the semantics of these abstraction mechanisms by introducing three different integration-abstraction relations - and the respective inverse relations - needed to represent the respective abstraction mechanisms. These relations provide the characterization of the IA set in a $\mathcal{L}^{CCR}$ language:

1. $abstract{-}by{-}generalization$ (a-$generalize$ for short), and the inverse $abstracted{-}by-generalization$ (a-$generalized$ for short). This relation represents generalizations between sets of entities of different schemata with standard subsumption semantics; looking at the right-most example in the top-most section of Figure 4), the entities $Cus.\_Agency$, $Sea\_Cus.$, $Airport\_Cus.$, and $Ground\_Cus.$ of CA1 are a-$generalized$ by the entity $Cus.\_Agency$ of C2 (i.e. $a\text{-}generalized(C1{:}sea\_cus., C2{:}cus.\_agency)$, and so on).
2. $abstract-by-forgetting$ (a-$forget$ for short), and the inverse a-$forgot$. It represents abstractions of source entities that are "sunk" in a more abstract target entity, discarding some details in the source representations; looking at the right-most example in the middle section of Figure 4), the entities $Good$ and $Category$ of IC1 are a-$forgot$ in the entity $Good$ of C2 (i.e. $a\text{-}forgot(IC1{:}category, C2{:}good)$ and $a\text{-}forget(IC1{:}good, C2{:}good)$).
3. $abstract-by-collapsing$ (a-$collapse$ for short), and the inverse a-$collapsed$. It represents abstraction mechanisms in which the target concept has a different meaning w.r.t. all the source concepts; looking at the example in the bottom-most section of Figure 4), the entities $Cus.\_Agency$ and $Cus.\_Decl.$ of schema C2 are a-$collapsed$ in the entity $Customs$ of schema C3 (i.e. $a\text{-}collapsed(C2{:}cus.\_agency, C3{:}cus)$ and $a\text{-}collapsed(C2{:}cus\_decl., C3{:}cus)$).

Intuitively, a-$collapse$ and a-$forget$ are quite similar, but a-$forget$ relations are polarized on an entity: there exists one entity in the source schema whose instances can be considered also instances of the abstract entity. This can be modeled by introducing also an abstraction by generalization relation for such an entity: e.g. in Figure 4 the entity $Good$ of C2 a-$generalize$ the entity $Good$ of IC1, which is represented by the sentence $a\text{-}generalize(C2{:}good, IC1{:}good)$. Generalizations cannot be established for a-$collapse$, where for none of the source entities it can be assumed that instances are also instances of the target entity; as an example, the entity $Customs$ in schema C3 represents the general notion of customs as public institutions; custom declarations and custom agencies are not "customs" according to this meaning. For this reason, the intuitive meaning of a-$collapse$ includes "part of"-like aggregation and the grouping relations introduced in [14] (customs as public institution are composed of

other entities, among which custom declarations and custom agencies), and is almost equivalent to unfolding relations as introduced in [15]. Observe that this shift in meaning might occur even when only one concept is $a\text{-}collapsed$ into another concept (e.g. imagine that custom agencies are represented by a concept named Custom).

How do available light-weight semantic Web languages behave w.r.t. the representation of the above mechanisms? First, consider that only abstraction by generalizations can be natively codified as relations, namely subsumption relations ($\sqsubseteq$), between two concepts. These relations can easily be interpreted as subsumptive mappings traditionally used in data integration [2]. If we consider abstraction by forgetting, subsumption relations between the source concept the forgetting mechanism is polarized on can be represented but the information about the other forget source concept is lost. If we consider abstraction by collapsing, none of the source concept can be mapped with a subsumption relation to the target concept. The representation of integration-abstraction relations is not straightforward.

In order to overcome these problems one could introduce specific relations, e.g. $a\text{-}forgot$ and $a\text{-}collapsed$ to be used in more complex axioms; as an example, consider to represent that $Cus.\_Agency$ of C2 is $a\text{-}collapsed$ in $Customs$ of C3. Here different options are available: (A) $Cus.\_Agency$ and $Customs$ are respectively domain and range of $a\text{-}collapsed$; (B) $Cus.\_Agency \sqsubseteq \forall a\text{-}collapsed.Customs$; (C) $Cus.\_Agency \sqsubseteq \exists a\text{-}collapsed.Customs$. The option (A) cannot be adopted because $a\text{-}collapsed$ relations have more than one concept as domain, as clearly represented in Figure 4 (the label "0,n" in the figure refers to intended interpretations of a CCR pattern $r(c, d)$ as [O,n] cardinality constraints between two classes C and D, e.g. in the UML model [14]); observe that the same argument applies to the inverse relations w.r.t. range restrictions. Unfortunately (A) is the only option available assuming RDFS or DL-Lite; hence the negative answer to Q1.

Moreover, assume to represent these multi-layered mappings (*multi-layering*) by means of one ontology that import all the CCR schemata and defines their mappings. The resulting ontology is clearly based on MURN, which means that more complex strategies are needed to represent integration-abstraction relations in the context addressed in this paper. This is another argument against the adoption of the option (A).

The option (B) is safe against the above arguments, but does not capture the strong commitments in the definition of the integration-abstraction relations. If option (C) is applied to $a\text{-}collapse$ and $a\text{-}forget$, we have a case similar to qualified universal restriction depicted in Figure 3. However, this does not occur if we consider their inverse relations $a\text{-}collapsed$ and $a\text{-}forgot$ because a set of source concepts are $a\text{-}collapsed$ and $a\text{-}forgot$ into at most one schema. We therefore propose a solution based on the functional interpretation of the relations $a\text{-}collapsed$ and $a\text{-}forgot$, and the exploitation of inverse property axioms (last two rows of Table 2). Formally, this interpretation is represented in Table 2.

4 Related Works

CCR models largely overlap with simple semantic nets whose nodes represent concepts (and not instances). CCR models are isomorphic to Concept Maps [16] and to RDFS

Table 2: Semantics for $\mathcal{L}^{CCR}$ ia-relations

$\mathcal{L}^{CCR}$		$\mathcal{SHOIQ}^{\mathcal{D}}$ (OWL-DL)
$a\text{-}generalized(S{:}e, S'{:}f)$	$\mapsto$	$C^{S:e} \sqsubseteq C^{S':f}$
$a\text{-}forgot(S{:}e, S'{:}f)$	$\mapsto$	$C^{S:e} \sqsubseteq \forall a\text{-}forgot.C^{S':f}$
$a\text{-}collapsed(S{:}e, S'{:}f)$	$\mapsto$	$C^{S:e} \sqsubseteq \forall a\text{-}collapsed.C^{S':f}$
$a\text{-}generalize(S{:}e, S'{:}f)$	$\mapsto$	$C^{S':f} \sqsubseteq C^{S:e}$
$a\text{-}forget$	$\mapsto$	$a\text{-}forget \equiv a\text{-}forgot^{-}$
$a\text{-}collapse$	$\mapsto$	$a\text{-}collapse \equiv a\text{-}collapsed^{-}$

under the interpretation given in Section 3 (if we assume not to consider property hierarchies, not relevant to the claim of the paper); moreover, tools like Semantic MediaWiki [8] and MoKi [9], which make the user specify global or local domain/range restrictions through quantifier and cardinality-free forms or shortcuts, are based on a front-end design language isomorphic to CCR models. CCR patterns in Semantic MediaWiki are based on RDFS [8], which means that, in theory, only SURN schemata can be represented. The interpretation of CCR patterns in MoKi is not clear from [9]; there are reason to believe that their interpretation is based on qualified existential range restriction (the third top-most interpretation represented in Figure 3)

The work more related to our proposal the translation from concept maps to OWL ontologies proposed in [6]. The proposed transformation covers more complex CCR models than the one covered in this paper (e.g. it considers also instances as part of the maps). They interpret Concept Maps *propositions* (analogous to CCR patterns) as domain/range global union, and also refer to WordNet to disambiguate between instances and concepts. However, they do not introduce any conditional domain/range union semantics, and therefore they do not capture specific conditional dependencies represented in the Concept Map propositions (see Section 3). Finally, the interpretation they provide for the specification of multiple ranges for a same property looks counterintuitive; e.g. the proposition $(Activity, hasType, \{Air\_Adv, Sea\_Adv\})$, is translated into the axiom $Activity \sqsubseteq \exists hasTypeAir\_Adv. \cap \exists hasTypeSea\_Adv.$, from which it can be derived that an activity has always two types.

The approach to schema integration in the large based on integration-abstraction primitive is based on the approach introduced in [11]. However, that approach was based on Entity Relationship schemata, while here we discuss how to exploit the approach in a semantic Web framework. Moreover, the classification of the three abstraction mechanisms, the relations to represent them, and their semantics are new contribution of this paper. This approach to schema integration is very close to traditional techniques for data integration, where the concepts of local schemata are mapped to the concepts of a global schema [2]. At a schema-level, we differ from traditional approach because we do not consider only subsumption-based mappings, which are the mappings that most of the techniques for ontology alignment provide [17], but more in general abstraction-based mappings; moreover we adopt a multi-layered integration approach because of the large amount of schema considered. As argued in the paper, nor RDFS or DL-Lite provide provide specific language constructs to model different kinds of integration-abstraction relationships.

Abstractions in conceptual modeling have been studied to support database design [18], database comprehension and schema summarization [19], formal characterizations of generic relationships [14], and, recently, theories of ontology granularity [15]. Abstraction based on *forgetting* has been applied to Web ontologies [20]. As for conceptual database design, abstraction primitives are exploited to refine or abstract conceptual schemata in top-down and bottom-up database design methodologies [18]. As for database comprehension, several papers address the problem of dominating complexity of large schemata by means of schema clustering techniques (see [21], [13] and references therein). Abstraction are exploited also in [19] to make flat conceptual schemata more comprehensible; the conceptual modeling language used in [19] is Object-Role Modeling (ORM), which is more expressive than ER. All the above mentioned approaches do not explicitly define the abstraction relations between the clusters of entities and their abstract representatives in terms of set-theoretic semantics; instead, the abstraction mechanisms are defined in terms of operations carried out on the schemata.

Generic relationships and their semantics in conceptual models are analyzed in [14]; some of these generic relationships, i.e. aggregation, generalization and grouping can be interpreted as or are related to abstraction relations between concepts. The exploitation of abstraction to enhance comprehension of ontologies and conceptual models has been also proposed in [15]. Three main types of abstractions representing three abstraction mechanisms are introduced: (i) the relation is remodeled as a function; (ii) multiple entities and relations fold into a different type of entity; (iii) semantically less relevant entities and relations are deleted. The primitives used in this paper for ER conceptual schemata overlap with the abstraction types discussed in [14],[15] and [20]. Forgetting in CCR schemata is very close to *deletion* in ontologies as defined in [15].

5 Conclusions

In this paper we consider a context where the representation and integration of semantic Web models (or schemata) is exploited to provide large organizations with an integrated view of the information they manage. Discussing a case study in the eGovernment domain and previous works of colleagues, we assume to adopt at the front-end level light weight graph-based Concept-to-Concept Relationship (CCR) representations. We therefore claim that, in the context of schema representation and integration in the large, light-weight semantic Web languages such as RDFS and DL-Lite (i) cannot be used to provide the semantics of individual CCR models and (ii) are not sufficient to provide appropriate semantics for the definition of the loose mappings needed for model integration. We therefore propose a new interpretation of CCR models semantics; moreover, based on the identification of three abstraction mechanisms exploited in the integration process, we define three main classes of integration-abstraction relations and their semantics.

The approach and the translations defined in the paper allow for the reuse of the methodology and the schemata in the repositories described in [10] in a semantic Web framework. Current research is aimed to develop effective and user friendly graphical interface to browse and edit multi-layered repositories of schemata.

References

1. Staab, S., Studer, R.: Handbook on Ontologies (International Handbooks on Information Systems). SpringerVerlag (2004)
2. Noy, N.F.: Semantic integration: a survey of ontology-based approaches. SIGMOD Rec. **33** (2004) 65–70
3. Baumeister, J., Reutelshoefer, J., Puppe, F.: Engineering on the knowledge formalization continuum. In: SemWiki'09: Proceedings of 4th Semantic Wiki workshop. (2009)
4. Paslaru, E., Simperl, B., Tempich, C., Sure, Y.: Ontocom: A cost estimation model for ontology engineering. In: Proceedings of the 5th International Semantic Web Conference ISWC2006. (2006)
5. Hepp, M.: Possible ontologies: How reality constrains the development of relevant ontologies. IEEE Internet Computing **11** (2007) 90–96
6. Simón, A., Ceccaroni, L., Rosete, A.: Generation of OWL ontologies from concept maps in shallow domains. (2007) 259–267
7. Calvanese, D., Lembo, D., Lenzerini, M., Rosati, R.: DL-Lite: Tractable Description Logics for ontologies. In: In Proc. of AAAI 2005. (2005) 602–607
8. Krötzsch, M., Vrandečić, D., Völkel, M., Haller, H., Studer, R.: Semantic wikipedia. Web Semant. **5** (2007) 251–261
9. Ghidini, C., Kump, B., Lindstaedt, S.N., Mahbub, N., Pammer, V., Rospocher, M., Serafini, L.: Moki: The enterprise modelling wiki. In Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvönen, E., Mizoguchi, R., Oren, E., Sabou, M., Simperl, E.P.B., eds.: ESWC. Volume 5554 of Lecture Notes in Computer Science., Springer (2009) 831–835
10. Batini, C., Barone, D., Garasi, M., Viscusi, G.: Design and use of ER repositories: Methodologies and experiences in egovernment initiatives. In Embley, D.W., Olivé, A., Ram, S., eds.: ER. Volume 4215 of Lecture Notes in Computer Science., Springer (2006) 399–412
11. Batini, C., Battista, G.D., Santucci, G.: Structuring primitives for a dictionary of entity relationship data schemas. IEEE Trans. Softw. Eng. **19** (1993) 344–365
12. Noy, N.F., Musen, M.A.: Specifying ontology views by traversal. In McIlraith, S.A., Plexousakis, D., van Harmelen, F., eds.: International Semantic Web Conference. Volume 3298 of Lecture Notes in Computer Science., Springer (2004) 713–725
13. Tavana, M., Joglekar, P., Redmond, M.A.: An automated entity-relationship clustering algorithm for conceptual database design. Inf. Syst. **32** (2007) 773–792
14. Dahchour, M., Pirotte, A., Zimányi, E.: Generic relationships in information modeling. J. Data Semantics IV **3730** (2005) 1–34
15. Keet, C.M.: Enhancing comprehension of ontologies and conceptual models through abstractions. In: AI*IA '07: Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence on AI*IA 2007, Berlin, Heidelberg, Springer-Verlag (2007) 813–821
16. Coffey, J.W., Hoffman, R.R., Cañas, A.J.: Concept map-based knowledge modeling: perspectives from information and knowledge visualization. Information Visualization **5** (2006) 192–201
17. Euzenat, J., Shvaiko, P.: Ontology matching. Springer-Verlag, Heidelberg (DE) (2007)
18. Batini, C., Ceri, S., Navathe, S.B.: Conceptual database design: an Entity-relationship approach. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA (1992)
19. Campbell, L.J., Halpin, T.A., Proper, H.A.: Conceptual schemas with abstractions making flat conceptual schemas more comprehensible. Data Knowl. Eng. **20** (1996) 39–85
20. Wang, Z., Wang, K., Topor, R.W., Pan, J.Z.: Forgetting concepts in DL-Lite. In Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M., eds.: ESWC. Volume 5021 of Lecture Notes in Computer Science., Springer (2008) 245–257
21. Sousa, P., de Jesus, L.P., Pereira, G., e Abreu, F.B.: Clustering relations into abstract er schemas for database reverse engineering. Sci. Comput. Program. **45** (2002) 137–153