An optimised Semantic Web query language implementation in Prolog

Jan Wielemaker

Human Computer Studies (HCS),
University of Amsterdam,
Kruislaan 419, 1098 VA Amsterdam, The Netherlands,
wielemak@science.uva.nl

Abstract. The Semantic Web is a rapidly growing research area aiming at the exchange of *semantic* information over the World Wide Web. The Semantic Web is built on top of RDF, an XML-based *exchange* language representing a triple-based data model. Higher languages such as the description logic based OWL language family are defined on top of RDF. Making inferences over triple collections is a promising application area for Prolog.
In this article we study query translation and optimization in the context of the SeRQL RDF query language. Queries are translated to Prolog goals, which are optimised by reordering *literals*. We study the domain specific issues of this general problem. Conjunctions are often large, but the danger of poor performance of the optimiser can be avoided by exploiting the nature of the triple store. We discuss the optimisation algorithms as well as the information required from the low level storage engine.

1 Introduction

The Semantic Web [1] initiative provides a common focus for Ontology Engineering and Artificial Intelligence based on a simple uniform triple based data model. Prolog is an obvious candidate language for managing graphs of triples.

Semantic Web languages, such as RDF [2] RDFS and OWL, [4] define which new triples can be deduced from the current triple set (i.e. are *entailed* by the triples under the language). In this paper we study our implementation of the SeRQL [3] query language in Prolog. SeRQL provides a declarative search specification for a sub-graph in the deductive closure under a specified Semantic Web language of an RDF triple set. The specification can be augmented with conditions to match literal text, do numerical comparison, etc.

The original implementation of the SeRQL language is provided by Sesame [3], a Java based client/server system. Sesame realises entailment reasoning by computing the complete deductive closure under the currently activated Semantic Web language and storing this either in memory or in an external database. I.e. Sesame uses pure *forward reasoning*.

We identified several problems using the Sesame implementation. Sesame stores both the explicitely provided triples and the triples that can be derived from them given de semantics of a specified Semantic Web language (e.g. 'RDFS') in one database. This implies that changing the language to (for example) 'OWL-DL' requires deleting the derived triples and computing the deductive closure for the new language. Also, where the full deductive closure for RDFS is still fairly small, it explodes for more expressive languages like OWL. Sesame is sensitive to the order in which path expressions are formulated in the query, which is considered undesirable for a declarative query language. Finally, Sesame is written in Java an we feel much more comfortable using Prolog for manipulating RDF graphs to prototype new inferencing strategies.

To overcome the above mentioned problems we realised a server hosting multiple reasoning engines realised as Prolog modules. Queries can be formulated in the SeRQL language and both queries and results are exchanged through the language independent Sesame HTTP based client/server protocol. We extend the basic storage and query system described in [18] with SeRQL over HTTP and a query optimiser.

Naive translation of a SeRQL query to a Prolog program is straightforward. Being a declarative query language however, authors of SeRQL queries should and do not pay attention to efficient ordering of the path expressions in the query and therefore naive translations often produces inefficient programs. This problem as well as our solution is very similar to what is described by Struyf and Blockeel in [16] for Prolog programs generated by an ILP [11] system. We compare our work in detail with Struyf in Sect. 11.

In Sect. 2 and Sect. 3 we describe the already available software components and introduce RDF. Section 4 to Sect. 9 discuss native translation of SeRQL to Prolog and optimizing the naive translation through reordering of literals.

2 Available components and targets

Sesame[1] and its query language SeRQL is one of the leading implementations of semantic web RDF storage and query systems [9]. Sesame consists of two Java based components. The *server* is a Java *servlet* providing HTTP access to manage the RDF store and run queries on it. The *client* provides a Java API to the HTTP server.

The SWI-Prolog[2] SemWeb package [18] is a library for loading and saving triples using the W3C RDF/XML standard format and making them available for querying through the Prolog predicate **rdf/3**. After several cycles we realised the memory-based triple-store as a foreign language extension to SWI-Prolog. Using foreign language (C) we optimised the data representation and indexing for RDF triples, dealing with upto 40 million triples on 32-bit hardware or virtually unlimited on 64-bit hardware. The SWI-Prolog HTTP

[1] http://www.openrdf.org
[2] http://www.swi-prolog.org

client/server packagehttp://www.swi-prolog.org/packages/http.html provides a multi-threaded [17] HTTP server and client library.

By reimplementing the Sesame architecture in Prolog we make our high performance triple-store available to the Java world. The options are illustrated in Fig. 1. In our project we needed access from Java applications to the Prolog server. Other people are interested in fetching sub-graphs from huge Sesame hosted triple sets stored in an external database to Prolog for further processing.

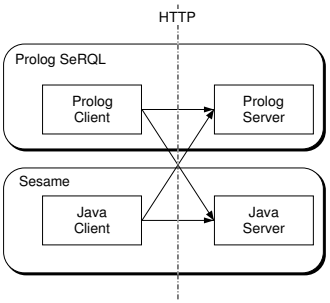


Fig. 1. With two client/server systems sharing the same HTTP API we have created four options for cooperation.

3 RDF graphs and SeRQL queries graphs

In this section we briefly introduce RDF graphs and SeRQL queries. The RDF data model is a set of triples of the format *<Subject Predicate Object>*. The model knows about two data types:[3] *resources* and *literals*. Resources are *Universal Resource Identifiers* (URI), in our toolkit represented by Prolog atoms. Representing resources using atoms exploits the common representation of atoms in Prolog implementations as a unique handle to a string. This representation avoids duplication of the string and allows for efficient equality testing, the only operation defined on resources. Literals are represented by the term **literal**(*atom*), where *atom* represents the textual literal.

A triple informally states *Subject* has an attribute named *Predicate* with value *Object*. Both *Subject* and *Predicate* are resources, *Object* is either a resource or a literal. As a resource appearing as *Object* can also appear as *Subject* or even *Predicate*, a set of triples form a *graph*. A simple RDF graph is shown in Fig. 2.

RDF triples are naturally expressed using the predicate **rdf/3** with the obvious arguments **rdf**(*Subject, Predicate, Object*). Finding a subgraph with certain properties is now easily expressed as a Prolog conjunction, for example

[3] Actually literals can be typed using a subset of the XML Schema primitive type hierarchy

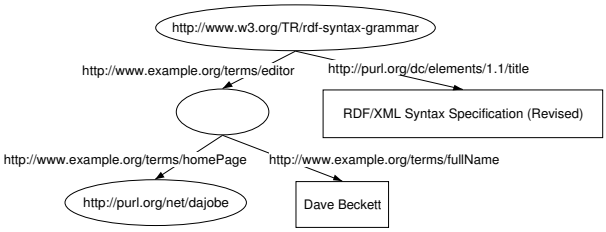


Fig. 2. A simple RDF graph. Ellipses are resources. Rectangles are literal values. Arrows point from *Subject* to *Object* and are labeled with the *Predicate*.

```
reports_by_person(Report, Name) :-
        rdf(Author, 'http://www.example.org/terms/fullName', literal(Name)),
        rdf(Report, 'http://www.example.org/terms/author', Author).
```

SeRQL is a language with a syntax inspired in SQL, useful to represent target subgraphs as a set of edges, possibly augmented with conditions. An example is given in Fig. 3.

4 Compiling SeRQL queries

The SWI-Prolog SeRQL implementation translates a SeRQL query into a Prolog goal, where edges on the target subgraph are represented as calls to **rdf**(*Subject, Predicate, Object*) and the WHERE clause is represented using natural Prolog conjunction and disjunction of predicates provided in the SeRQL runtime support module. The compiler is realised by a DCG parser, followed by a second pass resolving SeRQL namespace declarations and introducing variables. We illustrate this translation using an example from the SeRQL examples.[4] First we present the query in Fig. 3.

```
SELECT Painter, FName
FROM {Painter} <rdf:type>        {<cult:Painter>};
               <cult:first_name> {FName}
WHERE FName like "P*"
USING NAMESPACE
        cult = <!http://www.icom.com/schema.rdf#>
```

Fig. 3. Example SeRQL query asking for all resources of type `cult:Painter` whose name starts with `P`.

Below is the naive translation represented as a Prolog clause and modified for better readability using the variable names from the SeRQL query. To solve

[4] http://www.openrdf.org/sesame/serql/serql-examples.html

the query, this clause is executed in the context of an *entailment module* as illustrated in Fig. 4. An entailment module is a Prolog module providing a pure implementation of the predicate **rdf/3** that can generate as well as test all triples that can be derived from the actual triple store using the Semantic Web language the module defines. This implies the predicate can be called with any instantiation pattern, will bind all arguments and produce all alternatives that follow from the entailment rules on backtracking. If **rdf/3** satisfies these criteria, any naive translation of the SeRQL query is a valid Prolog program to solve the query. Primitive conditions from the WHERE clause are mapped to predicates defined in the SeRQL runtime module which is imported into the entailment module. As the translation of the WHERE clause always follows the translation of the path expression all variables have been instantiated.

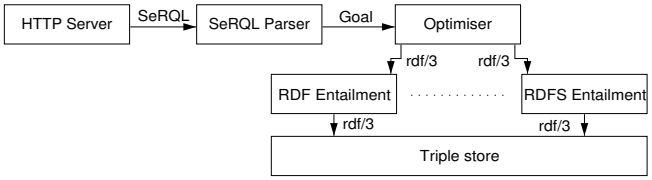


Fig. 4. Architecture, illustrating the role of *entailment modules*. These modules provide a pure implementation of **rdf/3** for the given Semantic Web language.

```
q(row(Painter, FName)) :-
        rdf(Painter,
            'http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
            'http://www.icom.com/schema.rdf#Painter'),
        rdf(Painter,
            'http://www.icom.com/schema.rdf#first_name',
            FName),
        serql_compare(like, FName, 'P*').
```

SeRQL path expressions between square brackets ([...]) are *optional*. They bind variables if they can be matched, but they do not change the core of the matched graph. Such path expressions are translated using the SWI-Prolog *soft-cut* control structure represented by `*->`,[5] for example, the SeRQL statement

```
SELECT Artist, FName
FROM   {Artist} <rdf:type>        {<cult:Artist>};
               [<cult:first_name> {FName}]
USING NAMESPACE
        cult = <!http://www.icom.com/schema.rdf#>
```

is translated into the code below. Note that this prolog code generates all available first names, leaving *FName* unbound if no first name can be found. The final `true` is the translation of the omitted WHERE clause.

[5] Some Prolog dialects (e.g. SICStus) call this construct **if/3**.

```
q(row(Artist, FName)) :-
        rdf(Artist,
            'http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
            'http://www.icom.com/schema.rdf#Artist'),
        (   rdf(Artist, 'http://www.icom.com/schema.rdf#first_name', FName)
        *-> true
        ;   true
        ),
        true.
```

5 The ordering problem

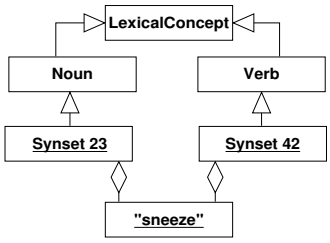
Given the purely logical definition of **rdf/3**, conjunctions of these goals can be placed in any order without influencing the result-set. Literals resulting from the WHERE clause are side-effect free boolean tests that can be executed as soon as the arguments have been instantiated. Note that Gooley [8] distinguishes 4 types of equivalence under optimization: *reflexive*, *set*, *tree* and *inequivalence*. We demand *set* equivalence, returning the same set of results where we do not care about ordering or duplicates.

To study the ordering problem in more detail we will consider the following example query on WordNet [10]. The query looks for words that can be interpreted in at least two different lexical categories.

```
SELECT DISTINCT L
   FROM {S1} <wns:wordForm> {L},
        {S2} <wns:wordForm> {L},
        {S1} <rdf:type> {C1},
        {S2} <rdf:type> {C2},
        {C1} <serql:directSubClassOf> {<wns:LexicalConcept>},
        {C2} <serql:directSubClassOf> {<wns:LexicalConcept>}
   WHERE not C1 = C2
   USING NAMESPACE
         wns = <!http://www.cogsci.princeton.edu/~wn/schema/>
```

WordNet is organised in *synsets*, an abstract entity roughly described by the associated *wordForms*. Synsets are RDFS instances of one of the subclasses of LexicalConcept. We are looking for a wordForm belonging to two synsets of a different subtype of LexicalConcept. Figure 5 illustrates a query result and gives some relevant metrics on WordNet.

To illustrate the need for optimisation as well as to provide material for further discussion we give two translations of this query. Figure 6 shows the direct translation, which requires 3.58 seconds CPU time on an AMD 1600+ processor as well as an alternative which requires 8,305 CPU seconds to execute, a slowdown of 2,320 times. Note that this translation could be the direct translation of another SeRQL query with the same semantics.



WordNet metrics	
Distinct wordForms	123,497
Distinct synsets	99,642
wordForm triples	174,002
Subclasses of LexicalConcept	4

Fig. 5. According to WordNet, the word "sneeze" can be interpreted as a *noun* as well as a *verb*. The tabel to the right gives some metrics of WordNet.

```
s1(L) :-
        rdf(S1, wns:wordForm, L), rdf(S2, wns:wordForm, L),
        rdf(S1, rdf:type, C1), rdf(S2, rdf:type, C2),
        rdf(C1, rdfs:subClassOf, wns:'LexicalConcept'),
        rdf(C2, rdfs:subClassOf, wns:'LexicalConcept'),
        C1 \== C2.

s2(L) :-
        rdf(C1, rdfs:subClassOf, wns:'LexicalConcept'),
        rdf(C2, rdfs:subClassOf, wns:'LexicalConcept'),
        C1 \== C2,
        rdf(S1, rdf:type, C1), rdf(S2, rdf:type, C2),
        rdf(S1, wns:wordForm, L), rdf(S2, wns:wordForm, L).
```

Fig. 6. Two translations for our query on WordNet. The first executes in 3.58 seconds, the second in 8,305.

Before we start discussing the alternatives for optimising the execution we explain *why* the execution times of these equivalent programs differs. Suppose we have a conjunction of completely independent literals A, B, C, where independent means no variables are shared between the members of the conjunction. If $b()$ denotes the number of solutions for a literal, the total solution space is $b(A) \times b(B) \times b(C)$ and therefore independent from the order. If we take the number of visited states rather than the solution space as a measure the formula becomes

$$b(A) + b(A) \times b(B) + b(A) \times b(B) \times b(C)$$

This measure is proportional to the number of *logical inferences* executed by Prolog and a good measure for the expected execution time [6]. It suggests to place literals with the smallest number of alternatives first, but as the last component is normally dominant the difference is not large and certainly cannot explain the difference between the two translations shown in Fig. 6. In fact the second is ordered on the branching factor *without considering dependencies*.

To understand this difference we must look at the dependencies, expressed by shared variables. Executing an **rdf/3** literal causes all its arguments to be grounded, restricting the number of alternatives for **rdf/3** literals sharing the grounded variables. What is really important is how much the set of alternatives of a literal is reduced by executing another literal before it. The order of **s1/1** in Fig. 6 executes the most unbound literal first (174,002 solutions), but wins because after the execution of this literal not much further branching is left.

6 Estimating the complexity

The first step towards optimising is having an estimate of the complexity of a particular translation. We use the number of visited nodes in the search-tree as an estimate for the execution time, ignoring the (small) differences in time required to execute the different **rdf/3** literals. Our estimate is based on two pieces of information extracted from the low-level database we have realised in the C language.

Estimated number of solutions for an rdf/3 call For each **rdf/3** goal for which zero or more of the arguments have a known value and the remaining arguments are known to be unbound we can easily estimate the complexity. If no arguments are known this estimate is the total number of triples in the database, a number that is easily incrementally maintained by the database manipulation routines. If all arguments are known the literal is a boolean test, whose solution set we estimate as 0.5 (see *Boolean tests* below). In all other cases we compute the indexing and return the length of the hash-chain for the computed index. Assuming a well distributed hash-function this is a reasonable estimate for the number of solutions the goal will provide, while the information can be maintained incrementally by the database primitives.

Estimating the branching factor of predicates Execution of literals binds variables, but unfortunately we do not know with what value(s). Observing queries however we see that for many literals we do know the predicate (2nd argument of **rdf/3**) at query time, leaving two interesting cases: subject bound to unknown value and object unbound and the other way around. We deal with those by defining *subject branch factor* (sbf) resp. *object branch factor* (obf):

$$sbf(P) = \frac{triples(P)}{distinctSubjects(P)}$$

This figure is not cheaply maintained on incremental basis and therefore computed. The result is cached with the predicate and only recomputed if the number of triples on the predicate has changed considerably.

Boolean tests Boolean tests resulting from the WHERE clause cannot cause branching. They can succeed or fail and their branching factor is estimated as 0.5, giving preference to locations early in the conjunction. This number may be wrong but, as we explained in Sect. 5, reordering of independent

members of the conjunction only has marginal impact on the execution time of the query. If not all arguments to the test are sufficiently instantiated computation of the branching factor fails, causing the conjunction permutation generator to generate a new alternative.

The total complexity of a conjunction is now easily expressed as the summed sizes of the search spaces after executing $1, 2, \ldots n$ steps of the conjunction. The branching factor for each step is deduced using symbolic execution of the conjunction, replacing each variable in a literal with a Skolem instance. Skolem instantiation is performed using SWI-Prolog *attributed variables* [5].

7 Optimising the conjunction

With a good and quick to compute metric for the complexity of a particular order, the optimisation problem is reduced to a generate-and-test problem. A conjunction of N members can be ordered in $N!$ different ways. As we have seen actual examples of N nearing 40, naive permutation is not an option. We do not have to search the entire space however as the order of sub-conjunctions that do not share any variables can be established independently, after which they can be ordered on the estimated number of solutions.

Initially, for most conjunctions all literals are related. After having executed a few literals, the grounded variables often break the remaining literals in multiple independent groups that can be optimised separately. The algorithm is show in Fig. 7.

```
order(conj)
{ make_subgraphs(conj, subconjs);
  if ( count(subconjs) > 1 )
  { maplist(order, subconjs, ordered_subs);
    sort_by_complexity(ordered_subs, sorted);

    return join_subgraphs(sorted);
  } else
  { first = select(conj, rest);         (*)
    skolem_bind(first);
    make_subgraphs(rest, subconjs);
    maplist(order, subconjs, ordered_subs);
    sort_by_complexity(ordered_subs, sorted);

    return first + join_subgraphs(sorted);
  }
}
```

Fig. 7. Generating permutations of a conjunction. Note that select(), marked (*), is non-deterministic.

Combining this generator with the complexity estimate of Sect. 6 and selecting the best completes the order selection process. As the permutation algorithm only returns results from reordering independent subgraphs and it selects the best one by sorting the independent subgraphs on their branching, the returned order is guaranteed to be optimal if the complexity estimate is perfect. In other words, the maximum performance difference between the optimal order and the computed order is the error of our estimation function.

8 Optional path expressions and control structures

As explained in Sect. 4, SeRQL optional path expressions are compiled into (Goal `*->` true ; true), where *Goal* is the result of compiling the path expression. We must handle *Goal* as well as other goals appearing in Prolog control structures resulting from compiling the WHERE clause as a unit. If such units are conjunctions they are subject to recursive invocation of our ordering algorithm. Please do note that the ordering and complexity of a conjunction depends on the variables that are already instantiated when the conjunction is entered. Conjunctions in control structures must therefore be ordered and have their complexity determined as part of estimating the complexity of the outer conjunction, as illustrated by the following simplified Prolog code fragment:

```
complexity((Goal0 *-> true ; true),
           (Goal *-> true ; true), Complexity) :-
        optimise(Goal0, Goal),
        complexity(Goal, Complexity).
```

Optional path expressions do not change the result set of the obligatory part of the query. It can only produce more variable bindings. Therefore we can simplify the optimisation process of a conjunction by first splitting it into its obligatory and optional part and then optimise the obligatory part followed by the optional part:

```
optimise(Goal0, Goal) :-
        split_optional(Goal0, Obligatory0, Optional0),
        optimise(Obligatory0, Obligatory),
        skolem_bind(Obligatory),
        optimise(Optional0, Optional),
        Goal = (Obligatory, Optional).
```

9 Solving independent path expressions

As we have seen in Sect. 7, the number of distinctive permutations is much smaller than the number of possible permutations of a goal due to the fact that after executing a few literals the remainder of the query breaks down into independent subgraphs. Independent subgraphs can be solved independently and the total result is simply the Cartesian product of all partial results. This approach has several advantages:

- The complexity of solving two independent goals A and B separately is $b(A) + b(B)$ rather than $b(A) + b(A) \times b(B)$.
- If any of the independent goals has no solutions we can abort the whole query and report it has no solutions.
- The subgoals can be solved in parallel.
- The result-set can be expressed as the Cartesian product of partial results, requiring much less communication between server and client.
- It eliminates the need for the 'sort_by_complexity' step in Fig. 7.

This optimisation can be performed after the reordering. It simply does symbolic evaluation and Skolem instantiation of the conjunction statement-by-statement and splits the remainder into subgraphs. If conjunction is represented as a list, the simplified Prolog code fragment below suffices.

```
cartesian(Conjunction, Carhesian) :-
        append(Before, After, Conjunction),
        skolem_bind(Before),
        make_subgraphs(After, SubGraphs),
        SubGraphs = [_,_|_], !,         % demand at least two
        append(Before, serql_carthesian(SubGraphs), Carhesian).
cartesian(Conjunction, Conjunction).
```

This optimisation can be performed recursively on the created independent subgraphs as well as on conjunctions inside control structures.

10 Results

The total code size of the server is approximately 6,700 lines (including comments, but excluding the 25-line GPL file headers). Major categories are show in Tab. 1. We think it is not realistic to compare this to the 86,000 lines of Java code spread over 439 files that make up Sesame. Although both systems share considerable functionality, they differ too much in functionality and what parts are reused from the respective libraries to make a detailed comparison feasible.

Category	**lines**
HTTP server actions	2,521
Entailment modules (3)	281
Result I/O (HTML, RDF/XML, Turtle)	1,307
SeRQL runtime library	192
SeRQL parser and naive compiler	874
Optimiser	878
Miscellaneous	647

Table 1. Size of the various components, counted in lines. RDF/XML I/O is only a wrapper around the SWI-Prolog RDF library.

We have evaluated our optimiser on two domains, the already mentioned WordNet and an RDF database about cultural relations in Spain with real-life queries on this database. Measurements have been executed on a dual AMD 2600+ machine running SuSE Linux and SWI-Prolog 5.5.15.

We have tested our optimiser on artificial as well as real-life SeRQL queries. In all observed cases the optimisation time is only a modest fraction of the execution time of the optimal order as well as generally shorter than the time required to parse the query.

First we study the example of Fig. 6. Our optimiser converts either translation into the goal shown in Fig. 8. The code for **s1/1** was handcrafted by us and can be considered an educated guess for best performance. The result of the optimiser came as a surprise, but actual testing proved the code of Fig. 8 is 1.7 times faster than the code of **s1/1** and indeed the fastest possible. The optimiser requires only 90ms, or just 4.3% of the execution time for the optimal solution.

```
q(L) :-
        rdf(S1, rdf:type, C1),
        rdf(S1, wns:wordForm, L),
        rdf(C2, rdfs:subClassOf, wns:'LexicalConcept'),
        rdf(S2, rdf:type, C2),
        rdf(S2, wns:wordForm, L),
        C1 \== C2,
        rdf(C1, rdfs:subClassOf, wns:LexicalConcept))
```

Fig. 8. Optimised WordNet query

The second test-set consisted of three queries on a database of 97,431 triples coming from a real project carried out at Isoco[6] These queries were selected because Sesame [3] could not answer them (2 out of 3) or performed poorly. Later examination revealed these queries consisted of multiple largely independent sub-queries, turning the result in a huge Cartesian product. Splitting them into multiple queries turned them into manageable queries for Sesame. Exploiting the analysis of independent path expressions described in Sect. 9, our server does not need this rewrite. The results are shown in Tab. 2. We could only verify the result of the 2nd query against Sesame, which returns the same 3,826 solutions in 132.72 seconds.

11 Related Work

Using logic for Semantic Web processing has been explored by various research groups. See for example [12] which exploits *Denotational Semantics* to provide a structured mapping from language to semantics. Most of these approaches

[6] www.isoco.com

Id	Edges	time (ms) optimise	complexity initial	complexity final	speedup	time (s) total	solutions
1	38	10ms	1.4e16	1.4e10	1e6	2.48s	79,778,496
2	30	10ms	2e13	1.3e5	1.7e8	0.51s	3,826
3	29	10ms	1.4315	5.1e7	2.7e7	11.7s	266,251,076

Table 2. Results on complex queries. The engine has been modified slightly to return the Cartesian product as a description instead of expanding it as the expansion does not fit in memory.

concentrate on correctness, while we concentrate on engineering issues and performance.

Much work has been done on optimising Prolog queries as well as database joins by reordering. We specifically refer to the work of Struyf and Blockeel [16] because it is recent and both the problem and solution are closely related. They describe the generation of programs through ILP [11]. The ILP system itself does not consider ordering for optimal execution performance, which is similar to compiling declarative SeRQL statements not producing optimal programs. In ILP, the generated program must be used to test a large number of positive and negative examples. Optimising the program before running is often worthwhile.

The described ILP problem differs in some places. First of all, for ILP one only has to prove that a certain program, given a certain input succeeds or fails, i.e. goals are ground. This implies they can use the cut to separate independent parts of the conjunction (section 4.2 of [16]). As we have non-ground goals and are interested in all distinct results we cannot use cuts but instead use the Cartesian product approach described in Sect. 9. Second, Struyf and Blockeel claim complexity of generate-and-test (order $N!$) is not a problem with the observed conjunctions with a maximum length of 6. We have seen conjunctions with 40 literals. We introduce breaking the conjunctions dynamically in independent parts (Sect. 7) can deal with this issue. Finally, the uniform nature of our data gave us the opportunity to build the required estimates for non-determinism into the low-level data structures and maintain them at low cost (Sect. 6).

12 Discussion

Sofar, we have been using Prolog in the Semantic Web domain for reasoning for annotation [13]. This reasoning was not based on formal Semantic Web languages, but using ad-hoc defined schemas. With our SeRQL implementation we have proven that we can deal completely and efficiently with RDFS. We have proven that SWI-Prolog, supporting threading, attributed variables and equipped with extensive libraries for graphics, XML, RDF triple store and HTTP is a suitable tool for building a variety of Semantic Web applications, covering both interactive and network server applications.

As the Semantic Web evolves with more powerful formal languages such as OWL and SWRL[7], it becomes unlikely we can compile these easily to efficient Prolog programs. TRIPLE [15] is an example of an F-logic based RDF query language realised in XSB Prolog [7]. We believe extensions to Prolog that facilitate more declarative behaviour will prove necessary to deal with the Semantic Web. Both XSB's tabling and constraint logic programming, notably CHR [14] are promising extensions.

13 Conclusions

We have employed Prolog for storing and querying Semantic Web data. In [18] we have demonstrated the performance and scalability of the storage module for use in Prolog. In this paper we have demonstrated the feasibility realising an efficient implementation of the declarative SeRQL RDF query language in Prolog. The resulting system can easily be expanded with new entailment reasoners and can be accessed from both Prolog and Java through the common HTTP interface.

The provided algorithm for optimising the matching process of SeRQL queries reaches optimal results if the complexity estimate is perfect. The worse case complexity of ordering a conjunction is poor, but for tested artificial and real-life queries the optimisation time is shorter than the time needed to execute the optimised query. For trivial queries this is not the case, but here the response time is dictated by the HTTP protocol overhead and parsing the SeRQL query.

The SeRQL server is available under the SWI-Prolog LGPL/GPL license from http://www.swi-prolog.org/packages/SeRQL.

Acknowledgements

We would like to thank Oscar Corcho for providing real-life data and queries. This research has been carried out as part of the HOPS project[8], IST-2002-507967. Jeen Broekstra provided useful explanations on SeRQL.

References

1. Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic Web. *Scientific American*, 284(5):34–43, May 2001.
2. D. Brickley and R. V. Guha (Eds). Resource description framework (RDF) schema specification 1.0. W3C Recommendation, March 2000. http://www.w3.org/TR/2000/CR-rdf-schema-20000327/.
3. Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In I. Horrocks and J. Hendler, editors, *Proceedings of the First Internation Semantic Web Conference*, number 2342 in Lecture Notes in Computer Science, pages 54–68. Springer Verlag, July 2002.

[7] http://www.daml.org/2003/11/swrl
[8] http://www.hops-fp6.org

4. Mike Dean, Guus Schreiber, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL web ontology language reference. Working draft, W3C, March 2003.
5. Bart Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium, oct 2002. URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW350.abs.html.
6. Carlos Escalante. A simple model of prolog's performance: extensional predicates. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 1119–1132. IBM Press, 1993.
7. Juliana Freire, David S. Warren, Konstantinos Sagonas, Prasad Rao, and Terrance Swift. XSB: A system for efficiently computing well-founded semantics. In *Proceedings of LPNMR 97*, pages 430–440, Berlin, Germany, jan 1997. Springer Verlag. LNCS 1265.
8. Markian M. Googley and Benjamin W. WAH. Efficient reordering of PROLOG programs. *IEEE Transactions on Knowledge and Data Engineering*, pages 470–482, 1989.
9. Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A comparison of rdf query languages. In *Proceedings of the Third International Semantic Web Conference, Hi roshima, Japan, 2004.*, NOV 2004.
10. G. Miller. WordNet: A lexical database for english. *Comm. ACM*, 38(11), November 1995.
11. S. Muggleton and L. De Raedt. Inductive Logic Programming: Theory and Method. *Journal of Logic Programming*, 19-20:629–679, 1994.
12. Kunal Patel and Gopal Gupta. Semantic processing of the semantic web. *Lecture Notes in Computer Science*, 2870:80–95, January 2003.
13. Guus Schreiber, Barbara Dubbeldam, Jan Wielemaker, and Bob Wielinga. Ontology-based photo annotation. *IEEE Intelligent Systems*, may/june 2001.
14. Tom Schrijvers and Bart Demoen. The K.U. Leuven CHR system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 430–440, 2004. ISSN 0939-5091.
15. Michael Sintek and Stefan Decker. *TRIPLE* — A query, inference, and transformation language for the Semantic Web. *Lecture Notes in Computer Science*, 2342:364–, 2002.
16. J. Struyf and H. Blockeel. Query optimization in inductive logic programming by reordering literals. In T. Horváth and A. Yamamoto, editors, *Proceedings of the 13th International Conference on Inductive Logic Programming*, volume 2835 of *Lecture Notes in Artificial Intelligence*, pages 329–346. Springer-Verlag, 2003.
17. Jan Wielemaker. Native preemptive threads in SWI-Prolog. In Catuscia Palamidessi, editor, *Practical Aspects of Declarative Languages*, pages 331–345, Berlin, Germany, december 2003. Springer Verlag. LNCS 2916.
18. Jan Wielemaker, Guus Schreiber, and Bob Wielinga. Prolog-based infrastructure for RDF: performance and scalability. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *The Semantic Web - Proceedings ISWC'03, Sanibel Island, Florida*, pages 644–658, Berlin, Germany, october 2003. Springer Verlag. LNCS 2870.