UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Logic programming for knowledge-intensive interactive applications

Wielemaker, J.

Publication date
2009

Link to publication

Citation for published version (APA):
Wielemaker, J. (2009). *Logic programming for knowledge-intensive interactive applications*. [Thesis, fully internal, Universiteit van Amsterdam].

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:07 Jun 2026

Chapter 4

An optimised Semantic Web query language implementation in Prolog

About this chapter This chapter was published at the ICLP-05 (Wielemaker 2005). It continues on research question 1 on knowledge representation, providing an answer to question 1b on efficient matching of RDF graph expressions. From section 4.6 onwards the paper has been updated to reflect the current state of the optimiser, enhance the presentation and create links to the remainder of this thesis in the conclusions.

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 RDFS and the 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 optimisation 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.

4.1 Introduction

The Semantic Web (Berners-Lee et al. 2001) 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 (Brickley and Guha 2000), RDFS and OWL, (Dean et al. 2004) 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 (Broekstra et al. 2002) 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 (Broekstra et al. 2002), 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 (*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 we want the reasoning in Prolog because we assume Prolog is a more suitable language for expressing application specific rules.

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 Wielemaker, Schreiber, and Wielinga (2003b) 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 not have to pay attention to efficient ordering of the path expressions in the query and therefore the query compiler has to derive the optimal order of joins (often called *query planning*). This problem as well as our solution is similar to what is described by Struyf and Blockeel (2003) for Prolog programs generated by an ILP (Muggleton and Raedt 1994) system. We compare our work in detail with Struyf in section 4.11.

In section 4.2 and section 4.3 we describe the already available software components and introduce RDF. Section 4.4 to section 4.9 discuss naive translation of SeRQL to Prolog and optimising the naive translation through reordering of literals.

4.2 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 (Haase et al. 2004). 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 SemWeb package (chapter 3, Wielemaker et al. 2003b) 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 iterations (see section 3.3.3) 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 about 20 million triples on 32-bit hardware or virtually unlimited on 64-bit hardware. The SWI-Prolog HTTP client/server library (chapter 7, Wielemaker et al. 2008) provides a multi-threaded (chapter 6, Wielemaker 2003a) HTTP server and client library.

By reimplementing the Sesame client/server architecture in Prolog we make our high performance triple store available to the Java world. Possible application scenarios are illustrated in figure 4.1. In our project we needed access from Java applications to the Prolog server. Other people are interested in fetching graphs from huge Sesame hosted triple sets stored in an external database to Prolog for further processing.

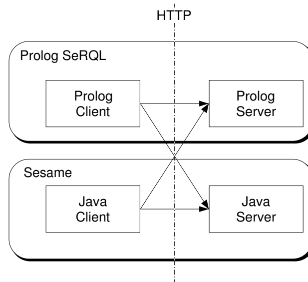


Figure 4.1: With two client/server systems sharing the same HTTP API we have created four scenarios for cooperation: Java or Prolog client connected to Java or Prolog server.

4.3 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

[1] `http://www.openrdf.org`

data types:[2] *resources* and *literals*. Resources are *Internationalised Resource Identifiers* (IRI, rfc3987[3]), 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`(*Value*), where *Value* is one of `type`(IRI, *Text*), `lang`(*LangID, Text*) or plain *Text*, and *Text* is the canonical textual value of the literal expressed as an atom (chapter 3, Wielemaker et al. 2003b).

A triple informally states that *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 *Predicate*, a set of triples forms a *graph*. A simple RDF graph is shown in figure 4.2 (from Beckett and McBride 2004).

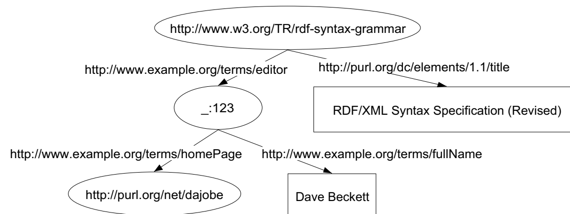


Figure 4.2: A simple RDF graph. Ellipses are resources. Rectangles are literal values. Arrows point from *Subject* to *Object* and are labelled with the *Predicate*.

RDF triples are naturally expressed using the Prolog predicate `rdf`(*Subject, Predicate, Object*). Finding a subgraph with certain properties is now expressed as a Prolog conjunction. The example below finds the home page for a named person in the graph of figure 4.2.

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

SeRQL is a language with a syntax inspired by SQL, useful to represent target subgraphs as a set of edges, possibly augmented with conditions. An example is given in figure 4.3.

4.4 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*)

[2]Actually literals can be typed using a subset of the XML Schema primitive type hierarchy, but this refinement is irrelevant to the discussion in this paper.

[3]`http://www.faqs.org/rfcs/rfc3987.html`

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 documentation,[4] shown in figure 4.3.

```
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#>
```

Figure 4.3: Example SeRQL query asking for all resources of type cult:Painter whose name starts with the capital P.

Figure 4.5 shows the naive translation represented as a Prolog clause and modified for better readability using the variable names from the SeRQL query. To solve the query, this clause is executed in the context of an *entailment module* as illustrated in figure 4.4. An entailment module is a Prolog module that provides 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 that 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. The call to serql_compare/3 in figure 4.5 is an example of calling a SeRQL runtime predicate that implements the like operator.

Optional path expressions SeRQL path expressions between square brackets ([...]) are *optional*. They bind variables if they can be matched, but they do not change the graph matched by the non-optional part of the expression. Optional path expressions are translated using the SWI-Prolog *soft-cut* control structure represented by *->.[5] Figure 4.6 shows a SeRQL query (top) that finds instances of class *cult:Painter* and enriches the result with the painter's first name(s) if known. The bottom of this figure shows the translation into Prolog,

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

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

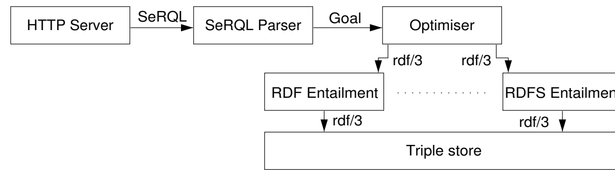


Figure 4.4: Architecture, illustrating the role of *entailment modules*. These modules provide a pure implementation of `rdf/3` for the given Semantic Web language.

```
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*').
```

Figure 4.5: Naive translation of the query of figure 4.3. The row-term in the head describes the bindings of a single result row of the SELECT query.

which leaves *FName* unbound if no first name is known and enumerates all known first names otherwise.

4.5 The ordering problem

Given the purely logical definition of `rdf/3`, conjunctions of RDF goals can be placed in any order without changing the result if we consider two results equivalent if they represent the same *set* of solutions (set-equivalence, Googley and WAH 1989). Literals that result from the WHERE clause are side-effect free boolean tests and can be executed as soon as their arguments have been instantiated.

To study the ordering problem in more detail we will consider the example query in figure 4.7 on WordNet (Miller 1995). The query finds words that can be interpreted in at least two different lexical categories. 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 4.8 illustrates a query result and gives some relevant metrics on WordNet. The pseudo property `serql:directSubClassOf`

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

```
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)
        *-> true
        ;   true
        ).
```

Figure 4.6: Compilation of a SeRQL optional path expression into the SWI-Prolog soft-cut control structure.

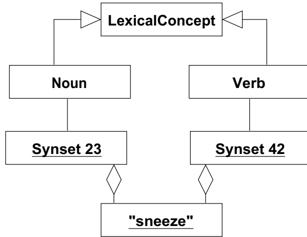
is defined by SeRQL as a non-transitive version of `rdfs:subClassOf`.

```
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/>
```

Figure 4.7: Example of a SeRQL query 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 4.9 shows the direct translation (s1), which requires 3.58 seconds CPU time on an AMD 1600+ processor as well as an alternative ordering (s2) 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.

Before we start discussing the alternatives for optimising the execution we explain *why*



WordNet metrics (version 1.6)	
Distinct wordForms	123,497
Distinct synsets	99,642
wordForm triples	174,002
Subclasses of LexicalConcept	4

Figure 4.8: 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).
```

Figure 4.9: Two translations for our query on WordNet. The first executes in 3.58 seconds, the second in 8,305.

the execution times of these equivalent programs differ. 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 of the order. If we take the number of calls+redos (Prolog *logical inferences*) rather than the solution space as a measure the formula becomes

$$\boxed{b(A)} + \boxed{b(A) \times b(B)} + \boxed{b(A) \times b(B) \times b(C)}$$

Logical inferences are a good measure for the expected execution time (Escalante 1993). It suggests to place literals with the smallest number of alternatives first, but as the last

component of the above formula is dominant the difference is not large and certainly cannot explain the difference between the two translations shown in figure 4.9. In fact the second is ordered on the branching factor *without considering dependencies* and as we will see below, dependencies are key to the problem.

Executing an `rdf/3` literal causes all its arguments to be grounded, reducing the number of solutions for `rdf/3` literals sharing the grounded variables. What is really important is how much the set of solutions of a literal is reduced by executing another literal before it. The order of `s1/1` in figure 4.9 executes the most unbound literal first (174,002 solutions), but wins because after the execution of this literal not much further branching is left.

4.6 Estimating the complexity

The first step towards optimising is having an estimate of the complexity of a particular translation. We use the number of logical inferences as an estimate for the execution time, ignoring the (small) differences in time required to execute the different `rdf/3` literals. Crucial to this is to estimate the number of solutions for an `rdf/3` literal. Our estimate is based on information extracted from the low-level database we have realised in the C-language. For this estimate we must consider the *mode* (instantiation pattern) and, if an argument is instantiated, we must distinguish the case where it is bound to a known value by the query and the case where is bound to an unknown value by a preceeding `rdf/3` literal. Below we enumerate the possible modes, where we use '-' for unbound, '+' for bound to a known value, '*' for bound to an unknown value and '@' for bound (known or unknown).

rdf(@,@,@) If all arguments are bound, the call will not create a choicepoint and can succeed or fail. We estimate the the number of solutions at 0.5.

rdf(-,-,-) If no argument is bound, we have a good estimate provided by the number of triples that is maintained by the database. The actual number can be smaller due to duplicates, but this is generally a small proportion.

rdf(-,+,-) Bound to a known predicate we have a good estimate in the number of triples per predicate that is maintained by the database too.

rdf(*,+,-) This is a common case where the subject is bound, but to an unknown value. Here, we are interested in the *average* number of objects associated to a subject with the given predicate. This is the total number of triples for this predicate divided by the total number of *distinct* subject values, which we will call the *subject branching factor* or *sbf*:

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

Where the total number of triples is known, the number of distinct subjects must be computed. The *sbf* is determined on the first request. The value is stored and only recomputed if the number of triples has significantly changed since it was determined.

rdf(-,+,*) The *object branching factor* or *obf* definition is completely analogous to the above.

rdf(*,-,-) As the database keeps track of the total number of distinct subjects, this can be computed much like the *sbf* with the following formula: $\frac{triples}{subjects}$

Otherwise If none of the above holds we use information from the hash-based index. With known values for some of the arguments we locate the hash chain that would be used for this query and assume that the length is a good estimate for the total number of solutions. This assumes a well distributed hash function.[6] The length of the hash chains is maintained incrementally by the database primitives.

The above handles '*' as '-' for some (rare) patterns, which can cause a much too high estimate. We currently ignore this issue.

Boolean tests resulting from the WHERE clause cannot cause branching. They can succeed or fail and their branching factor is, as ground `rdf/3` calls, estimated as 0.5 which gives preference to locations early in the conjunction. This number may be wrong but, as we explained in section 4.5, reordering of independent members of the conjunction only has marginal impact on the execution time of the query. If not all arguments to a test are sufficiently instantiated computation of the branching factor fails, causing the conjunction permutation generator to generate a new order.

The total complexity of a conjunction is now expressed as the summed sizes of the search spaces after executing $1, 2, \ldots n$ steps of the conjunction (see formula in section 4.5). 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* (Demoen 2002).

4.7 Optimising the conjunction

With a precise and quick method to compute 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. However, we do not have to search the entire space 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.

[6] We use MurmurHash from `http://murmurhash.googlepages.com/`

Initially, for most conjunctions in a query all literals are related through shared variables.[7] As execution of the conjunction progresses, more and more variables are bound. This may cause the remainder of the conjunction to break into multiple independent sub-conjunctions. For example, the query below is initially fully related over *Painter*. After executing one of the literals, *Painter* is bound and the two remaining literals become independent. Independent sub-conjunctions can be optimised separately.

```
q(row(Painter, FirstName, Painting)) :-
        rdf(Painter, rdf:type, cult:'Painter'),
        rdf(Painter, cult:first_name, Name),
        rdf(Painter, cult:creator_of, Painting).
```

The algorithm in figure 4.10 generates on backtracking alternative orderings with an estimate for their time complexity and number of solutions. The predicate `estimate_complexity/3` estimates the number of solutions as described above, while the time estimate is 1 (i.e., our unit of time is the time an `rdf/3` call needs to generate or test a triple). The predicate `combine/3` concatenates the ordered sub-conjunctions and computes the combined complexity.

Using the generator above, we can enumerate all permutations and select the fastest one. 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. We have seen in section 4.6 that the error margin varies, depending on the types of `rdf/3` calls. We do not have access to a sufficiently large set of real-world SeRQL queries to assess the accuracy in any detail.

4.8 Optional path expressions and control structures

As explained in section 4.4, SeRQL optional path expressions are compiled into (Goal `*->` true ; true), where *Goal* is the result of compiling the path expression. We preserve control structures and apply our ordering algorithm to conjunctions that are embedded in the control structures.

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 as shown below. The predicate `combine/2` is the same as from figure 4.10. We must do the Skolem binding of the obligatory part because it affects the result when ordering conjunctions in the optional part of the expression.

[7] This is not necessarily the case, but a query that consists of independent sub-queries is answered by the Cartesian product of the answers for each of the sub-queries (see section 4.9). Processing the results of each independent sub-query is more efficient and generally easier.

```
%%      order(+Conjunction, -Result) is nondet.
%
%       @param Result   o(Time, Solutions, Order)

order([One], o(T,N,[One])) :- !,
        estimate_complexity(One, T, N),
order(Conj, Result) :-
        make_subgraphs(Conj, SubConjs),
        (   SubConjs = [_,_|_]
        ->  maplist(order, SubConjs, OSubs),
            sort(OSubs, OList),
            combine(OList, Result)
        ;   select(First, Conj, Rest),
            skolem_bind(First),
            order(Rest, o(T0,N0,O0)),
            estimate_complexity(First, T1, N1),
            T is T1+N1*T0,
            N is N0*N1,
            Result = o(T,N,[First|O0])
        ).
```

Figure 4.10: Order a conjunction. Alternative orderings are generated due to the non-deterministic `select/3`

```
order(Goal, Result) :-
        split_optional(Goal, Obligatory0, Optional0),
        order(Obligatory0, Obligatory),
        skolem_bind(Obligatory),
        order(Optional0, Optional),
        combine([Optional, Optional], Result.
```

4.9 Solving independent path expressions

As we have seen in section 4.7, 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 solution is the Cartesian product of all partial solutions. This approach has several advantages:

- The complexity of solving two independent goals A and B separately is $b(A) + b(B)$ rather than $b(A) + b(A) \times b(B)$.

- The subgoals can be solved in parallel.
- If any of the independent goals has no solutions we can abort the whole query and report it has no solutions.
- The solution can be expressed much more concisely as the Cartesian product of partial results. This feature can be used to reduce the communication overhead.

This optimisation can be achieved by replacing the calls to `sort/2` and `combine/2` in the algorithm of figure 4.10. The replacement performs two task: identify which (independent) projection variables of the query appear in each of the sub-conjunctions and create a call to the runtime routine `serql_carthesian`(+*GoalsAndVars, -Solution*) that is responsible for planning and executing the independent goals.

4.10 Evaluation

We evaluated three aspects of the system. The amount of source code illustrates the power of Prolog for this type of task. Next, we study the optimisation of the WordNet query given in figure 4.7 and finally we examine 3 queries originating from a real-world project that triggered this research.

Source code metrics The total code size of the server is approximately 6,700 lines. Major categories are show in table 4.1. We think it is not meaningful 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 how much is reused from the respective system libraries to make a detailed comparison feasible. As often seen, most of the code is related to I/O (57%), while the core (query compiler and optimiser) is responsible for only 26% of the code.

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
Total	6,700

Table 4.1: Size of the various components, counted in lines. RDF/XML I/O is only a wrapper around the SWI-Prolog RDF library.

Evaluating the optimiser We have evaluated our optimiser on two domains: the already mentioned WordNet and an RDF set with accompanying queries from an existing application. Measurements have been executed on a dual AMD 2600+ machine running SuSE Linux and SWI-Prolog 5.5.15.

First we study the example of figure 4.9 on page 56. The code for `s1/1` was handcrafted by us and can be considered an educated guess for best performance. The described optimiser converts both `s1/1` and `s2/1` into `o1/1` as shown in figure 4.11. Table 4.2 confirms that the optimiser produced a better version than our educated guess and that the time to optimise this query is negligible.

```
o1(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))
```

Figure 4.11: Machine optimised WordNet query

Translation	opt. time	exec time	total time
`s1/1`	-	3.58	3.58
`o1/1`	0.09	2.10	2.19

Table 4.2: Timing of human (s1/1) and machine (o1/1) optimised translations of the query of figure 4.7. *Opt. time* is the time used optimise the query and *exec time* is the time used to execute the 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.[8] These queries were selected because Sesame (Broekstra et al. 2002) 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 section 4.9, our server does not need this rewrite. The results are shown in table 4.3. *Edges* is the number of graph edges that appear in the query. The actual time spent in the optimiser (3th column) is again negligible. The next three columns

[8] www.isoco.com

present the initial complexity estimate, the estimate for the optimised query and the relative optimisation based on these estimates. Because the initial queries do not terminate in a reasonable time we can not verify the estimated speedup ratio. The *time* column gives the total query processing time for both our implementation and Sesame. For Sesame we only have results for the second query as the others did not terminate overnight. The last column gives the total number of rows in the output table. The solutions of the first and last queries are Cartesian products (see section 4.9).

		time	complexity			time		
Id	Edges	optimise	initial	final	speedup	Us	Sesame	solutions
1	38	0.01	1.4e16	1.4e10	1.0e6	2.48	-	79,778,496
2	30	0.01	2.0e13	1.3e05	1.7e8	0.51	132	3,826
3	29	0.01	1.4e15	5.1e07	2.7e7	11.7	-	266,251,076

Table 4.3: Results on optimising complex queries. Time fields are in seconds.

4.11 Related Work

Using logic for Semantic Web processing has been explored by various research groups. See for example (Patel and Gupta 2003) which exploits *Denotational Semantics* to provide a structured mapping from language to semantics. Most of these approaches 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 (Struyf and Blockeel 2003) because it is recent and both the problem and solution are closely related. They describe the generation of programs through ILP (Muggleton and Raedt 1994). 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 Struyf and Blockeel (2003)). 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 section 4.9. 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 (section 4.7) 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 (section 4.6).

4.12 Discussion

Current Semantics Web query languages deal with graph expressions and entailment under some Semantics Web language, which is typically implemented by computing the deductive closure under this language. As we demonstrated, graph expressions and conditions are easily translated into pure Prolog programs that can be optimised using reordering. For languages that are not expressive such as RDFS, entailment can be realised both using forward chaining and backward chaining. Figure 4.4 describes how we realise multiple reasoning schemes using Prolog modules and backward chaining. As long as the entailment rules are simple, the optimiser described here can be adapted to deal with the time and solution count estimates related to executing these rules.

As the Semantic Web evolves with more powerful formal languages such as OWL and SWRL,[9] it becomes unlikely we can compile these easily into efficient Prolog programs and provide estimates for their complexity. TRIPLE (Sintek and Decker 2002) is an example of an F-logic-based RDF query language realised in XSB Prolog (Freire et al. 1997). 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 (Frühwirth 1998; Schrijvers and Demoen 2004) are promising extensions.

4.13 Conclusions

In chapter 3 (Wielemaker et al. 2003b) we have demonstrated the performance and scalability of an RDF storage module for use with Prolog. In this paper we have demonstrated the feasibility of realising an efficient implementation of the declarative SeRQL RDF query language in Prolog.

The 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 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 fact that Prolog is a reflexive language (a language where programs and goals can be expressed as data) together with the observation that a graph pattern expressed as a list of triples with variables is equivalent to a Prolog conjunction of `rdf/3` statements provides a natural API to RDF graph expressions and the optimiser: translating a complex Prolog goal into an optimised one. Prolog's non-determinism greatly simplifies exploration of the complex search space in the generate-and-test cycle for finding the optimal program. The optimisation relies (section 4.6) on metrics maintained by the RDF store described in chapter 3 (Wielemaker et al. 2003b). Many of these metrics only apply to RDF, which justifies

[9]http://www.daml.org/2003/11/swrl

our decision to implement `rdf/3` as a foreign language extension in favour of adding optimisations to Prolog that make `rdf/3` feasible as a normal Prolog dynamic predicate.

The HTTP frontend we developed to create a Sesame compliant HTTP API has been the basis for the development of ClioPatria chapter 10 (Wielemaker et al. 2008). The server has been extended with support for the W3C standard SPARQL language and HTTP API and is now an integral component of ClioPatria (figure 10.5). Reasoning inside ClioPatria uses direct Prolog queries on the RDF store and the optimisation library described in this paper is used for dynamic optimisation of goals with embedded `rdf/3` statements where the ordering is not obvious to the programmer.

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,[10] IST-2002-507967. Jeen Broekstra provided useful explanations on SeRQL.

[10] `http://www.hops-fp6.org`