Search the semantic web

van der Sluijs, K.A.M.

Award date:
2004

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

Search the Semantic Web

by
K.a.m. van der Sluijs

Supervisors: dr.ir. G.J.P.M. Houben
ir. R.Vdovjak

Eindhoven, August 2004

1 Preface

This document is the result of my graduation project. The graduation project concludes the Technical Computer Science course at Technische Universiteit Eindhoven (TU/e), which I started in 1997. The graduation project was carried out at the Information Systems group of the Computer Science Department at Technische Universiteit Eindhoven. I would like to thank the following persons for their contribution and their support to the project:

dr. ir. G.J.P.M. Houben as being my graduation advisor
ir. R.Vdovjak as being my graduation advisor
dr. A.T.M. Aerts as being member of the examination board

Further I would like to thank my fellow graduation students in room 7.45 for the pleasant time and many interesting discussions.

Finally, I want to thank my girlfriend Krista van Soldt as being my inspiration, motivation and support.

Eindhoven, August 16, 2004.
Kees van der Sluijs

2 Index

3 Introduction

This project centres on creating a search engine for the Semantic Web. In order to place this in a context this section will first give a short outline of the structure of the current Web and give a sketch of the implementation of a typical Web search engine. Then we will give an overview of the ideas behind the Semantic Web, and define the goals for this project.

3.1 The Web

In the past the World Wide Web consisted of a huge collection of mainly static HTML pages. An HTML page consists of formatted text, which is the content, and links. A link is a connection from one HTML page to some other file, with no further explicit semantics. This file can be another HTML page, but may also be some other file. Multimedia files, like video, images, music, etcetera, can be "embedded" in an HTML page. This means that a Web browser integrates the file that is referenced by an embedded multimedia link in the page view (so that the Web browser makes it seem as a part of the page, as part of the content).

Newer standards of HTML also allows for some restricted interaction through the use of scripting languages like JavaScript, which have limited functionality. Server side scripting languages like PHP and ASP introduce some richer behavior and allow for two-way communication (between server and client browser) and an adaptive behavior. Furthermore, currently most information is contained in databases and HTML pages are generated for presentation of the information to the user. The retrieval of information from the database may be triggered by either a specific user request (e.g. Google query results) or some implicit action by the user (e.g. selecting a news item on Tweakers.net).

The connection between Web pages consists of links. Links between pages do not have any further explicit semantics than that there is some kind of a connection. Therefore, nothing can be formally derived from a link. However, informally users tend to almost always use links as "recommendation". A link may target any file that can have a URL. If the target is an HTML page the link can also specify a named start anchor in the target HTML page with use of the '#' sign. If this is the case the browser automatically will show the part of the document starting with the named anchor (especially useful for large documents). A URL can also contain directives for document generation. The Web server that contains the resource that is identified by the URL then automatically recognizes these directives and executes the necessary steps.

3.2 Web searching

The currently used Internet search processes are keyword based. User input consists of a string of characters that denote keywords representing the core of the user's request for information. The result consists of a collection of links to documents that contain the keywords of the input.

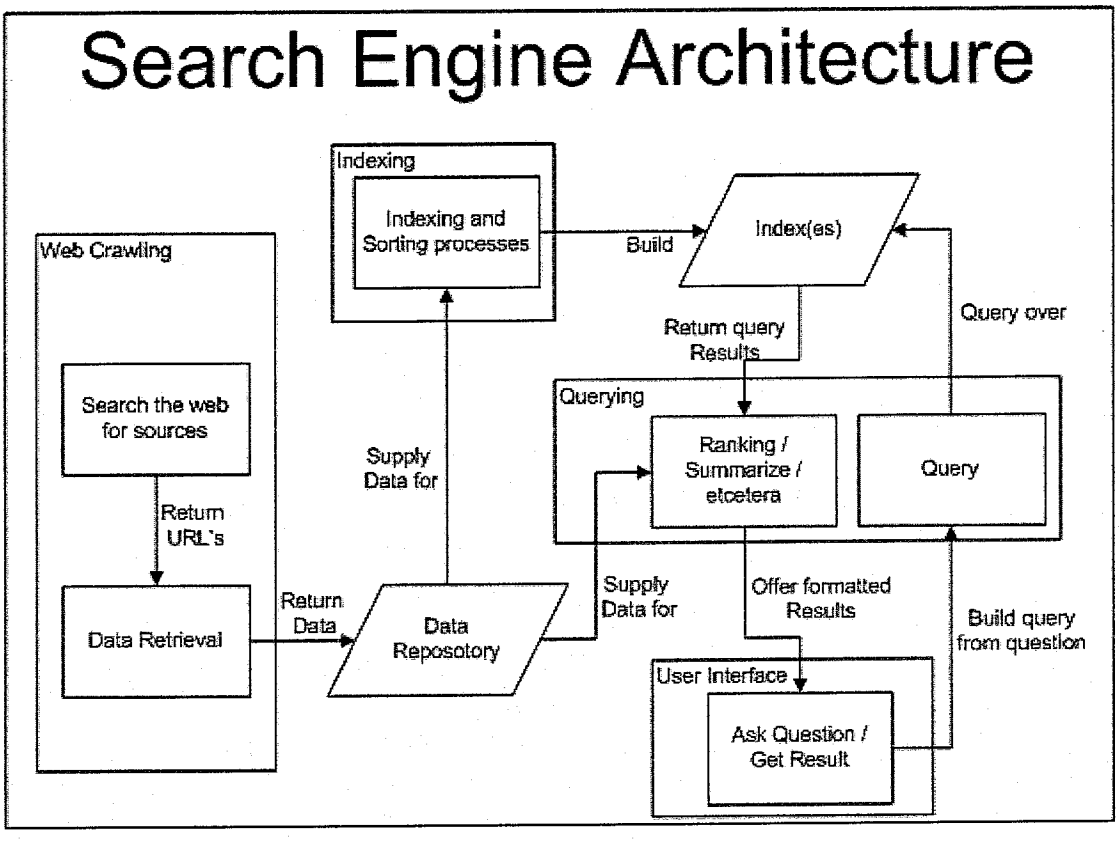

Figure 3-1: Model of the common used search engine architecture

Figure 3-1 is a simplified and general model of the architecture of present-day Web search engines. The rectangles represent the actions that such a system undertakes. The parallelograms represent data storage. The arrows represent the flow of information and the caption indicates the data relation between entities. A search engine consists of four main components and two data structures. The main components are Web Crawling, Indexing, Querying and the User Interface. The data structures are the data repository and the indices on the data repository

A Web search engine collects HTML pages and indexes the keywords found in those pages (with references to the corresponding pages). A search is executed by matching the query words in the indices. Then these matches are combined in a result list of links to resources that could contain the searched information. Before the results are presented to the user, the list of results is sorted on "relevancy". This is called ranking.

The ranking of a result set can be done in a lot of different ways. Existing algorithms can be divided in two groups. One group ranks the relevancy of a document for some query by comparing the contents of the document with the contents of the query. The other group ranks the relevancy of a document by its importance that is measured indirectly by considering the link structure as recommendations; thus a link from one Web document 'X' to another Web document 'Y' is a recommendation of 'X' for 'Y'. It is also possible to mix the two algorithm groups.

Current Web search engines can only exploit document contents and links, which both lack explicit computer interpretable semantics, because that is just the structure of the current Web.

New Web structures like the Semantic Web could provide the search engines with more information that can be utilized to improve searches.

3.3 Semantic Web

The World Wide Web, as it is, is human-readable and machine-processable, but it is not machine-interpretable for semantic interpretation or inference. In other words: machines have "no idea" what the content of Web documents is and are therefore unable to utilize the Web documents semantic contents.

The Semantic Web was invented to provide for semantic interpretable documents. The Semantic Web is envisioned to be machine "understandable", which means that semantic structures are explicated so that they can be utilized for all kind of purposes.

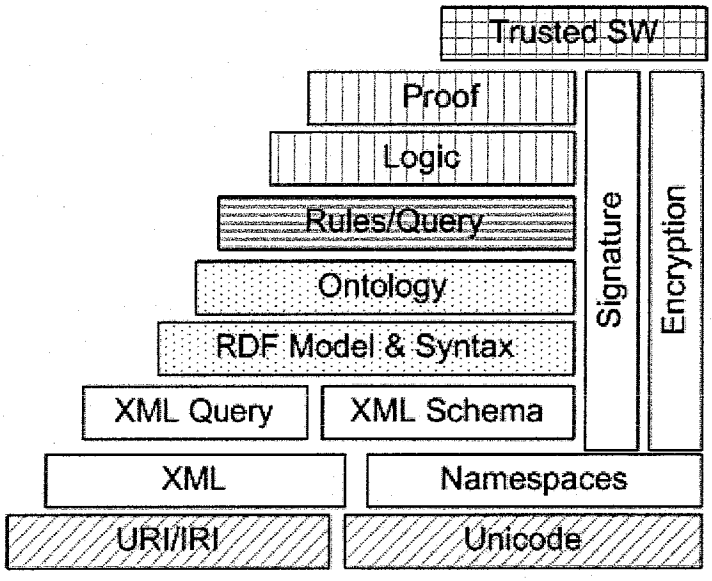

Figure 3-2: Berners-Lee Semantic Web Stack

The proposed Semantic Web can be thought of as a stack of extending layers. Figure 3-2 is the semantic stack as proposed by Berners-Lee, the inventor of both the World Wide Web and the Semantic Web. The bottom (diagonally striped) blocks of the Semantic Web depict the syntactic structure of the Semantic Web at character level. These blocks form the absolute basis and consist of plain text Unicode and URIs. Unicode defines the available character set and URIs provides a way to refer to abstract or physical resources. The white blocks (without pattern) depict the syntactic structure of the Semantic Web at text format level. This syntactic structure consists of the tag based XML structure, extended with the query languages for that structure. The dotted part is the semantic structure part, which consist of the RDF metadata structures and the ontology languages like RDFS and OWL. This part provides for the definition of resources and concepts and the relation between concepts. These semantic structures can be utilized by the application of rules or exploited by query facilities (horizontal striped part). The step after that is denoted by the vertically striped building blocks, which represent the logic and proofs between processes to convince one process of the correctness of the assumptions of the other process. The logic and proofs are introduced to enable processes and users to judge the reliability of results. And the final block is represented as the trusted Semantic Web, which is reached by the preceding steps extended with the services encryption, for security, and digital signatures. These last two are facilities that can be applied to the whole semantic part of the SW Stack. Encryption and signature are stacked on XML and

namespaces, because they are built on top of these facilities; an encrypted XML file should still be valid XML (where tags and content may be (partial) encrypted) and signatures could be exchanged in XML format. Encryption and signatures should enable determination of the level that some source of information can be trusted.

Note that the Semantic Web is a vision. It is something that may be established one day. The Semantic Web is actually the whole stack as displayed in Figure 3-2. Thus far only some of the building blocks of the Semantic Web have been implemented. RDF and RDFS now exist for only a few years. OWL is even younger and has only recently been accepted by W3C. If, in the rest of thesis, we speak about the Semantic Web, we mean the part of the Semantic Web that is yet realized. More specifically, in this thesis we will focus on RDFS.

3.4 Goal and project structure

The goal of this project is to investigate new techniques to improve the search process by utilizing the Semantic Web. Users should have a query expression power comparable with database queries, while the language should be kept simple enough to be usable by "regular users". The users should not be compelled to have inside knowledge of the data structure or schema information of the data they search in, yet once they know or discover it they should be able to take advantage of it.

This thesis will proceed with first giving a problem description in section 4. This problem description gives an outline of what the problems are that we want to solve in this thesis. Then, an extensive overview is given on related work in section 5. That section first concentrates on giving detailed descriptions of the fields that form the basis for this project, namely search engines and Semantic Web. After that we give an overview of work that is related to this project, so that one can relate this project to the ongoing research in the field. Section 5 is particularly useful for them, who are new to the field. They who are experienced in the concerning fields can safely skip section 5.

Section 6, the requirements, contains specific descriptions of the tasks we want our system to be able to do. There we distinguish between the types of queries we want to be able to execute and the query results processing we want the system be able to do. In section 7 we present a data structure that is able to efficiently accommodate for the requirements. Then, in section 8, we show that our data structure indeed accommodates for the execution of the queries we specified in section 6. We do this by giving the corresponding (conceptual) algorithms that use the data structure of section 7 to solve them. In section 9 we do the same for the results manipulation tasks.

In section 10 we describe an implementation of our system on top of an existing RDF database system. This system has its own data structure and query constructs on that data structure available. Then, in section 12, we make a performance analysis of both the conceptual data structure and the concrete implementation in Sesame. In both cases we measure the space overhead and the time complexity of the queries. Then we make a comparison between the two based on the measurements and draw some conclusions from this comparison.

Section 11 contains some details about the integration of the implementation, as described in section 10, in a GUI for browsing RDFS ontologies called EROS. The conclusive section, section 13, contains a retrospect on the project together with the conclusions that can be drawn. It concludes with a number of recommendations for improvements that can be made during future work.

4 Problem description

This project aims at investigating new techniques to improve the search process by utilizing the Semantic Web. This section motivates why new techniques in the search engine field are needed and where the current techniques fall short.

To capture the improvement, compared to current search engines, in a nutshell:
We want to improve the query power of the search engine to the level of (e.g. relational or object oriented) databases, while at the same time minimizing loss of the simplicity of the current search engine query languages.

A traditional database has a predefined schema and the users that query over it are supposed to know that schema. The Semantic Web however, does not have a fixed schema. For the Semantic Web everybody may define its own schema and may say anything about anything. This freedom of data modeling is of great use (because the world that is described by the data is heterogeneous itself), but poses some problems for the query engine that should search these structures. Furthermore we think the user should not be required to have any prior knowledge of the schema he searches on, like users are not required to have any prior knowledge of the data they search on on the Web. Therefore we want to keep it possible to use keywords in queries, instead of complete URIs. These requirements imply that a one to one translation of a database query engine is not possible. The intention of this project is to find a compromise between the two extremes.

The rest of the section gives examples of queries we want to be able to evaluate but cannot be evaluated by regular keyword based search engines. Further we point out how semantic information could further be exploited. An example of this is to find results that the user might also be interested in without these results relating directly to the user query.

Note that the syntax used in the examples is only informal, purely intended for illustration.

4.1 Metadata search

Current search engines enable users to search the full text of Web documents. They do not enable users to search for specific metadata. Possible meta information on Web documents comprises (for example):

- Date of (first) creation
- Author
- Subject
- Language
- Document Format
- Date of last update

The Web also consists of several kinds of documents. Beside the HTML documents there also are other document formats like PDF documents or Word documents. But in addition to text documents other media like video, pictures, sound and executables are also available on the Web, which all have their own format (MPG, AVI, JPG, PNG, WAV, MP3 to name a few). These files may have their own set of format specific meta-data. For instance the MP3 ID3v2 tag consist of the following metadata fields:

- Title
- Artist
- Album
- Year
- Genre
- Comment
- Composer
- Orig. Artist
- Copyright
- URL
- Encoded by

The following query cannot be executed on current search engines, but is a question you would like to be able to ask a search engine:

An example of a metadata query is:

All documents with: Subject {X} ∧ Author {Y}

Where X and Y may be (regular) boolean term expressions, e.g. X may be "Databases" OR "Hypermedia" and Y may be "Geert-Jan" AND "Houben".

Somewhat more complex is to allow other than boolean operators, for example ranges or dates. Consider the following query:

All documents with: Subject (X) ∧ Date {Between(Y,Z)}

Where X may be a (regular) boolean term expression and the date query may contain a range between two dates Y and Z, e.g. X may be "Databases OR Hypermedia" and Data may be "Between(1999,2001)". The query should only return documents dating from between 1999 and 2001.

Users should also be able to combine several queries with boolean operators, e.g. a query like "return all documents that have author Geert-Jan Houben or subject "HERA".

4.2 Path expressions

The Semantic Web does not only allow metadata on some document or file but also on the metadata itself. For example, beside stating who the author of some Web document is, one can also define some data on the author himself, like the author's background (industry, university, etc) or the author's e-mail address.

The semantics of the Semantic Web can be modeled as a directed labeled graph with data modeled as nodes and relations between data modeled as directed edges. This graph modeling is based on the fact that in a RDF triple the object may also function as the subject of another triple (except for literals). Querying the Semantic Web structure is analogous to querying a directed graph. This graph structure of the Semantic Web also allows for more complex queries then a normal metadata query.

RDF triples have the following pattern: <S,p,O>

Queries from section 4.1 can be expressed with such a pattern by defining which parts of the pattern have to fulfill a condition and which parts of the pattern are variable. Consider the query: "return all resources that have author Geert-Jan Houben". This could be expressed in pattern by: <?,author,Geert-Jan Houben>.

Now consider the following pattern, which represents a path:
$<X_0 , p_0, X_1, p_1, X_2, \ldots, X_i , p_i, X_{i+1}, \ldots, p_{n-1}, X_n>$

The defined path consists of 2n+1 elements. The defined path can be viewed as n triples. The presented pattern for a path relates to triples by means of the following predicate:
$[\forall j: 0<j<n : <X_{j-1}, p_{j-1}, X_j, p_j, X_{j+1}> \equiv <X_{j-1}, p_{j-1}, X_j > < X_j, p_j, X_{j+1}>]$

We now consider the category of queries, which we call path queries. A path query is a path pattern that defines conditions or variables for its components.

An instantiation of such a path query in natural language is:

Return everything with an "author" relation "De Bra" , who has a "faculty" relation, which has a "part of university" relation "TUE".

This can be written in a path pattern as follows:
< ?, author, De Bra, faculty, ? , part of university , TUE>

According to the given predicate this comes down to all the combinations of three triples that respectively fulfill the following triple patterns:
<?, author, De Bra>
<De Bra, faculty, 'x1'>
<'x1', part of university, TUE>

Where 'x1' is a variable that is used twice as 'x1' instead of '?' to express that the values must coincide.

A problem with the stated definition for the category of path queries is that we want that users do not need to know the data schema they query on. They, for example, may not know the structure: "author --> faculty --> university". To translate this, we define the notion of *path distance*. In a Semantic Web graph the *path distance* between a start and end node is the number of directed edges that have to be traversed to get from the start node to the end node. We could then let a user define a maximum distance between two node values instead of obliging them to know the complete model. If we search for a author "De Bra" who is somehow connected to "TUE" we could define a maximum distance of , for instance, 3 (larger values imply that a relation with greater distance is too "vague"). The example path query could then (in natural language) be:

Give me all resources that have a relation "author" with a value "De Bra" that has a maximum distance of "3" with some node that has value "TUE".

As we noted before the Semantic Web can be represented by a directed labeled graph. The query examples stated before were following the direction of the arcs in the graph. It would

increase the expression power if the query also allows for following arcs in the opposite direction. This would enable queries like:

Give me all documents that are written by authors that have written something about "adaptive hypermedia systems".

4.3 Composite queries

Beside the singular queries that are proposed earlier this section, users often have a more complicated view of the results that they are looking for. As a group of more complicated queries we consider composite queries. An instance of composite queries is the group of queries that fulfill multiple conditions. Queries like

Give me all papers of "De Bra", but they have to be in PDF format.

Another example are queries that defines several possibilities of which the results should fulfill at least one, like

Give me all papers that have author "De Bra", but papers that have subject "Hypermedia" will also do.

As last example we consider queries that must fulfill some condition, but may not fulfill another, like

Give me all papers of "De Bra", but they may not be in postscript format.

4.4 Language ambiguity

A problem for search engines (including current ones) is the ambiguity of natural language and the existence of several synonyms or denomination for the same concept (possibly even in different languages). Further, words or names may have multiple correct spellings. Therefore it is desirable that a search engine offers its users some tools to try to overcome these problems.

An example of this is if a user searches for all resources that have a relation "author" with value "X". Then, the search system should also return the resources that have a relation "writer" with value "X". The search engine should therefore infer that the concept "author" is equivalent to the concept "writer". Another example is a search for all "authors" that have written something about "search engines". If the search engine knows (e.g. by user preferences) that the user also understands Dutch and German you may also want to find authors that have written about "search engines" in those languages, so also e.g. writers that have written about "zoekmachines" or "Suchmaschinen".

Preferably this reasoning should not have to be manually inserted into the search engine. Inference rules could be generated with help of existing RDF(S) or OWL lexicon graphs and a number of inference rules on those graphs. Another possibility is to let a group of users define their own inference rules (possibly on top of existing rules) and share these rules for all users for possible later use. Note that it is important that users are informed about how the query results relate to the query.

For the spelling problem users could get informed if some of the terms or URIs in the query return very few results, while a syntactic variation of the term (with e.g. a 90% correspondence) does return many more results.

4.5 Performance and Scalability

Building a search engine for the (Semantic) Web, one has to take into account the size of the Web. The Web is vast. The Semantic Web is actually quite new, and therefore by far not comparable in size with the Web. But even indexing the entire Semantic Web will pose scalability problems. Therefore a Semantic Web search engine must be scalable. This means that for designing a search engine it should always be kept in mind what the storage overhead and time complexity for the several algorithms are. Furthermore it must be kept in mind that scaling-up also means parallelization. To process Web sized amounts of data one cannot possibly suffice with one computer, but requires a network of collaborating parallel computers. To accommodate for this, underlying data structures and data processing algorithms should be designed so that they can easily be fitted in a parallel environment.

355

5 Related Work

Before designing a Semantic Web Search engine it is important to know the current state of the art in the concerned field(s). This section outlines the findings of a literature study on that state of the art.

This project is aimed at researching the feasibility of a Semantic Web search engine. This touches two fields: information (Web) retrieval and Semantic Web. Therefore we will first look at literature about information retrieval, then at the Semantic Web and conclusively on existing (retrieval) tools for the Semantic Web.

For the Semantic Web section 5.2 contains a description of the Semantic Web languages from the lowest syntactic level (RDF) to the highest currently available ontology language (OWL). The concluding section, section 5.3, contains an overview of some existing tools for the Semantic Web. Most of the reviewed tools are based on querying, but there is also a Semantic Web visualization tool.

5.1 Information retrieval

Because we want to design a search engine it is useful to investigate the currently used techniques for information retrieval and Web retrieval in particular. For general information retrieval techniques we refer to the book "Modern Information Retrieval" [1]. This section will discuss the working of the most popular search engine today – Google [2].

The strong points of Google are:

- A very simple intuitive understandable user interface
- Efficiency optimization of both storage needs and the number of computing operations.
- Massive use of parallelism, which enables the speedup of searching, indexing and querying
- Brute force computing (>20.000 computers), which results in a large number of indexed pages and quick query evaluation
- The pagerank algorithm measures the importance of pages, which is used for the ranking of results.

Figure 5-1 ([2]) shows the high-level Google architecture. The paper mainly deals with three parts of a search engine: the crawling part, the indexing part and the search part. This paper leaves out the User Interface.

The Web crawler (or spider) part of a search engine is the link of the system with the Web. The Web crawler browses the Web in an automated and methodical way to collect data, in order to enable the search engine to search over it. In order to scale to the enormous amount of data on the Web, Google has a fast and distributed crawling system. DNS lookups are a major stress with crawling. Therefore Google maintains a DNS cache so that the crawling of documents do not always require a DNS lookup. Furthermore to cut down bandwidth usage, their servers are on distributed locations so that the machines run relative close to the sites they crawl. All crawled pages are stored (compressed) in a repository.

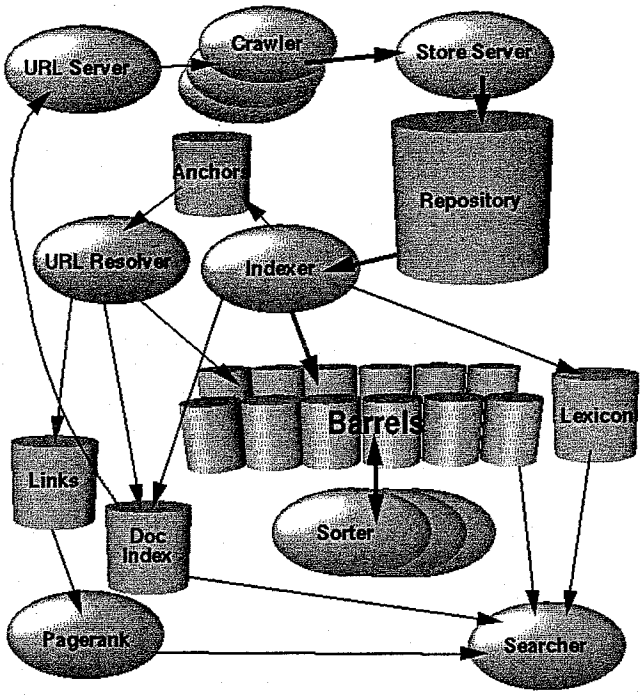

Figure 5-1: High Level Google Architecture

After crawling, the crawled data from the repository is prepared for querying by indexing and sorting the data. This is done in the indexing part of the search engine. The first part of indexing data is parsing the Web document. The problem with Web documents is that very little can be assumed about them and syntax errors in for example HTML are very diverse. "*These [errors] range from typos in HTML tags to kilobytes of zeros in the middle of a tag, non-ASCII characters, HTML tags nested hundreds deep, and a great variety of other errors that challenge anyone's imagination to come up with equally creative ones*" [2]. Therefore Google needed (and implemented) a very robust and flexible parser, which is quick enough to keep up with the crawling pace.

The Google search engine contains a number of large data structures. The important ones besides the repository are the document index, the forward index, the inverted index and the lexicon. The document index maintains information about every Web document known to the search engine, like current document status and some statistics. After a document is parsed it is written into the forward index. The forward index (and also the inverted index) is divided in parts called barrels. Every barrel contains a range of wordIDs. The lexicon contains a list of words and their corresponding wordIDs. After parsing, if a document contains a word that is located in some barrel, the documentID is added to that barrel followed by a list of wordIDs with hit lists (a list of occurrences of a word within a document). The inverted index is similar to the forward index, only now processed by the sorter. Google maintains two inverted indices. One inverted index for title or anchor hits and one for full text hits. The smaller title and anchor hit index is searched first because of the probability of better hits and if there are not enough results also the full text index is searched (currently Google stops after finding 1000 results).

After indexing and sorting, the data is ready for user queries. When a user query is inserted, the query is first parsed and the words are converted in wordIDs. After that the short inverted index is searched for every word. The doclists are scanned until there is a document that

matches all the search terms. Following, the rank of the document for the query is computed. This is repeated until there are enough hits or the end of the short inverted index is reached. If the end of the short inverted index is reached and there are not enough matches the process is continued for the full text inverted index.

Google's distinguishes from other search engines with their ranking algorithm. Ranking is the art of sorting search results on relevancy. Googles ranking algorithm consists of the combination of an IR score and a Pagerank score. The IR score is similar to other search engines' ranking algorithms. Documents that contain some query word relatively often are ranked higher than documents that contain these query words less often. Word position (words in title or at the beginning of a document are considered more important) and word format (bold words or words in a relatively larger fonts are considered more important) are also taken in account. These numbers are combined and form the IR score of the document against the query. The Pagerank of a document is calculated by the link structure. A page is more important if more important sites link to it. Combined, the IR score and the Pagerank give a document a weight in relation to the query and decide the position of the document in the ranking.

To accommodate for the enormous amount of indexed data (estimates indicate about 6000 terabytes of data) and the evaluation of the enormous amount of queries (estimates indicate over 200 million queries a day) Google must rely on parallelism. Google's philosophy is to cluster many cheap computers that are easily replaceable on failure. Therefore the algorithms and data structure for this parallel cluster are constructed so that failure of one machine has no impact on indexed data or query evaluation. Estimates indicate a number between 10.000 and 80.000 servers in the cluster. Exact information figures are classified because of competition considerations.

5.2 Semantic Web languages

Because we want to construct a search engine that uses the Semantic Web, we should have a look on the available Semantic Web languages. As mentioned before, the Semantic Web is a vision, something that is still in the making. Therefore, in this section we only treat the Semantic Web languages that are constructed thus far and for which an implementation exist.

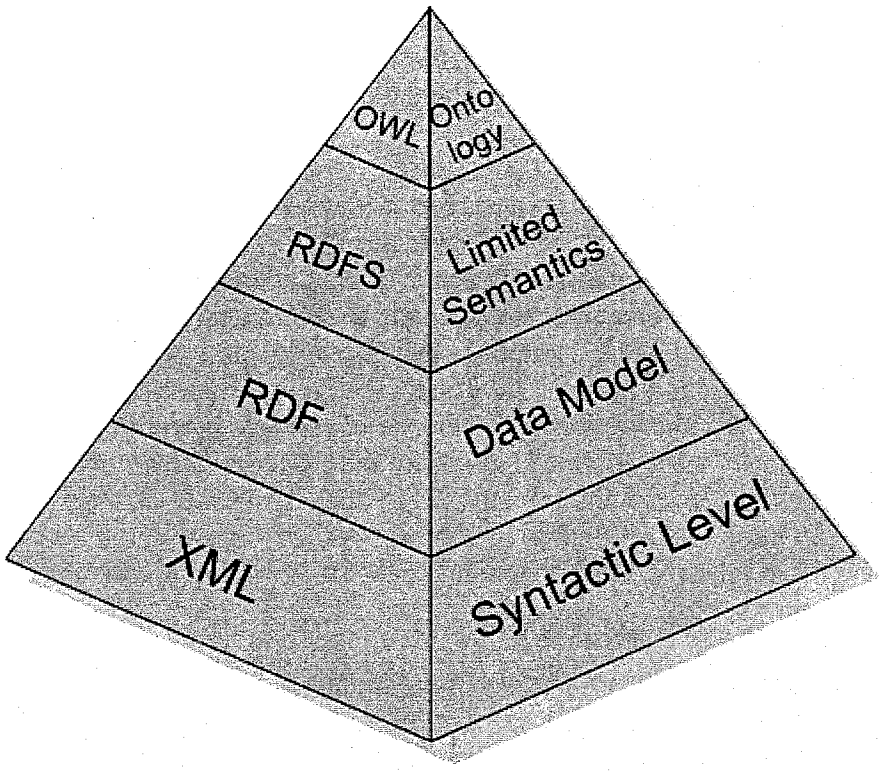

Figure 5-2: Semantic Web languages Pyramid

As Figure 5-2 shows, the current available languages can be viewed as part of a pyramid. The lowest part of the pyramid consists of a syntax specification. The consecutive top parts are extensions of its lower parts. They subsequently add data modeling and semantics to its lower parts until the top part of the pyramid, which enables to create full-blown ontologies.

In this section we will look at the following languages: XML, RDF, RDFS and OWL.

5.2.1 XML

The lowest syntactic part is XML. XML is basically intended as a syntactical uniform format language that is platform, user and application independent. Important design goals for XML were ([6]):

- XML shall be straightforwardly usable over the Internet.
- XML shall support a wide variety of applications.
- It shall be easy to write programs, which process XML documents.
- The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
- XML documents should be human-legible and reasonably clear.
- The design of XML shall be formal and concise.
- XML documents shall be easy to create.

XML does not provide for semantics. It only provides a flexible syntax in which almost anything can be expressed. XML can be seen as the counterpart of a database. Where a database has a rigid schema that is predefined, XML can be used heterogeneously. The schema

within XML may be irregular and only partially defined and is therefore called semi-structured.

Semi structured data can be characterized as follows ([7]):

- object-like
- schemaless
- self-describing

Here follows a very brief and not complete outline of the syntactic structure (based on [7]):

XML is like HTML. XML uses tags to create structure. However, XML has no fixed set of tags, no fixed semantics of tags and no fixed structure. The author may freely choose tags. However every opening tag must have a matching closing tag. Furthermore tag pairs may be (properly) nested. Tags can contain attributes. Attributes have a unique name, which have a value (must be quoted). Data may be freely mixed with tags. Usually there is some connection between data and the tag pair it is enclosed by, but this is not required.

5.2.2 RDF

RDF, or Resource Description Framework, is a metadata language built on the syntax of XML. RDF is meant to make statements or "descriptions" about resources. The foundation of RDF is a model for representing named properties and property values. The basic data model of RDF consists of three object types.

- Resources. All things being described by RDF expressions are called resources. This may be entire Web pages, a part of a webpage, a collection of webpages, or every other object that can be uniquely identified (like a printed book). The Resource identifier is called URI.
- Properties. A property is some kind of relation used to describe a resource.
- Statements. A specific resource together with a named property plus the value of that property for that resource is an RDF statement.

Every RDF statement is a triple with a subject, predicate and object. The subject is some resource with an URI (or blank node). The subject may also be a RDF statement, which provides for nesting in RDF. The predicate defines a property of that resource and the object defines the value of that property. A predicate is defined by an URI of the property it denotes. The object may be another resource URI, a blank node, or it may be a literal. A literal can be a simple string or another primitive datatype defined by XML.

The subject-predicate-object triples can be interpreted as a directed labeled graph. Both subject and object are nodes and the predicate defines an edge between the nodes and represents the relation between the nodes. Subject and object are interchangeable. The object of one triple may be the subject of another and the other way around (except literals, which may only be used as object of a triple).

The former rules for RDF are enough to express anything about anything. In RDF no other data modeling commitments are made.

For more information on RDF refer to [8] and [9].

5.2.3 RDFS

RDF properties may be thought of as attributes of resources and in this sense correspond to traditional attribute-value pairs. RDF properties also represent relationships between resources. However, even though RDF provides for resource description with help of a graph model it still is not machine-interpretable. Because anything may be stated about anything a machine still cannot interpret the intended semantics of some relation nor does RDF provide any mechanisms for describing the relationships between properties and other resources. The good thing about RDF is that with its data model it abstracts from the used syntax. There is however still a need for a universal agreement on the semantics of certain terms and the interpretation of certain statements, so that semantics of statements are machine understandable and graph models are reusable for several sources.

RDFS (S stands for Schema) provides for the missing semantics of RDF. With RDF Schema one can define particular vocabularies for RDF data and specify the kinds of objects to which predicates can be applied. RDFS expressions are also valid RDF expressions. Examples of RDFS agreed semantics are properties like subClassOf, domain and range. This extra structure can be used to define more expressive queries.

RDF graphs (or parts of a graph) are substituted with partially defined semantics. This approach enables the utilization of semantic relations in the graph model. For example if the query searches for all instances of some class, also all instances of subclasses can be taken into account.

Here follows an overview of the most important concepts in RDFS:

- **Class**: RDFS defines the notion of class, which is similar to the class notion of object oriented programming languages. Associated with each class is a set, called the class extension of the class, which is the set of the instances of the class. Two classes may have the same set of instances but be different classes. A class may be a member of its own class extension and may be an instance of itself. The group of resources that are RDF Schema classes is itself a class. ([10])
- **subClassOf**: RDFS defines subClassOf as a hierarchical structure between classes. If a class C is a subclass of a class C', then all instances of C will also be instances of C'.
- **Property**: Property is a concept of RDF. Within RDFS this concept is extended with the concept subPropertyOf, Domain and Range.
- **SubPropertyOf**: RDFS defines subPropertyOf as a hierarchical structure between properties. If a property P is a subproperty of property P', then all pairs of resources which are related by P are also related by P'.
- **Domain**: rdfs:domain is an instance of Property that is used to state that any resource that has a given property is an instance of one or more classes. A triple of the form: P rdfs:domain C states that resources that have property P are instances of class C.
- **Range**: rdfs:range is an instance of Property that is used to state that the values of a property are instances of one or more classes. A triple of the form: P rdfs:range C states that the value of property P of some resource is a instance of class C.

For more (extensive) information about RDFS refer to [10].

5.2.4 OWL

RDFS is an extension of RDF and adds predefined semantics to RDF statements. However, the number of semantic facilities is quite restricted. Thereby, the facilities that are available are not powerful enough and in some cases not sufficient. To extend the Web such that it is machine understandable in the way that machines can robustly reason about it, we need a powerful language to create and formally describe ontologies.

Several ontology languages have been developed. The most used today is probably DAML+OIL. However, W3C has created a new ontology language based on DAML+OIL, which is called OWL. OWL is designed as an extension of RDF(S).

In comparison with RDFS, OWL adds more vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes. These features enable a precise and formal definition of semantics of concepts in an ontology.

OWL exists in three "flavors", where each sublanguage is an extension of its simpler predecessors, both in what can be legally expressed and in what can be validly concluded. OWL Lite supports the simplest constraint features and is the easiest to use but misses the expressive power of its successors. OWL DL supports the maximum expressiveness that is possible without losing computational completeness and decidability of reasoning systems. The name is chosen in aligning its correspondence with description logic. OWL Full supports maximum expressive power and syntactic freedom, but does this at the expense of any computational guarantees. An example of an OWL Full construct that is not in OWL DL is that in OWL Full classes can be treated simultaneously as a collection of individuals and as an individual in its own right, where this is not the case in OWL DL (but is in RDFS).

For more (extensive) information about OWL refer to [11].

5.3 Semantic Web tools

We studied the intended use of the Semantic Web and the available languages for the Semantic Web. It is easy to see that, even though the Semantic Web languages are syntactically kept fairly simple, they are still far too complex for "regular", non-computer scientists, to manually handle. Therefore there is a great need for tools that assist users in handling the Semantic Web. The tools we examined are (in one way or the other) Semantic Web query tools except for one, which is a GUI for ontologies. We consider a number of query systems that can query increasingly complex semantic structures. QUEST (section 5.3.1) is able to query syntactic structures like XML. QuizRDF (section 5.3.2) tries to exploit the RDF model. OWLIR (section 5.3.3) uses semantic annotations to expand queries for keyword based search engines. HOWLIR (section 5.3.4) is a specialized agent that uses semantic annotations to keep track on events. The Semantic Web tool section is ended with a look at EROS (section 5.3.5), an RDFS visualization tool, which is useful to get an idea of possibilities to create an understandable GUI for this Semantic Web search project.

5.3.1 QUEST

The QUEST system is a query system for OHTML [12]. OHTML is an XML-like language with HTML and OEM references, which in turn can contain other OEM references. The authors claim that their query language is general enough to also be applicable to XML documents. They describe their system as: "QUEST is a system for querying hypertext documents that also embed some object structures. Due to the diversity of the Web, we cannot expect that documents with object structures will conform to a fixed schema, as in a classical database. Object structures that show some regularity, but do not follow a strict explicit schema, are captured by semi structured data model" [12].
OHTML can be represented as a graph, with the objects as vertices and the references as edges. Therefore it should also be possible to make QUEST applicable for RDF, which also can be represented as a graph.

Queries in Quest consist mainly of two parts: A graphical query graph part and a constraint part (see Figure 5-3 for a screenshot). The query graph is matched against the database graph during the search phase. The constraint part is used to specify constraints for particular vertices. If some vertex happens to contain some numerical value, the constraint can be used to restrict the range of the results for that value.

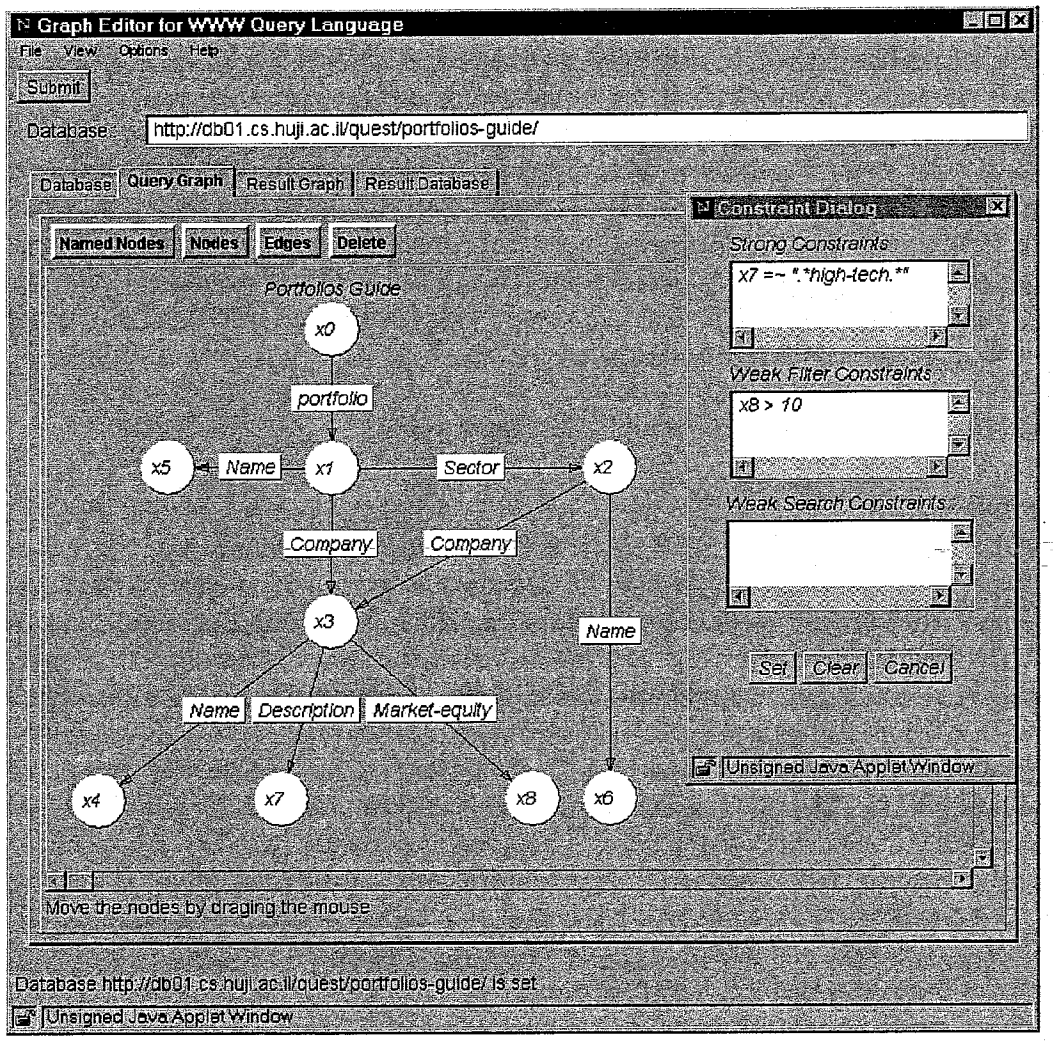

Figure 5-3: Graphical query part of QUEST

Query evaluation is executed in three phases in QUEST. The first phase is called the search phase. In this phase the query graph is matched to the database graph in search for similarity patterns. In the second phase the constraints of the constraint part of the query are used to filter the found matches. The final phase is the construction phase. The result set is now built from the result of the filtering process. The result set is in turn a collection of OHTML pages.

QUEST's strong point is its GUI. The GUI may make users intuitively familiar with graph like structures of data. The downside is that QUEST is made for OHTML and not for a W3C standard like RDF, but this could be changed in the future. The contrast with the system we target for is that QUEST requires users to know the schema information of the data they search for and we want a system that not specifically requires this knowledge. Further, QUEST's evaluation method of first doing the graph pattern matching and then applying constraints can pose a performance threat for large data sets. However note that, unlike the system we target for, QUEST is probably not intended for Web sized data sets.

5.3.2 QuizRDF

QuizRDF is a search engine that combines free-text search with a capability to exploit RDF metadata in searching and browsing ([13]).QuizRDF indexes Web documents as triples mapped to a URL. In general, sets of the type <literal,class,property>->URL are produced during indexing. The literal may be some RDF literal but also a part of the body text of the indexed document. The class and property part of the triple may be empty.

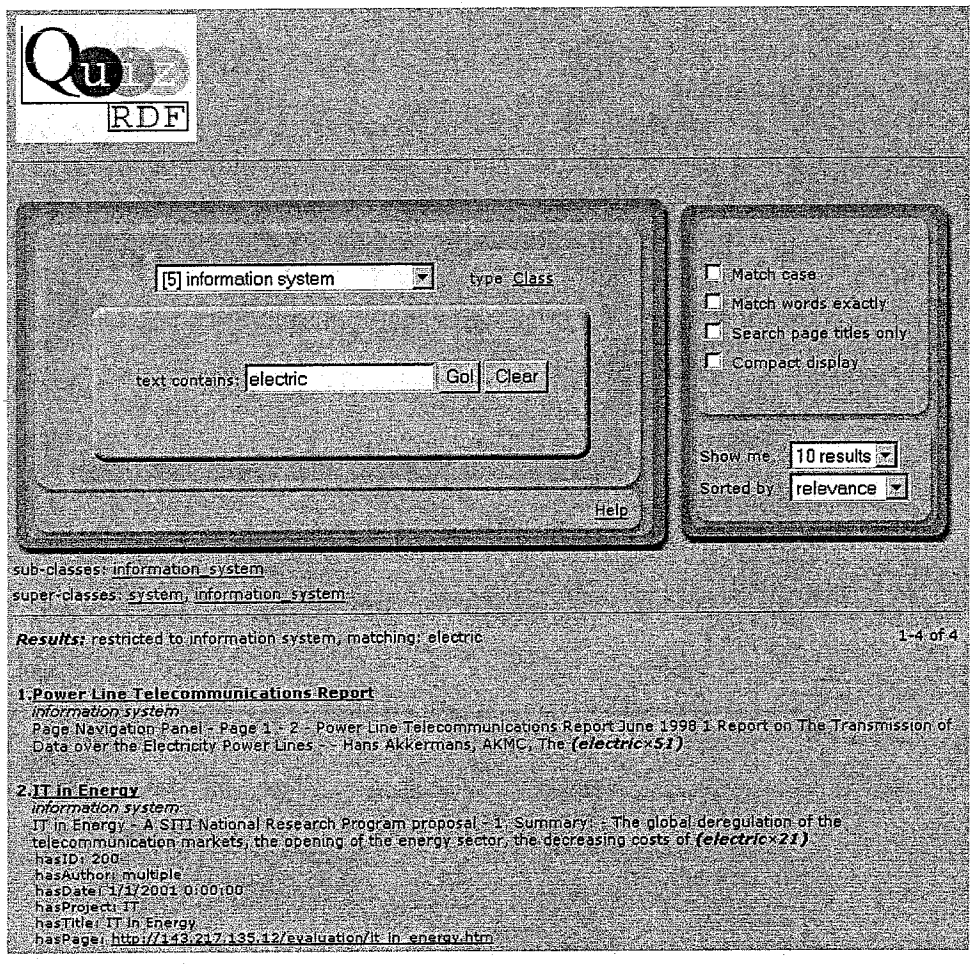

Figure 5-4: QuizRDF GUI

The user interface of the QuizRDF (see Figure 5-4 for a screenshot) is kept as simple as possible. Opening the QuizRDF webpage shows a textbox in the center of the user interface, which works basically the same as current search engines: query terms are entered and if the search is initiated those terms are searched in the full body text indices. On the right of the textbox some options are displayed which are pretty straightforward and also common in regular search engines. A dropdown box located above the search text box makes the difference with regular search engines. Initially this box contains a list of all resource types stored in the QuizRDF index. After some keyword search this list is replaced by a compiled list of the classes to which each document belongs. Selecting a class from this list will filter the initial results, and result in the documents of the results that are of the selected class. When a class is selected from a dropdown box two lists are compiled. One contains the list of superclass of the selected class and the other contains the list of subclasses of the selected class. Selecting one of the items of these lists will result in filtering the initial results for the results that are of the selected class or one of the classes in the subtree of the selected class.

The strength of this system is its simplicity. The query system is, like Google, keyword based, only now with an explicit metadata search. Further it contains the class tree navigation for filtering results. However, it differs with the system that we target in the sense that it does not increase the expression power of its queries. The system we are interested in should be more a hybrid between keyword based and database expression power, while QuizRDF clearly chooses for the keyword based approach.

5.3.3 OWLIR

OWLIR is a SW information retrieval framework that is constructed to solve t he following difficulties SW information retrieval ([14]):

- Current Web search techniques are not directly suited to indexing and retrieval of semantic markup
- Current Web search techniques cannot use semantic markup to improve text retrieval
- Likewise, text is not useful during inference
- There is no current standard for creating or manipulating documents that contain both HTML text and semantic markup.

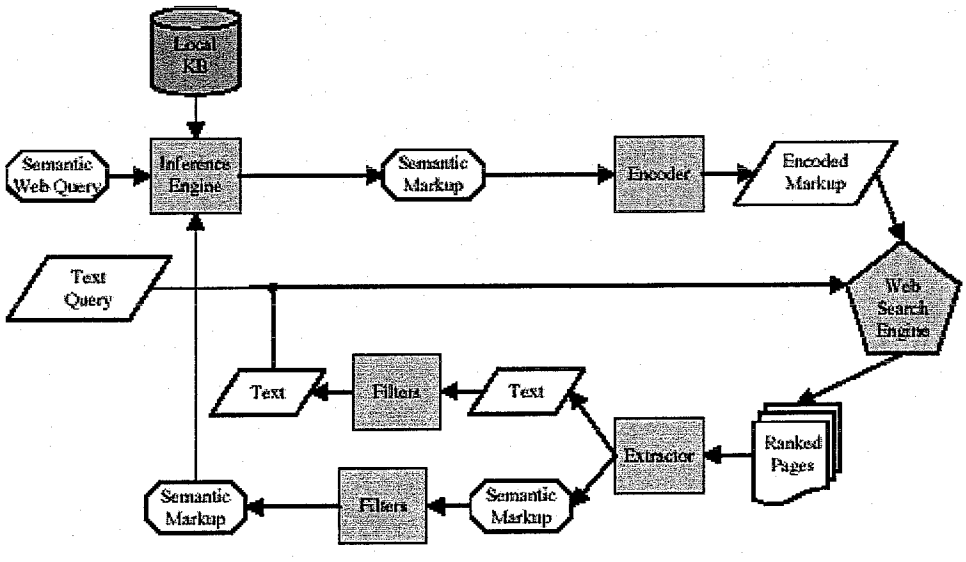

Figure 5-5: Model of OWLIR

The proposed framework, OWLIR, basically works like this (this is modeled in Figure 5-5 ([14])):

1. A Semantic Web query is inserted into the framework
2. With help of a local Knowledge base and an inference engine the query is expanded
3. Following, it is encoded and 'swangled' to form the input of regular Web search engine
4. The output of the regular Web search engine is extracted into two parts: text and semantic markup.
5. The text that is found is filtered and if needed used to refine the query for the regular Web search engine
6. The semantic markup is filtered and merged, with help of the inference engine, into the knowledge base and is if needed used to refine the semantic markup part of the query.
7. After the results are judged as sufficient, they are returned to the user

Neither [14] nor the model clarifies what stop criteria is used for both, how "semantic markup" is encoded to a query for a regular term based Web search engine and how semantic markup is extracted from query results from a regular search engine. The authors claim (however tested on a very small test set) that their approach is three to four times better then regular text searching (measured in mean average precision).

The working of OWLIR is based on the authors' observation of a typical information retrieval session ([14]):

1. A person mentally forms a semantic query.
2. The person encodes the query as a combination of words and phrases that are thought to characterize documents that contain information needed to answer the query.
3. The computer system retrieves a ranked set of documents matching the text query.
4. The person reviews some of the highly ranked documents, reading and extracting some of their meaning.
5. If the semantic query can now be answered, the process terminates with success. Otherwise,
6. Some of the newly extracted facts and knowledge are used to reformulate the text query, and the process is repeated.

At time of writing no working prototype of OWLIR was available. Therefore the performance of this system could not be directly measured. However, some things may be inferred from the model. OWLIR is founded on an existing web search engine. A query therefore at least takes the time of querying the underlying existing Web search engine. OWLIR, however, uses the results of the underlying Web search engine for query refinement. This triggers a number of new queries for the Web search engine. Thus a query on the OWLIR system takes at least a multiplication of the evaluation time of the underlying Web search engine.

OWLIR is again different from the system that we target because it, like QuizRDF, focuses on constructing a term based query engine (or uses such an engine as its basis). This in contrast with our goal of target of constructing a search engine that provides for a database like expression power.

5.3.4 HOWLIR

The HOWLIR system ([15]) is an information retrieval agent that keeps track of 'events' in the university. The HOWLIR system is created to support three basic scenarios:

- Information retrieval (IR) - e.g., identify and rank relevant pages or documents
- Simple question answering (Q&A) - e.g., who is the governor of Alaska
- Complex question answering - e.g., what is the current situation in Algeria

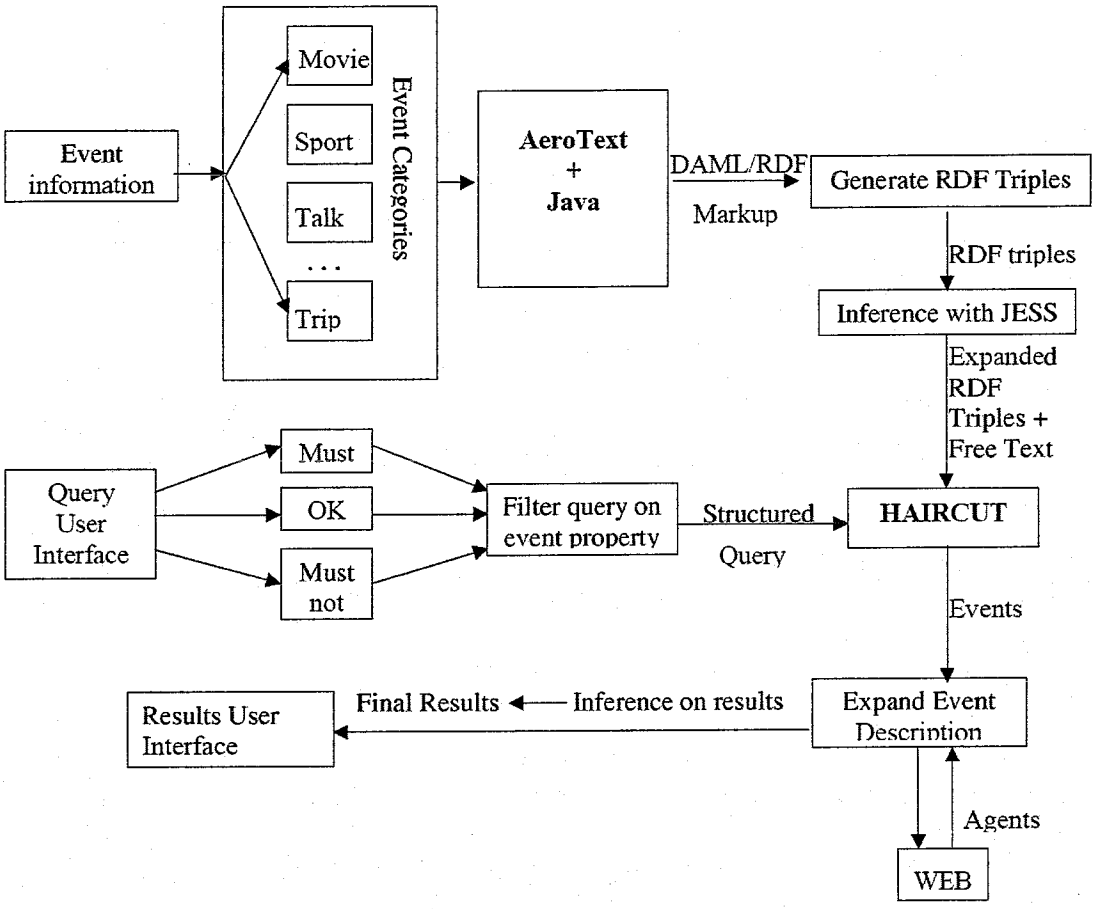

Figure 5-6: HOWLIR process flow

The HOWLIR system works as follows (see Figure 5-6 ([15])): Event information is gathered. Events are classified following some event categorization. After that semantic information is automatically generated with the AeroText system and this semantic information (in the form of RDF triples) is expanded with extra inferred triples and free text by the JESS system. These triples and free text are then indexed in the regular way, with a triple counted as one term.

The user can query the system with a DAML syntax query, with some particular operators like Must, Maybe and Must Not. The indexed data is then queried using the HAIRCUT information retrieval system. The result of this query is a set of events. Following, the event description is expanded, generated with use of Web agents. The results that are gathered are then filtered with help of inference on the results and then presented to the user.

According to the author a prototype of HOWLIR is finished. The ongoing work is on building a sophisticated inference engine, which can develop an enhanced knowledge base from implicit and explicit inferences made from the DAML+OIL marked up documents ([15]).

The HOWLIR system is not intended as a Web search engine, but rather as specialized search agent. Therefore it is difficult to compare it to a system we target for. What is special about

HOWLIR is that it takes regular text as input, instead of e.g. RDF(S), and then infers the semantic markup for that input. Further, both the generated DAML/RDF triples and the free text are stored and indexed for retrieval. This is roughly similar with what we want to do: searching through both the schema and the free text hidden within the schema data.

5.3.5 EROS

EROS (Explorer for Rdf(s)-based OntologieS [16]) is a GUI for browsing of RDFS-based ontologies. In principle there are two common solutions for displaying such ontologies: graph-based solutions and tree based solutions. The advantage of graph-based solutions is that they clearly depict the internal structure of an ontology. The disadvantage of graph-based solutions is that the view tends to grow uncontrollably and more difficult to compute with an increasing number of nodes. Furthermore it is difficult to grasp the hierarchical structure, which is "hidden" behind the special edges and not reflected by the position of the class nodes. The advantage of tree-based solutions is that they are intuitively understood, because people are already used to the tree metaphor. The disadvantage of tree-based solution is that ontology graphs cannot be mapped to an equivalent tree.

The EROS system tries to combine the good part of both metaphors. The system defines two major interfaces, the Class Centric Approach (Figure 5-7 left) and the Property Centric Approach (Figure 5-7 right). For the class centric approach two identical hierarchical class trees are built, based on the rdfs:subClassOf property. Selecting a class from the left hierarchy will result in a visualization of all properties of that class that have as object one of the classes in the hierarchy. This visualization is rendered with an arrow from the selected subject class in the left tree to the concerning object classes in the right tree view. The property name is attached to the arrow as a label.

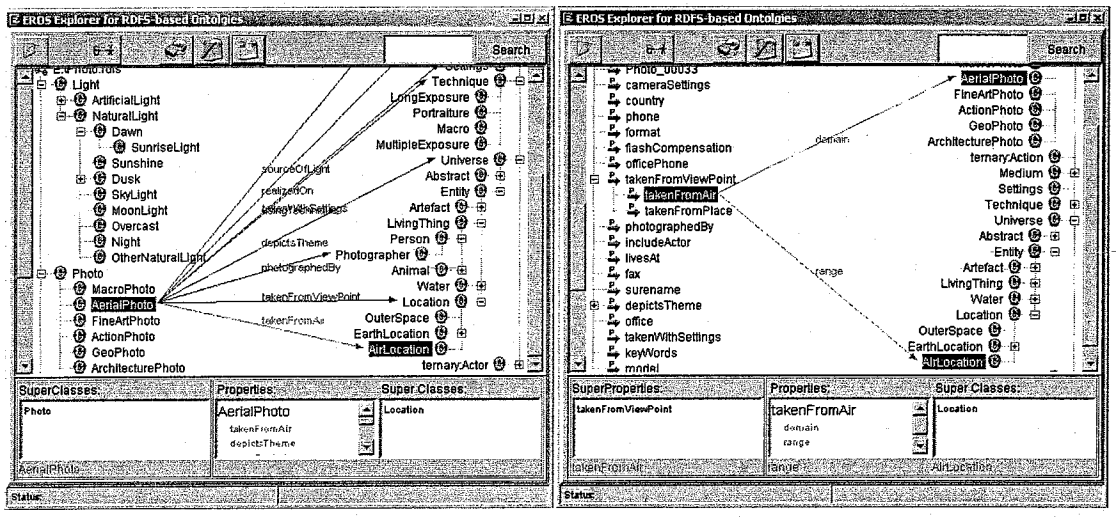

Figure 5-7: EROS class centric (left) and property centric (right) approach

For the property centric approach two trees are built. On the left is the hierarchical property tree, based on the rdfs:subPropertyOf property. On the right is the hierarchical class tree, based on the rdfs:subClassOf property. Selecting a property will visualize its domain and range property (rdfs:domain, rdfs:range) if the value of this properties is a class from the class hierarchy. The rendering is accomplished by an arrow from the selected property to the concerning classes.

790

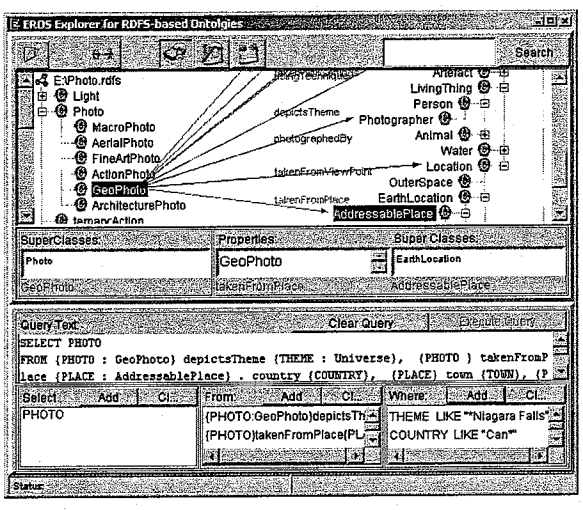

Figure 5-8: EROS query-building mode

This interface also enables users to formulate RQL queries with visual aid (see Figure 5-8). This is done by letting the user select classes and properties and let him/her be able to type in constraints for properties. With this information the system generates a RQL query for the user.

795

EROS is a GUI and not comparable with the system we target for. However, it may serve as a front end for our system. EROS strength is helping the user to visualize and browse ontologies. When browsing the ontology a user may be triggered for some information need and the system we target for could then be the back end that helps the user to satisfy this need. Note

800 that a problem with making a GUI for browsing web sized ontologies in general is that some node may have very many connections with other nodes. For instance the class "class" can have millions of subclasses. So, if one wants a visual front-end system for the search engine, extra facilities should be built in to accommodate for these 'size' problems. These problems occur with both discussed metaphors.

6 Requirements

Because of limited time, this project will not aim at the design and implementation of an entire Web Search Engine. It will focus on the query part of the search engine (and thus not the crawling and GUI part). It will also focus, at least for the design, on the indexing part because that part is important for query performance (and as described in section 4.5 we want a scalable solution). Further, as we explained in section 3.3, the Semantic Web is a vision, and it is only partially implemented. In this thesis we will therefore constrain ourselves up to the RDFS part of the Semantic Web.

This section will informally define the queries that the resulting search engine should be able to solve. The requirements of these queries are based on the problem description in section 4. The requirements can be divided in two groups, requirements for queries, and requirements for result manipulation algorithms. Further, as described in section 4.5, we want our search engine to be scalable to large data sets. Therefore, we want that evaluation of the basic query types be optimal concerning time complexity, preferable logarithmic.

6.1 Queries

Here we will define a range of queries the system should be able to execute and define some behaviour we expect from the system at evaluation. We assume that the queried structure is RDFS, and thus can utilize the class and property constructs. The queries are based on the problem description as stated in sections 4.1, 4.2 and 4.3. We define the following query types:

- Term queries (paragraph 6.1.1)
- Structure queries (paragraph 6.1.2)
- Class queries (paragraph 6.1.3)
- Property queries (paragraph 6.1.4)
- Path queries (paragraph 6.1.5)
- Union queries (paragraph 6.1.6
- Intersection queries (paragraph 6.1.7)
- Difference queries (paragraph 6.1.8)

6.1.1 Term queries

As we mentioned at the beginning of section 4 we want to want to construct a compromise between a keyword based search engine and a database. We think it is crucial to support, besides more powerful queries, also simple keyword search. Therefore we introduce the notion of "term" queries. An example of such kind of a query is:

Return all resources that contain term 'X'.

'X' consists of one or several terms. Note that for the queries we define in the rest of section 6.1, we always want to be able to replace URI's and literals, as existing within RDF(S) triples, by a term query.

6.1.2 Structure queries

Using RDFS, with its class and property structure, we not only want to be able to search for data using this structure, but also want to be able to search through these structures themselves.

Therefore we introduce the notion of structure queries. An example of such a structure query is:

Return all classes (or properties) that contain term 'X'

The query for properties is similar. Besides searching for some term in the (semi-) tree we also want typical tree operations to be executable. Therefore we want to be able to execute the following class structure queries (again 'classes' may be substituted by 'properties'):

Return all classes that are a child of class 'X'

Return all classes that belong to the sub-tree of class 'X'

Return all classes that are a parent of class 'X'

Return all classes that are ancestor of class 'X'

6.1.3 Class queries

The class structure in RDFS provides a mechanism for grouping resources into classes. A query type that naturally goes with this mechanism is query for all instances of some class. We call this type of queries class queries. An example of a class query is the query of the form:

Return all instances of class 'X'.

But we also want to consider the case that X is not exactly known, like

Return all instance of classes 'X' that contain the term 'Y'.

6.1.4 Property queries

Similar as with classes we want the system to be able to quickly solve queries that involve properties. Every predicate of an RDF triple is a property. So instead of only querying for properties (with for instance returning all subject and object pairs with the concerning property) it would be more general to query over triples. We call all queries of this type a property query. Property queries are able to fulfil the metadata queries as mentioned in section 4.1, except for the boolean combination part. The boolean combination part of the queries as mentioned in section 4.1 can be solved with the query combination primitives as described in section 6.1.6, 6.1.7 and 6.1.8. Note that queries of type property as described in this section are more general than the metadata queries. For property queries the subject does not need to be of type "document" but may be any resource, like for instance another RDF statement.

As already mentioned, we query triples. Triples consist of a subject, predicate, and an object. We want to be able to query with either one as the unknown factor. So, for example, the following query should be possible:

Return all triples that have predicate 'X',

Or

Return all triples that have predicate 'X' with object value 'Y'.

Or

Return all triples that have a subject value 'X' and an object value 'Y'.

Etcetera.

More formally defined, the following property query over triples must be possible:

$<S \mid ?, p \mid ?, O \mid ?>$,

where | represents the OR-operator, and ? means variable. For all S, p and O's, term queries may be used.

6.1.5 Path expressions

Beside singular queries we also want to be able to pose more difficult queries like path expressions as described in section 4.2. An example of a path expression is:

Return all instances of class 'X1' that have a property 'Y1' with an object value of class 'X2' that has a property 'Y2' with an object value 'Z'.

A concrete example of such a query is illustrated by the following realistic example:

Return all paintings that are painted by a painter living in the Netherlands.

We define the path query type by the following recursive definition:

$Path_0 :: <X_0 \mid ?, p_0 \mid ?, Path_1>$
$Path_i :: X_i \mid ?, p_i \mid ?, Path_{i+1} \quad \{0 < i < n\}$
$Path_n :: X_n \mid ? \qquad \{n>0\}$

where | represents the OR-operator, and ? means variable. For all X_i and p's, term queries may be used.

The result of this query is a number of groups consisting of n triples. One group would then be of the form:

$<X_0, p_0, X_1>$
$<X_1, p_1, X_2>$
...
$<X_i, p_i, X_{i+1}>$
...
$<X_{n-1}, p_{n-1}, X_n>$

Beside the normal path queries we also want to be able to use the path distance operator (defined in paragraph 4.2). The definition reads: In a Semantic Web graph the *path distance* between a start and end node is the number of directed edges that have to traversed to get from the start node at the end node. This yield the following path definition (where $[d_i]$ denotes a maximum distance operator):

$Path_0 :: <X_0 \mid ?, [d_0] \mid p_0 \mid ?, Path_1> \{ d_0 \varepsilon N \}$

$Path_i :: X_i \mid ?, [d_i] \mid p_i \mid ?, Path_{i+1} \quad \{ 0 < i < n , d_i \varepsilon N \}$

$Path_i :: X_n \mid ? , \{n>0\}$

6.1.6 Union queries

In section 4.3 we described composite queries. Those composite queries can be interpreted as set operators. We distinguish the union, intersection and difference operator. The union operator is equivalent with the queries (as described in section 4.3) that "define several possibilities of which the results should fulfill at least one." The intersection operator is equivalent with the queries (as described in section 4.3) that fulfill multiple conditions. And the difference operator is equivalent with the queries (as described in section 4.3) that must fulfill some condition, but may not fulfill another

All set operators expect two input queries, where the operator applies on the results of the results of these queries. We want it to be possible that the input query may again be a union, intersection or a difference query, so that these query types may be nested. This way a union of three items would be denoted in a way like: Union(Union(x1,x2),x3) or some equivalent form.

Union queries can be defined as:

Return all instances that fulfil one of the following queries.

The result of the union should be all triples that are in the result set of at least one of both queries. The union operator is similar to the logical OR.

6.1.7 Intersection queries

Intersection queries can be defined as:

Return all instances that fulfil all of the following queries.

The result of the intersection should be all triples that are in the result set of both queries. The intersection operator is similar to the logical AND.

6.1.8 Difference queries

Difference queries can be defined as:

Return all instances that fulfil query1, but not fulfil query2.

The result of the difference should be all triples that are in the result set of the first query, but do not appear in the result set of the second query. The difference operator is similar to the logical NOT.

6.2 Relaxing or strengthening the query

Because of the unfamiliarity of the users with the data they search and the inability of humans to formulate queries that precisely fit their information need, we want the SW search engine to be able to aid users by exploiting the RDFS structure. The algorithm requirements that are described in this section present some requirements of solutions for the problems described in

section 4.4.

If some query leads to no or too few results, it is desired that the system relaxes the query (somewhat) and looks for more results that may be found by using this relaxed query. Because RDFS contains a logical structure (e.g. class and property constructions) it is preferable for the system to exploit this structure to determine extra result candidates. The system should relax the constraint and continue searching until the desired number of results is found, some time constraint is reached or the expansion possibilities are exhausted.

A query may also appear to be too general, if e.g. it surpasses some maximum number of results. The system is then not able to decide how to strengthen the query, because the user query is probably just not specific enough (i.e. the query does not give hints for more specific interpretation). But looking at the RDFS structure it may try to make some query strengthening suggestions (like suitable subclasses for some query class terms). It would be preferable to make this maximum number of results a customisable variable, or make it relative to the amount of indexed data (because if the amount of data that is queried over is larger, the expected number of results is also larger). If we introduce the parameters "minimum number of results" and "maximum execution time" we want them both to be adaptable.

6.3 Structuring results

If some query execution leads to numerous results (e.g. thousands) it is very unlikely that the user wants to inspect all of them. Thus the problem is to how to support the user to interpret the results. The first concern is how to sort these results, so that the results that have the highest probability to be relevant to the user are on top. Another good possibility would be to look for similarities between results and use this to group the results based on these similarities.

7 Data structure

In this, and the following two, sections we present a theoretic framework for our system by presenting a data structure and the corresponding algorithms that together form a solution for the requirements. We call our system (data structure plus algorithms) SNEL, which stands for Search Nearly Everything Language.

We need to design SNEL so that it accommodates for the queries and algorithms as specified in section 6. As mentioned in section 6.1.1 an important type of queries we want to be able to evaluate are term queries. To be able to accommodate for this type of query we need to maintain an alphabetically sorted list of all occurring terms in all the input data, which we assume are RDF triples. Now we could attach to every term the triples in which it occurs. However, we want to accommodate for querying over every separate component of the triple. Therefore we decide not to maintain a list of triples for every term in which they occur, but rather to maintain a (sorted) list of all URI's/Literals in which they occur. To quickly retrieve all triples in which these URI's/Literals occur we maintain several lists of all triples, each sorted on one component of the triple. Note that here we choose to explicitly exchange storage overhead for a faster query evaluation.

To accommodate efficient evaluation of the structural queries we need to keep the two tree structures in accordance with the class and property structure in RDFS. To efficiently search in these tree structures we maintain an index on both structures, i.e. a sorted list of all classes/properties with accompanying references to their appearances in the tree. To accommodate for the class instance queries we maintain for every class and property, in the corresponding trees, an instance list. Note that by maintaining the property tree for all properties together with its instances pairs, the URI to triple list sorted on property is redundant.

The former considerations lead to the design as depicted Figure 7-1, which contains the data structure we constructed for SNEL. The sections 7.1 up to 7.5 contain more detailed descriptions on the various components of the design.

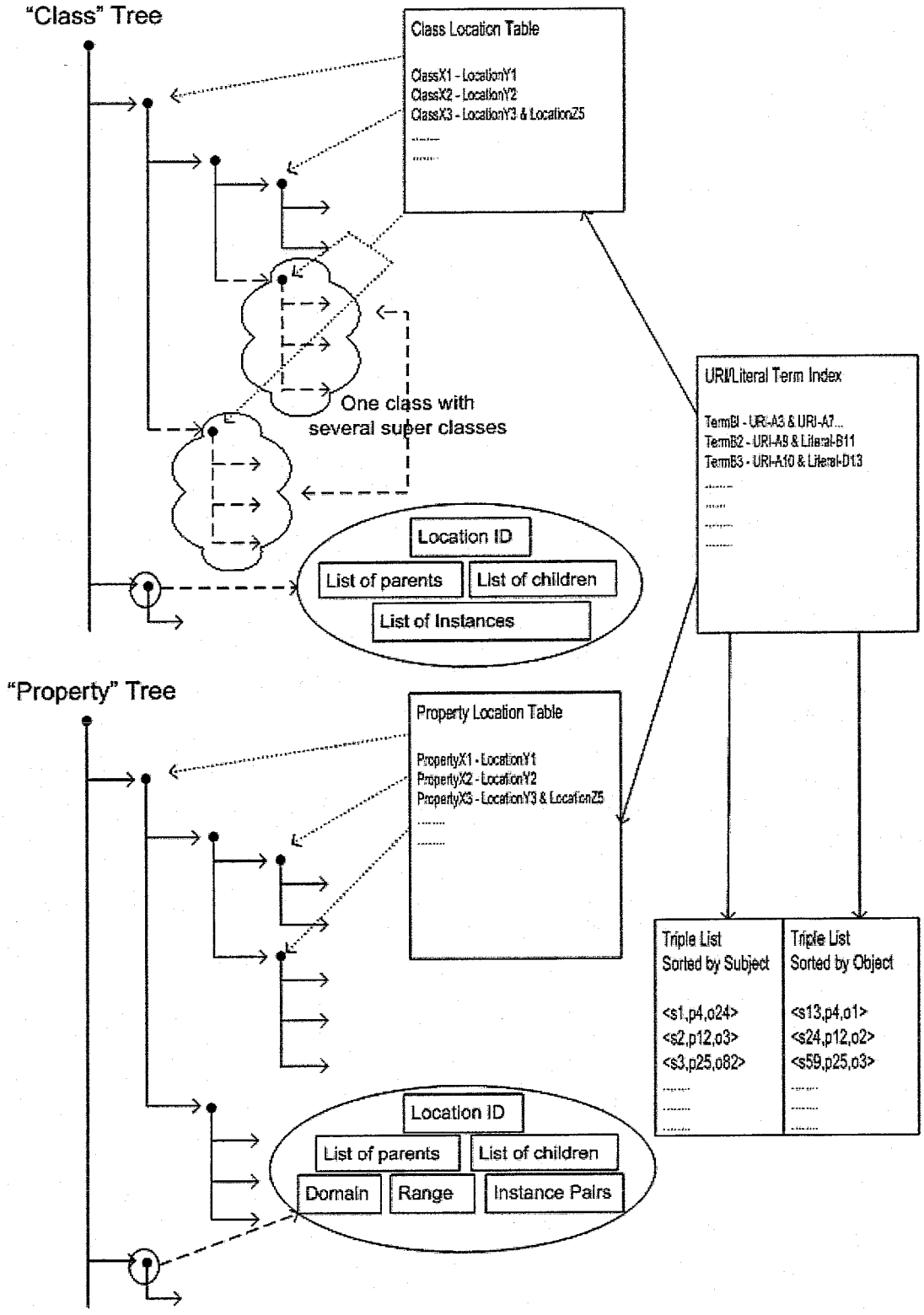

Figure 7-1: SW index data structure

7.1 Class and property tree

1035 The main structures in the index data structure are the "class tree" and "property tree". We define the terms of "parent" and "child" of a node in the usual way. The tree is based on the RDFS subclass and subproperty hierarchical structure. This hierarchical structure cannot directly be translated into a tree, because RDFS allows multiple inheritance. In spite of this a

class may neither be a subclass of itself nor a subclass of one of its own subclasses, i.e. the inheritance graph is cycle-free. Note that this restriction is removed from the latest RDFS specification [10], because of compatibility reasons with OWL. For this thesis we will assume the constraint, however. If, in the future, this would pose problems the data structure could be adapted to support for the absence of this restriction. The mentioned translation problem is solved by copying the node for every subclass or subproperty (child) declaration. Thus, if a class is declared as subclass of more than one superclass it is separately copied as a subclass for every of those superclasses.

There is an important implementation issue here that should be taken care of. One way to implement this tree translation would be to maintain one copy of the specific class and to insert links to that copy instead of duplicating the information. This implementation prevents update issues and space overhead. However, this method can bring up a speed issue if the search engine is implemented to run in parallel on several computers. Then query evaluation may have to repeatedly switch between several physical machines, because of the distribution of the data. This would greatly increase the communication overhead and thus slow down query evaluation. Another way to implement the tree translation would be to duplicate the node for every appearance in the tree. This would build-in some redundancy and thus introduce some extra space overhead. It would also introduce an update issue. However, it would eliminate the extra communication overhead in an implementation on a parallel network. The latter solution would probably be the most efficient for a Web sized repository, but note that, for this section, we can abstract from the actual choice for a specific implementation.

7.2 Tree nodes

Every node in both trees stores some information. For the class tree the following properties are recorded:

- Location ID
- List of parents
- List of children
- List of instances

The location ID structure should be chosen such, that knowing this ID enables the system to instantly retrieve the corresponding node of the tree. A possibility would be to choose the class / property identifiers themselves as an ID, but this is not necessary. If another approach is chosen, the names of the properties/classes should also be included in the node for identification. The list of a node's parents and children is maintained to enable traversing the tree, which will be used later to exploit the RDFS structure for evaluation of queries. Furthermore, for every node the list of direct instances (not including the instances of its children) is maintained so that the often-occurring operation of retrieving all the instances of a class can be executed quickly.

The information that is recorded for the nodes in the property tree is slightly different, namely:

- Location ID
- List of parents
- List of children
- Domain
- Range

- List of instance pairs

Location ID, list of parents and list of children function similarly to the equally named counterparts in the class nodes. The domain and range of a property are also stored and may be used to restrict a search to certain classes. Furthermore the property nodes contain a property called "list of instance pairs", which is the counterpart of the class nodes' "list of instances". A property in RDFS can be seen as a directed edge between two nodes. The property instance then consist of the two nodes. The instance pairs can be sorted in two ways; sorted on subject or object of the property. It basically does not really matter which sort method is chosen. If statistic query execution data shows that one of the two is searched over more often that order can be utilized. We consider maintaining two identical copies of the instance pair list: one sorted on subject and the other sorted on object. This would be an exchange of space overhead for faster query evaluation (for certain queries). For the rest of the document we will assume one instance pair list sorted on subject.

Note that every triple indexed by the system will appear in the property tree exactly once.

7.3 Class and property location table

To be able to find specific classes in the class tree a class location table is maintained that contains a sorted list of all classes, and for all those classes references to the position of the class in the class tree. Because classes can occur on several places in the class tree there may be more than one position-coordinate for one class.

Similar to the class location table for the classes in the hierarchy, a property location table is maintained for properties. This property location table is structured similarly to the class location table.

Note that classes are of type URI or blank node and properties are of type URI.

7.4 URI/literal-term index

Users do not know the exact class and property names, or more generally the specific syntax in triples, they search (which are typically URIs). They typically only want to have to define some keywords that occur in the triples. To accommodate for this an URI/literal term index is maintained. For this index URIs and literals are broken down into separate terms and are sorted by term. Every term in the index is accompanied by a list of URIs and Literals that contain the term. The process of breaking down URI's and literals in individual terms is left as a separate concern.

The alternative for breaking down the URIs and literals, but to still provide for term search, would be to store complete URIs (or literals). Then, instead of a binary search for some term do a pattern search in all data. Although this last process can be executed concurrently for different parts of the class term index (if it would be divided over several physical machines) this will still lead to a substantial performance downgrade (from order O(Log(||index||)) to order O (||index||)).

7.5 Triple lists

A URI or literal can occur in several triples and URIs also may represent a property or a class. To locate in which triples some URI or literal occurs and if some URI is a class or property four tables can be consulted. To determine if a URI is a class or a property the class and

property location tables can be used. To generally look if some URI occurs as the subject or object of some triple (for properties the property location table is used) two triple lists are maintained. They are both equal in content but one is sorted on subject (and object is the second sort key) and one is sorted on object (and subject is the second sort key). Literals only have to be looked up in the triple list sorted by object because they can only be used as the object part of a triple.

Note that we have not treated the notion of blank node in our design. W3C mentions in [17]: "A convention used by some linear representations of an RDF graph to allow several statements to reference the same unidentified resource is to use a blank node identifier, which is a local identifier that can be distinguished from all URIs and literals." We will apply the same method. A blank node is given an identifier and is further treated similar to an URI. Note that these two groups are disjoint. Because blank nodes are not really a relevant factor for our project we will not consider them any further.

8 Query solving algorithms

8.1 Introduction

In section 6 we introduced the types of queries we wanted to be able to solve with our system. In section 7 we introduced the data structure for our system, SNEL, of which we claimed it enables us to solve these queries. In this section we describe the algorithms that use the data type described in section 7 to solve the queries mentioned in section 6. In section 12.2 we will discuss the efficiency of the algorithms presented here.

8.2 Term queries

Term queries are queries of the type:

Return all resources that contain term 'X'.

The informal algorithm to solve this query:

1. *Search the URI/Literal term index for 'X' and return the corresponding URIs/literals.*
2. *Search the URIs/Literals found in 1 in the following tables (may be executed concurrently):*
    1. *In the triple list sorted by subject*
        1. *Search for the URIs in the Subject with a binary search and return results*
    2. *In the triple list sorted by object*
        1. *Search for the URIs in the object with a binary search and return results*
    3. *In the Property Location Table*
        1. *Search for the URIs as property with a binary search and return the resulting locations in "location list"*
        2. *For every location in "location list" construct triples from its instance pairs combined with the corresponding property and return these triples*
3. *Return all the results found in step 2 to the user.*

8.3 Structure queries

We considered the following five structure queries:

1. Return all classes that contain term 'Y'
2. Return all classes that are a child of class 'X'
3. Return all classes that belong to the sub-tree of class 'X'
4. Return all classes that are a parent of class 'X'
5. Return all classes that are ancestor of class 'X'

The informal algorithm to solve query 1:

1. *Search the URI/Literal term index for 'Y' and return the corresponding URIs/Literals.*
2. *Search the URIs/Literals found in 1 in the class location table and return the URIs that occur in that table as results of the query to the user.*

The informal algorithm to solve query 2:

1. *Search the URI 'X' in the class location table and return a location ID of the class.*
2. *Look up the ID in the "class" tree and return a list of children that result from the query to the user.*

The informal algorithm to solve query 3:

1. *Search the URI 'X' in the class location table and return a location ID of the class.*
2. *Look up the ID in the "class" tree and return the list of children.*
3. *Create a result list and a todo list and put the result of 2 in both lists.*
4. *Select a class from the todo list. Return all its children and add these children to the todo list and the results list. Remove the selected class from the todo list.*
5. *Repeat 4 until the todo list is empty.*

The informal algorithm to solve query 4:

1. *Search the URI 'X' in the class location table and return a location ID of the class.*
2. *Look up the ID in the "class" tree and return the list of parents as result of the query to the user.*

The informal algorithm to solve query 5:

1. *Search the URI 'X' in the class location table and return a location ID of the class.*
2. *Look up the ID in the "Class" tree and return the list of parents.*
3. *Create a result list and a todo list and put the result of step 2 in both lists.*
4. *Select a class from the todo list. Return all its parents and add these parents to the todo list and the results list. Remove the selected class from the todo list.*
5. *Repeat step 4 until the todo list is empty.*

8.4 Class queries

For class queries we considered two types of queries, namely:

1. Return all instances of class 'X'.
2. Return all instance of classes 'X' that contain the term 'Y'.

The informal algorithm to solve query 1:

1. *Search the URI 'X' in the class location table and return a location ID of the class.*
2. *Look up the ID in the "Class" tree and return the list of instances as result of the query to the user.*

The informal algorithm to solve query 2:

1. *Search the URI/Literal term index for 'Y' and return the corresponding URIs/Literals.*
2. *Search the URIs/Literals found in step 1 in the class location table and return the URIs that occur in that table in a list called 'X'..*
3. *Select a class from 'X', search its URI in the class location table, return a location ID of the class and remove the class from the 'X'.*
4. *Look up the ID in the "class" tree and return the list of instances in a list called "results".*
5. *Repeat step 3 and step 4 until 'X' is empty. Then return "results" as result of the query to the user*

8.5 Property queries

For property queries we consider the following type of queries:

Return all resources that fulfil the pattern: <S | ? , p | ? , O | ?>

The informal algorithm to solve this query:

1. *In case p is known:*
   1. *Search the URI 'X' in the property location table and return a location ID of the class.*
   2. *Look up the ID in the "property" tree and return the list of instance pair as a list called "results".*
   3. *In case S is known*
      1. *Search for the elements in "results" that have the value of S in the subject and delete the rest of "results". (Binary search if instance pairs are sorted on subject, linear search otherwise)*
   4. *In case O is known*
      1. *Search for the elements in "results" that have the value of O in the object and delete the rest of "results". (Binary search if instance pairs are sorted on object, linear search otherwise)*
   5. *Return "results" as results of the query to the user*
2. *In case p is variable:*
   1. *In case S is known*
      1. *Search for S in triple list sorted by subject and return the resulting triples in a list called "results"*
      2. *In case O is known*
         1. *Delete all triples in "results" that don't have O as object*
      3. *Return "results" as results of the query to the user*
   2. *In case O is known and S is variable*
      1. *Search for O in triple list sorted by object and return the resulting triples in a list called "results"*
      2. *Return "results" as results of the query to the user*
   3. *In case O and S are variable*
      1. *Return the triple list sorted by subject as the results of the query to the user*

8.6 Path queries

For path queries we consider the following type of queries:

$Path_0 :: <X_0 \mid ?, [d_0] \mid p_0 \mid ?, Path_1>$

$Path_i :: X_i \mid ?, [d_i] \mid p_i \mid ?, Path_{i+1} \quad \{ 0 < i < n , d \in \mathbb{N} \}$

$Path_i :: X_n \mid ? \ , \{n>0\}$

The algorithm to solve this query starts with expanding the query through substitution of the values d_j as follows:

1. *We have a collection of queries, which we call 'query-collection', that initially only contains the input query.*
2. *For every d_j in the query with some value 'm' we create m times the number of queries in 'query-collection', where every query in 'query-collection' d_j is replaced by '?,?^{n}', for all n between 1 and m, $1 \leq n \leq m$*

An example of this algorithm is substitution of some distance operator d_a with value '3'. This would create three queries: one with d_a substituted by '?,?', one with d_a substituted by '?,?,?,?' and one with d_a substituted by '?,?,?,?,?,?'

We now have a collection of queries that are of the form: $<X_0|?,\ldots,p_i|?,X_i|?,p_{i+1}|?,\ldots,X_n>$. We solve this query collection as follows:

1. *Evaluate every query in the collection (may be done concurrently)*
2. *Take a union of the results of all those queries (see 8.7 for an algorithm for union of result sets).*

For evaluation of one query we use the following informal algorithm:

1. *A query always has a number of 2n+1 constants/variables and can be decomposed into n property queries parts.*
2. *Every property query part is numbered according to its occurrence in the query (from left to right).*
3. *Evaluate the property query parts (according to algorithm in 8.5). This may be done concurrently.*
4. *Next we combine these results as follows:*
    1. *Construct a list called "results", which initially contains the results of the first property query part. Mark the first query part as processed.*
    2. *Take the results of the query part with the lowest number, which is not yet marked as "processed" and call it QP.*
    3. *For every result R in the "results" list:*
        1. *Take the object of the last triple of R*
        2. *Search for R in the subject of all triples in QP*
        3. *The number of triples that have that subject match is called "a"*
        4. *For every match in QP the "results" list is extended with R appended with the matching triple (which will lead to in total "a" times R new results).*
        5. *The original R is deleted from the "results" list (thus if there are no matches in QP the "results" list becomes smaller).*
    4. *Mark the query part that corresponds to QP as processed.*
    5. *Repeat step 2 to 4 until every query part is processed.*
5. *Return "results" as the results of the query to the user.*

To perform the given algorithm as efficiently as possible the instance pairs in the property tree should be sorted on subject. However, if it would be better to sort the instance pairs in the property-tree on object the algorithm can easily be built to evaluate from right to left instead of from left to right.

8.7 Union queries

For union queries we consider the following type of queries:

Return all instances that fulfil one of the following two queries.

The two queries are expected to return triples (instances) as results. The informal algorithm for the union then is:

1. *Construct a result set of triples, called "results", that is initially empty.*
2. *For all triples in both result sets do:*
    1. *(Binary) Search to see if the triple is in "results"*
    2. *If not insert the triple in results (at the right place).*
3. *Return "results" as the results of the query to the user.*

Note that the result of this union is again a set of triples.

8.8 Intersection queries

For intersection queries we consider the following type of queries:

Return all instances that fulfil both of the following two queries.

The two queries are expected to return triples (instances) as results. The informal algorithm for the intersection then is:

1. *Construct a result set of triples, called "results", that is initially empty.*
2. *For all triples in the first result set do:*
   1. *Search to see if the triple is in the second result set. If this is true then:*
      1. *(Binary) Search to see if the triple is in "results"*
      2. *If not insert the triple in results (at the right place).*
3. *Return "results" as the results of the query to the user.*

Note that the result of this intersection is again a set of triples.

8.9 Difference queries

For difference queries we consider the following type of queries:

Return all instances that fulfil query1, but not fulfil query2.

The two queries are expected to return triples (instances) as results. The informal algorithm for the difference then is:

1. *Construct a result set of triples, called "results", that is initially filled with the results of query1.*
2. *For all triples in the result set of query2 do:*
   1. *(Binary) Search to see if the triple is in "results"*
   2. *If this is so delete the triple from "results".*
3. *Return "results" as the results of the query to the user.*

Note that the result of this difference is again a set of triples.

9 Result manipulation algorithms

In section 6 we introduced, besides queries, a number of query expansion and query strengthening methods, which we want to have in our system (SNEL). In this section we describe the algorithms that use the data type described in section 7 to execute those methods. In section 12.2 we will discuss the efficiency of the expansion algorithms presented here.

9.1 Relaxing the query

When a predefined minimum number of results is not found upon evaluation of the query, or the user explicitly indicates that the number of results is too limited, the query may be relaxed. The idea behind relaxation is to find extra candidates that do not fulfil the query, but are for some reason likely to be relevant to the query. The goal for the relaxation is to help the users to find additional results that they might be interested in, but which are not found as an exact result for the query, because the query was too strict.

We will discuss some methods to do that in this section. In paragraph 9.1.1 we discuss the "regular" methods. With "regular methods" we mean methods that only concentrate solely on the query terms without exploiting the RDFS structure. From paragraph 9.1.2 and on we discuss methods to relax the query exploiting the RDFS structure.

9.1.1 Regular methods

There are already methods available to relax queries in the state of the art search engines. One of these methods is to look for small syntactic variations of terms posed in the query. Consider a class query for some class that contains the term "aabbccdd" (but it works equally for terms in properties or instances). Now suppose that searching for the term "aabbccd" in the class term index returns no or only a few results. Furthermore suppose there is a term "aabbcdd" in the class term index that has lots of resulting classes. This term "aabbcdd" only differs one letter from "aabbccd" so maybe the user misspelled the term. So if there are syntactical variations for some term these may be included in the search. Algorithms for measuring syntactic closeness are publicly available; think for instance of spell-check systems in word processors (and especially the correction suggestions).

A common method of query relaxation involves the normalization of terms. This means that both indexed terms and query terms are brought down to a normal form. Examples of normalisation are stemming of verbs ("walking" and "walks" are both stemmed to "walk" for instance) and the removal of plural forms. Thus declensions are taken into account during searches so that obvious candidates that fulfil the query conditions are not accidentally missed.

Besides taking small syntactic and semantic variations into account it would also be a good idea to look at synonyms of terms. People may namely use different terms when they actually refer to the same thing. This could also be taken into account during the search (unless off course it is explicitly stated by the user not to do so). So, for example, if somebody searches for the term "picture" also terms as "image" or "photo" should be found. To support this, the system could use a synonym dictionary.

9.1.2 Subclass and subproperty expansion

In this and the following paragraphs we focus on RDFS exploitation for query expansion. If a query contains some class or property expression the RDFS subclass and RDFS subproperty trees are good candidates to utilize.

Subclasses of some class and subproperties of some property define a special case of the specific class or property. This relation between classes and subclasses (similar for properties) is illustrated by the following definition of W3C ([10]): "*If a class C is a subclass of a class C', then all instances of C will also be instances of C'.*"
In our architecture we chose to store only a class' direct instances. Thus the classes of the class' subtree are not computed in advance and included in the class' instances. This is done so that a class' direct instances can be distinguished from its inherited instances. To include the instances of a class' subtree we traverse through the tree. The algorithm for including the instances of the subclasses in class X is an adaptation of the subtree structure query. It becomes as follows:

1. *Create an instances set and a todo set and put the cildren of X initially in the todo set and keep the instances set initially empty.*
2. *Select a class from the todo set. Add all its instances to the instances set and add all its children to the todo set. Remove the selected class from the todo set.*
3. *Repeat step 2 until the todo set is empty (or some execution time limit is reached).*

This algorithm is similar for the property tree.

Because of the stated definition of W3C (*If a class C is a subclass of a class C', then all instances of C will also be instances of C.*) subclass and subproperty expansion is a good and safe way to expand the query results.

9.1.3 Superclass expansion

If the subclass and subproperty expansion method still results in too few results a generalisation of the class and property query parts may be executed by examining its superclass(es). For instance, consider a search for images on animals of the class "painting". One may then also be interested in images of the class "art", which is for instance the superclass of "painting". And next, if there are still not enough results, also the children of the superclass, in this case "art", may be included in the search, such that classes as "photograph", "drawing", etcetera are also included (depending on the specific class structure).

The superclass / subclass examining algorithm is very similar to the algorithm noted in the previous paragraph and will not be repeated here.

Superclass expansion will return extra results that are in someway related to results that do precisely fulfil the query. However, the superclass may be a too large abstraction of its subclass and then produce some unwanted too general solutions. Therefore the superclass expansion method will expand the query results with possible valuable candidates, but the value of these additions is more uncertain than those of the subclass / subproperty expansion method.

9.1.4 Instance membership expansion

Another form of query relaxation we introduce here is the utilization of instance memberships. This part of query expansion is only applicable for classes (not for properties, because properties have instance pairs, which are unlikely to occur in the identical subject-object composition for another property). If we find a few, but not enough, instances that fulfil some query conditions we may analyse the results to try to find new results candidates. We try to utilize the possibility that some resource may be the instance of several classes at the same time.

There are cases where this might add valuable results. For instance if the query is about the class "actors" that fulfil some conditions and some instances that are found are also instance of the class "director". We then may add a search for resources of the class "director" that fulfil the other conditions as a query expansion. Note that this method is highly speculative. Consider for instance the same query about the class "actors" of which an instance is also instance of the class "American president". If we now add a search for resources of the class "American president" that fulfil the other conditions this may deliver a number of unwanted results.

We do this with using the following algorithm:

1. Take some instance 'a' that is instance of class 'X'.
2. Look up 'a' in the triple list sorted by subject.
3. Search the resulting triples for type definitions.
4. If type definitions are found that are not already known do the following for all new classes:
    1. Look up the object of the new type definition in the class location table
    2. Return the list of instances of the found class as alternatives for 'a'
    3. Optionally return the instances of the subtree as extra alternatives for 'a'.
5. Repeat 1-4 for some other instance, or stop if there are no other instances, the number of results is satisfactorily or some execution time limit is reached.

Instance membership expansion will return extra results that might be related to the results that precisely fulfil the query. This method is however more speculative than superclass expansion method. In many cases this search does not have to come up with desired results and should therefore be used as the last query expansion option after the subclass / subproperty and the superclass method.

9.2 Query refinement

In many cases the user's query is just not specific enough to find the results that the user actually expects. The query should then be refined to express what the user really wants; this cannot be decided by the system itself (it doesn't know what the user wants). The system can however return some suggestions for refinement.

If some class is too general, i.e. has too many instances (including its subtree instances), the query engine may return its children as possible replacement for the concerning class. These are good suggestions because the subclass relation of a class implies it is more specialised than its parent. If, for instance, one searches for a "vehicle" with some properties and this returns too many results, the search engine could suggest to restrict to "sports car" if it is a subclass of "vehicle". The former works similar for properties.

If some term in the query is polysemous (word with several meanings), the system may suggest some terms that indicate the intended use of the concerning word. An example of a polysemous word is "bug", which may indicate a beetle, a programming error or the verb "to annoy". Searching with the term "bug" may lead to results that use the term in any of these contexts. So for "bug" additional query terms like "pesticide", "language" or "irritating" may be used to select the appropriate context.

9.3 Structuring results

In this section we discuss two subjects related to structuring results. In paragraph 9.3.1 we discuss the sorting of query results. We discuss how to rank them, the techniques that are often used, and how to involve the Semantic Web. In paragraph 9.3.2 we discuss the grouping of results. Grouping is similar to sorting results, as it is ranking with a different granularity. We will discuss some criteria that could be taken into account for grouping.

9.3.1 Sorting query results

If the query delivers many results something must be done to present those numerous results to the user. An important issue is sorting the results. The first results must be of the best quality as these are the results that the user is most likely to inspect.

There currently are basically two types of algorithms for sorting query results on Web pages. The first type is to judge the importance of some document for a certain query by analysing quantity and position of the query terms in the candidate document results. This type takes heuristics into account like the number of occurrences of query terms and the positioning of those terms. The second type is using the link structure to assign importance values to pages. As mentioned earlier in section 3.2, the main idea of this type is that a link from one page X to a page Y means that page X thinks page Y is important in some way. The importance of some page is then measured by looking at the number of important incoming links it has. Besides the two mentioned types there are also some less used types. The sorting can for instance also be fine-tuned by counting how many people will inspect some resource if it is returned in the top ranked pages. If this is almost never the case, the system may conclude that the document is possibly too highly ranked and this may be corrected. Also other user statistics could be used in the same manner.

The sorting problem is a complex problem and we will discuss it only briefly here. The system we are targeting should use a ranking system that uses a combination of the techniques discussed before. Semantic Web pages are full of references to other Semantic Web pages. These references could be used similar as the links in Web pages. We could measure importance of, say class definitions (instead of e.g. complete SW files), by the number of times they are referred to by triples of important resources. References in Semantic Web pages often refer to a part of a page and not only to whole pages as is often the case with HTML pages. In this way the importance of part of pages can be computed. Furthermore if some important SW page (or part of a page) claims to be an instance of some class it may receive a special importance for some specific term (instead of a global importance only). In this way a term can have several importance values for different terms. An example for this is if you have some SW annotated page about some well known painter and it is mentioned on the page that one of the hobbies of the painter is "drinking beer", then the page may be an authority for the term painting because it has important "paint" references, but it is less important for "beer", because not many important incoming links are "beer" references.

For path queries the distance operator could be involved for ranking. Results with a lower distance can be ranked higher (because the relation between nodes is higher) than results with a higher distance.

The ranking methods described above are based on query results that are exact matches for the query. In earlier sections we also described finding solutions that are not exactly conformed to the query but are estimates that almost fulfil the query. Because estimates are not as good results as exact results, we will rank estimate results lower than exact results: the "closer" the

estimate, the higher the ranking. For example if extra results are found through the superclass expansion method these are ranked higher that the extra results that are found through the instance membership expansion method.

9.3.2 Grouping results

Another (additional) possibility, besides ranking, is to group results so that results that are somehow related to each other are collected together in one group. The question is what the selection criteria should be for a result to belong to one group instead of another.

What the final grouping method we will use in our system will look like will be kept for later concern and not be treated in this thesis. Here we will mention some criteria that could be taken into account for grouping.

A good candidate for grouping would be to utilize class and property instances. We could group these instances on the class or property they belong to. If a user decides that some class (similar for properties) is not what he was looking for, he may disregard the entire class. In this way sub-groups may be formed as well. For instance if the result set was expanded by the subclass / subproperty expansion method or the superclass expansion method on some class, these extra results might be visualized as offspring of that class. A possible example of how to implement this is the blended browsing and querying in EROS ([16]).

Not all resources have to be defined as being instances of classes or properties. How do we group those results? A possibility is to look at special keyword combinations. A special keyword in some document is a term that is relatively rare in the complete collection of all indexed resource, but appears relative frequently in the specific document. In query results there are normally subsets that are about a corresponding subject. Collections of documents that handle a specific subject often have a set of special keywords in common. A collection of document with a corresponding set of special keywords may be grouped together.

The last grouping criterion mentioned here is grouping on physical location. Normally if some website domain handles some subject at least a part of its sub domain also treats the same subject. Therefore a search may produce a relative large number of results from one domain. These results can then be grouped together based on the domain they belong to.

A remaining problem is the sort order in which to return the groups (including a possible rest group). We somehow want to include sorting to decide which group to show fist, so that highly ranked result are not in the group that is visualized the last (which the users may not be willing to inspect anymore). We could return groups on basis of the highest ranked result only. But this could lead to unwanted results like some group with a lot of high ranked results being placed after a group with one very highly ranked results and further only very low-ranked results. Suppose ranking values vary between lower bound Y and upper bound X. We could define some threshold between Y and X above which results are considered to be high ranked. Then we could sum all high ranked results per group and assign this value as the group ranking. Then the groups could be sorted on this group ranking.

10 Implementation in Sesame

In the previous sections we showed our ideas for implementing SNEL as proposed with a data structure (section 7) and its corresponding query evaluation algorithms (section 8). For our implementation, however, we chose to implement SNEL's query-part on top of an existing RDF(s) database system, namely Sesame ([18]). We chose to implement our system on an existing system because of two reasons. The first reason is the time constraint, of about 9 months, for this project is not sufficient to implement a complete system. The second reason is we are interested in a case study using an off the shelf engine, so we can analyse its performance in comparison with our design. In section 12 we look at the feasibility of this solution by making the comparison between the theoretic framework and the actual implementation on top of Sesame.

10.1 Sesame's architecture

Sesame is a system that allows storage and querying of RDF(s) data.

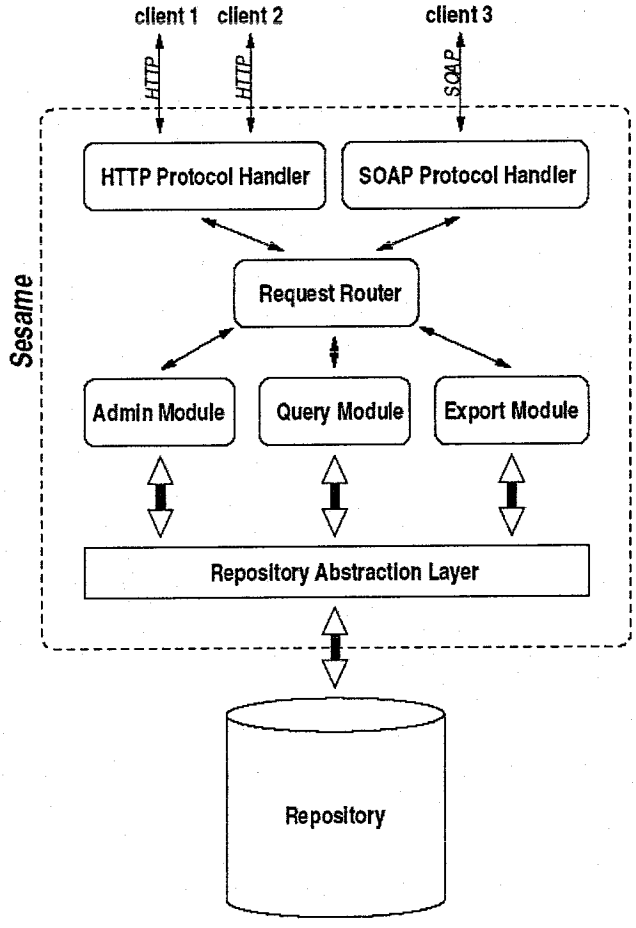

Figure 10-1: Sesame's architecture

Figure 10-1 ([18]) is an overview of Sesame's architecture. The first thing one probably notices is the modularity of the design. Sesame is independent of repository. This means that different storage modules (e.g. databases) may be used, as long as translation of operation primitives that are used by Sesame (protocol handler) are integrated in the Repository Abstraction Layer. Current storage modules that can be handled are the PostgreSQL, MySQL and Oracle 9i databases and the in-memory module implemented in Sesame itself.

RDF is defined as a model and is actually independent of its chosen encoding, even if this is usually XML encoding. Therefore, to allow for different encodings Sesame stores the data on a structure level, i.e. it stores triples instead of XML data. This way other encodings can be allowed by adding compilers for the specific encodings. Current allowed encodings are XML, N3 and N-triples. These encodings can also be used as return format for queries (besides the special query results return format in HTML tables).

The query module let the user query the data in the repository. Again things are modular here. A variety of query languages may be implemented and added to the system. The current available query languages are RQL, RDQL and Sesame's own SeRQL.

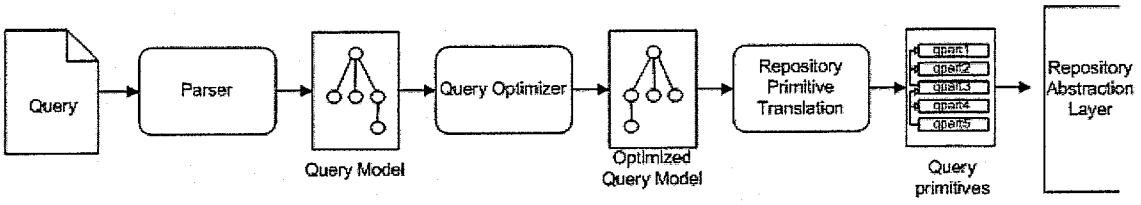

Figure 10-2: Sesame query evaluation

Figure 10-2 is an overview of how query evaluation in Sesame works. RDF can be interpreted as a graph model. To query this graph model Sesame parses the query, stated in its particular query language, and translates this in a partial graph model, the so-called query model. This partial graph model can be used as a pattern, which must be matched within the graph model of the stored data in the repository. Before matching the graph model is first optimized and transformed into an equivalent model that can be evaluated faster than the original query model. These optimizations mainly consist of a set of heuristics for query subclause move-around ([18]). Then, this optimized query model is matched with the graph model in the repository.

Sesame is repository independent. This means that some repository may store the RDF data model as it wishes as long as it provides an interface for certain query primitives. The query model is then translated into a set of those query primitives. The query results that are provided by the repository are then combined by Sesame's query engine. Sesame was designed this way (instead of trusting the repository for solving the entire query) to keep the dependence on the repository as low as possible.

The last aspect of Sesame that we will discuss here is its API. Sesame's API is also designed modular. For some protocol to integrate in Sesame, a handler for that protocol has to be integrated in Sesame's request router. In this way Sesame is accessible from different protocols. The only protocol handler that is implemented currently is the HTTP protocol. However, a SOAP protocol handler is in the making.

10.2 Sesame's Web application

Figure 10-3 is a screenshot of Sesame's Web application that ships with their server. This Web application is built such that it utilizes all of Sesame's functionality. The login screen (no screenshot included) lets the user select a repository to work with and optionally log-in if this is necessary for the selected repository. After that the GUI as shown in Figure 10-3 is displayed. The user is shown the permitted action on the right of the menu at the top of the screen. Users may have two different rights: Read rights and Modify rights. These rights may

also be defined for anonymous users. If a user has read rights it can select a query language in which it wants to query the data in the selected repository or it can select "Extract" to extract all data or "Explore" to browse through the data. If a user has modify rights some modifying actions can be undertaken. These modifying actions are administrative actions of adding and removing RDF(s) data to the repository.

For Sesame's SeRQL query language two interfaces have been created: one for HTML output, the so called SeRQL select queries, and one for RDF output, which are called SeRQL construct queries. This project will only deal with the latter, because construct queries returns results in RDF format. This for instance facilitates querying the results of a query or combining several query results.

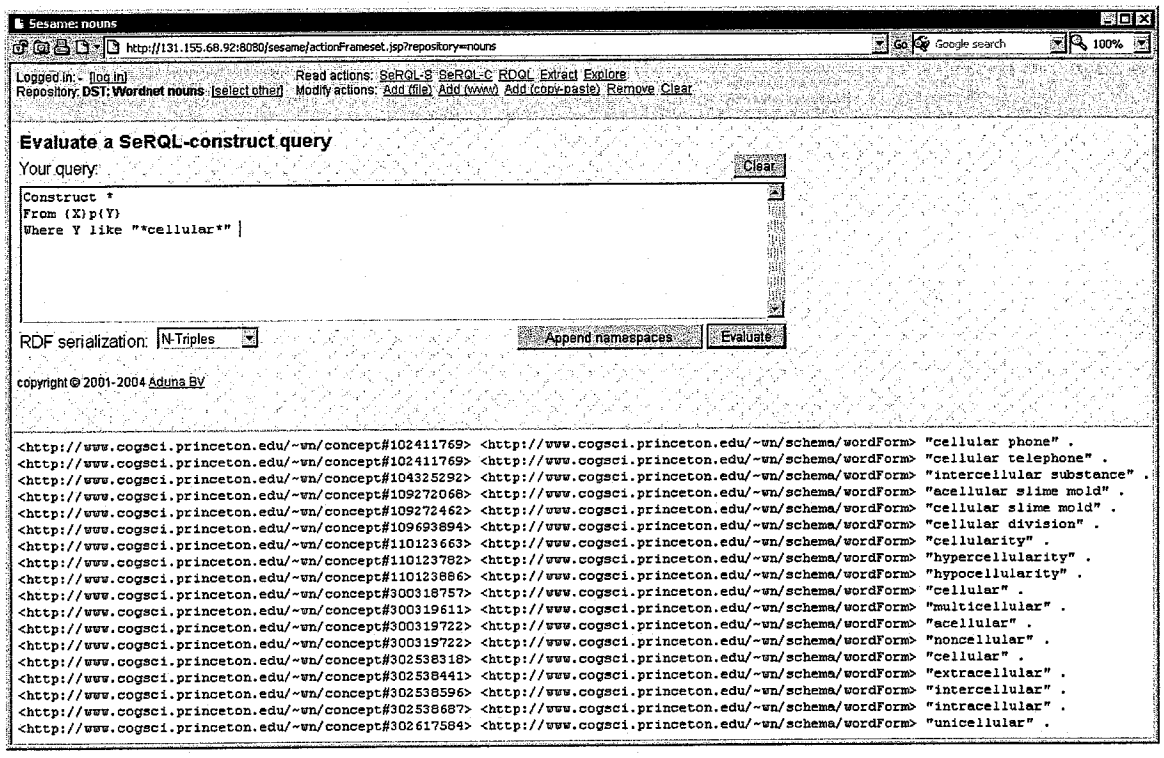

Figure 10-3: Screenshot of Sesame's Web application

For more information on Sesame, Sesame's Web application, or Sesame's query language SeRQL refer to [18], [19] and http://www.openrdf.org.

10.3 SNEL

We implement our system of queries in Sesame as a new query language. We name this system SNEL, just like the theoretic framework, but will refer to it as the SNEL implementation in Sesame. To implement SNEL we modify and add functionality to Sesame's Server and the Web application.

10.3.1 Environment

The Sesame server is implemented as a Java servlet application. Sesame should work on any Java servlet container that supports Servlet 2.2 and JSP 1.1 specifications, but is specifically tested on Apache's Tomcat 3.x or higher.

The used development environment is:

Operating system	Windows XP Professional
Servlet container / Web server	Jakarta Tomcat 5.0.18
RDF query engine	Sesame 1.0
Database	MySQL 4.0.17
Compiler	Java 2 SDK, version 1.4.2
Code Editor	Textpad 4.7 JBuilder Enterprise 9.0
Web browser	Opera 7.21 Internet Explorer 6.0

10.3.2 Architecture

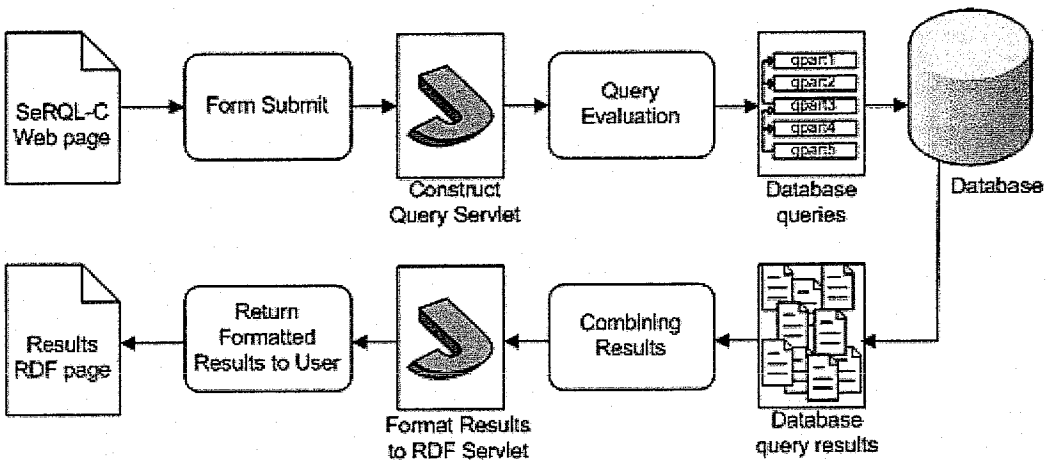

Figure 10-4: Current query evaluation schema

Figure 10-4 is an overview of the SeRQL-C query evaluation schema. It shows the steps that are taken from posting the query until showing the query results to the user. The SNEL query engine should do evaluation roughly in the same way. We observe that some query constructs of SNEL can be transformed into an equivalent SeRQL query. Furthermore the other query constructs of SNEL can be translated into a number of SeRQL query parts of which the results should be combined in some way. Therefore, instead of building a query engine from scratch, we build SNEL on top of the SeRQL-C query language. We will translate the easy SNEL constructs to an equivalent SeRQL-C query and then send that SeRQL-C to the "Construct Query Servlet" for further evaluation. For the more complex SNEL constructs we construct a number of SeRQL-C constructs and intercept the result of that query. If all SeRQL-C queries are then evaluated the results are combined in the applicable way and then returned to the user. Figure 10-5 gives an outline of the proposed query schema.

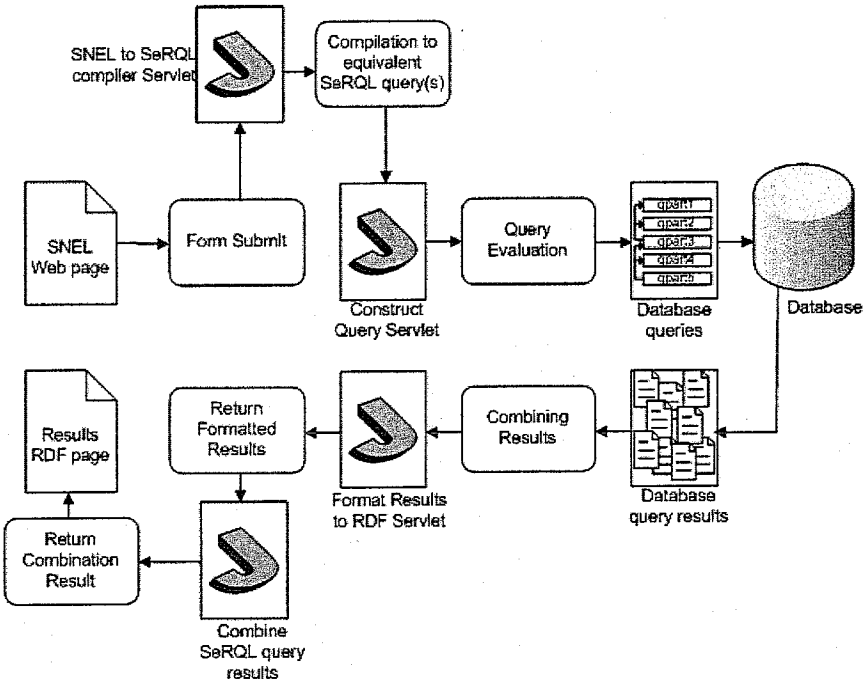

Figure 10-5: SNEL query evaluation schema

10.3.3 Compiler

The compiler that translates SNEL queries is generated by a compiler generation package called ANTLR (for more information see [20]). "ANTRL is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C++, or C# actions". In order to generate a compiler ANTLR takes three types of grammars as input, namely for the parser, lexer and tree-parser. Those grammars should be defined in an extended BNF (Backus Naur Form) notation. This eBNF notation should define the syntax of the language that is to be compiled and is used to generate a parser (or lexer) for the defined language. If some language construct is recognized by the parser, Java statements may be added to execute some actions. These actions could for instance be creating a translation based on the language construct that was found.

Grammar rules are translated into procedures during construction of the compiler. For these procedures input and output parameters may be defined. Furthermore, a grammar rule may contain references to some other grammar rules. The Java statements that are imbedded in the grammar may also refer to the language constructs they are associated with.

```
classq [int depth] returns[String out]
            {
              // Class queries. Syntax: Class[Z]. Z should be of type class. This
              // query will then return the instances of Z (thus: <?,rdf:type,Z>).
              // Instead of Z also the Term construct may be used.
              out=new String(); String nextterm=new String();
            }
      : QCLASS {out+="Construct DISTINCT *\r\nFrom {X}<rdf:type>";}
          ((nextterm=term {out+="{<"+nextterm+">}";}
            |nextterm=url  {out+="{<!"+nextterm+">}";}
            )
            |nextterm=regularq[depth+1,'3',""] {out+="{Y}\r\nWhere "+nextterm;}
          )
        RSQBR
      ;
```

Figure 10-6: Grammar fragment for compiling class queries

Figure 10-6 is an example of a grammar for class queries. The "class query" recognizer "classq" is called with an input parameter of type int (why is not relevant here) and it returns a string. The capital letter word QCLASS refers to detection of the class keyword together with a left square bracket. If a class operator is encountered, already a SeRQL query part can be generated (the out+="…" statement). Next, the term that is enclosed by the class operator is parsed with help of other parser rules, namely the "term", "url" and "regularq" parser rules that refer to a abbreviated URI, a complete URL and a term respectively (their grammar is left out here). The class query syntax should be ended with a RSQBR (right square bracket). This recognizer translates the SNEL query

Class[value]

into:

*Construct DISTINCT **
From {X}<rdf:type>{<value>}

Or:

Class[Term[value]]

To:

*Construct DISTINCT **
From {X}<rdf:type>{Y}
*Where Y like "*value*"*

SNEL Construct	Equivalent SeRQL-C Construct
Term[value]	Construct DISTINCT * From {X}p{Y} Where (X like "*value*") OR (p like "*value*") OR (Y like "*value*")
Structure[class,child, value]	Construct DISTINCT * From {X}<serql:directSubClassOf>{Y} Where Y = value
Class[value]	Construct DISTINCT * From {X}<rdf:type>{<value>}
Property[?,value1,value2]	Construct DISTINCT * From {X}p{Y} Where (p = value1) AND (Y = value 2)
Path[value1,[2],value2,?,value3]	Construct DISTINCT * From {<value1>} P0 {<value2>} P2 {<value3>} ∪ Construct DISTINCT * From {<value1>} P0 {X0} P1 {<value2>} P2 {<value3>}

Table 10-1: Some typical translations from SNEL to SeRQL-C

For some typical translations of SNEL constructs to equivalent SeRQL-C constructs see Table 10-1. For a description of the complete SNEL syntax see appendix B. Note that the SNEL Path construct is translated into (in this case) two SeRQL-C queries, of which the results should be united later on. Union, Intersection and Difference queries in SNEL are translated by translating their subqueries and later combining the results (thus there is no direct translation from those SNEL constructs to a SeRQL-C construct).

10.3.4 Combining the results

If a translation of some SNEL query produces more than one SeRQL-C query the results of those queries should be combined in some way. This is the case for SNELs Path, Union, Intersection and Difference queries.

This combination of query results is implemented with the help of Sesame primitives (as opposed to implementing the union, intersection and difference operators for arbitrary RDF documents). The implementation uses a temporary Sesame repository (can be a bottleneck, depending on the number of results of the subqueries). Sesame offers basically two modifications actions on its repositories: adding and deleting triples. Furthermore, Sesame only stores unique triples, and deletion of a triple that is not in the repository leads to a skip.

Uniting a number of query results is then done as follows: add all triples of all the results (note that results of SeRQL-C queries are by definition triples) to the temporary repository. After that extracting the entire temporary repository gives the result of the union. A difference query on two queries is implemented as follows: add all triples of the results of the first query to the temporary repository and delete all triples of the second query. Then extract the contents of the temporary repository, which is the result of the difference query. The intersection query is a bit harder. This query can be defined in terms of a union and some difference queries. If we denote a union between two sets by the infix operator '∪' and a difference between two sets by the infix operator '-', then we can compute intersection(A∩B) by (A∪B)-(A-B)-(B-A).

11 Implementation in EROS

This chapter gives a brief description of integrating the SNEL query engine (the Sesame implementation) in the ontology browser EROS. This briefness is because of two reasons:

1. Implementation is not yet completed and it will only be continued after the end of this project.
2. Implementation is rather straightforward and therefore only the biggest issues will be mentioned.

EROS has been described in section 5.3.5. Figure 11-1 shows a screenshot of EROS in query-building mode. In this mode an RQL query can be built with help of the ontology browser. If a class or property is selected in the browser the bottom panes can be used to formulate a query based on the selected part.

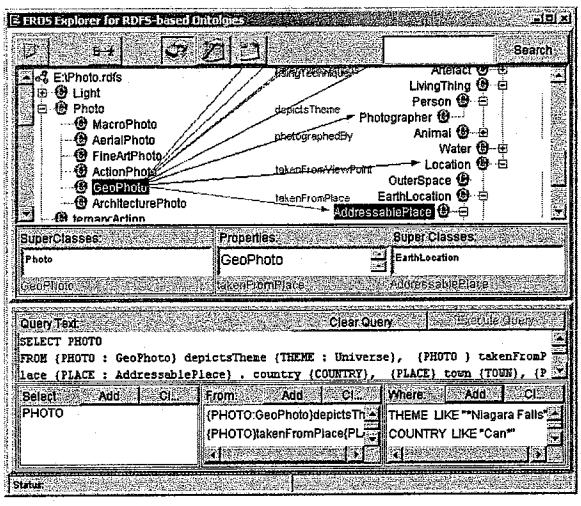

Figure 11-1: EROS query-building mode

A problem is that RQL queries are difficult for inexperienced users. Further this query builder obliges the user to build a query in three parts. Therefore, because of its simplicity and its special focus on RDFS data we wanted to add support for SNEL to EROS (as an alternative for RQL). Furthermore, the connection between Sesame and EROS, which was not yet functional, should be implemented.

For the implementation we want to use the Sesame server as a repository and query engine for the data. Therefore we can also use the implemented SNEL system (as described in section 10) that is integrated in Sesame. We adapted Sesame a bit such that the SNEL query engine is also available in the Sesame API. This is done by letting the API create an HTTP session, sending the corresponding parameters and returning the results of the query (similarly to Sesame's methods for its queries).

Next we constructed the GUI to let the user set up a connection with a Sesame server. Figure 11-2 shows a screenshot of the connection dialog. In this dialog an URL to the SNEL server (Sesame server extended with the SNEL system) and the repository to browse or query over must be defined. And optionally, if this is necessary for read access on the according repository, a username and password can be defined.

The repository name may be entered manually, but the available repositories (with read access) on the defined SNEL server with the corresponding username and password can also be automatically detected by clicking the "Fill Repository" button. The connection with the server may take a long time, especially if the URL is wrong. Therefore, the connection with the SNEL server is executed in a separate thread so that the GUI does not lock up. This is not trivial in Java's GUI component (called Swing). Swing is namely inherently threadless.

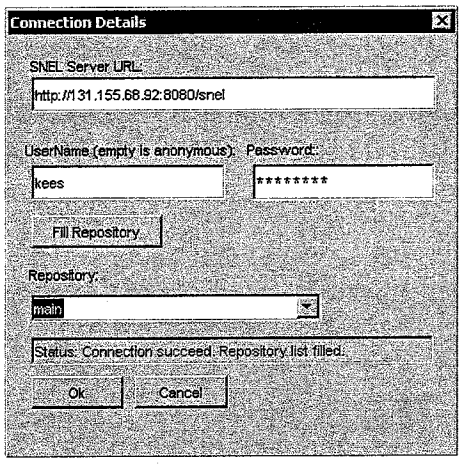

Figure 11-2: EROS - Connection Dialog

After the SNEL connection details are filled in, and the "Ok" button is pressed EROS retrieves the schema information from the defined repository for browsing (not yet implemented).

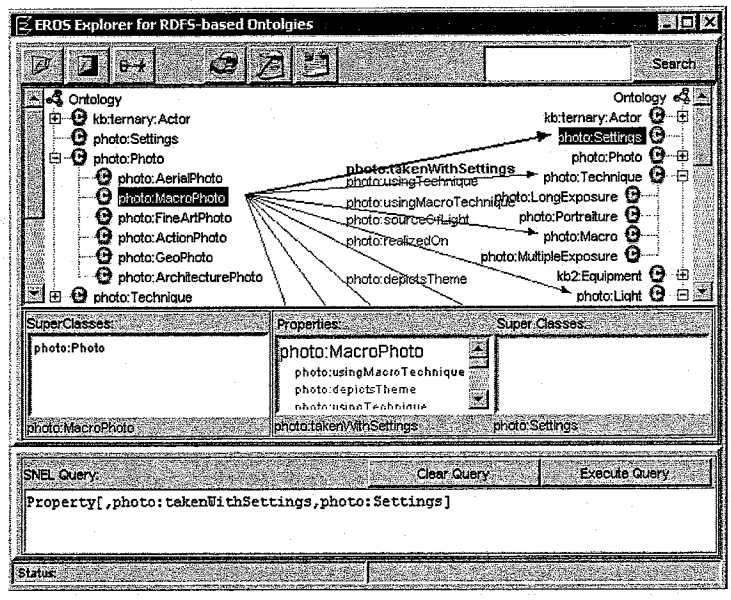

Figure 11-3: EROS explorer

Figure 11-3 shows a screenshot of the altered EROS explorer screen. In the SNEL Query text box a SNEL query can be entered, which can be executed by clicking the "Execute Query" button. This will open a results window, where the results of the query are displayed in RDF/XML format. This functionality is implemented. Sending the query to the server and waiting for the results is again implemented in a separate thread, so that the GUI does not lock up.

What yet has to be implemented is user assistance to (semi) automatically generate queries by performing some actions in the ontology browser and the option to switch between SNEL and (the already implemented) RQL mode. An example for automatic query generation would be to enable the user to perform a right click on a class in the browser and select "retrieve instances" to generate a class query that retrieves all instances for the selected class.

12 Efficiency and scalability

In this section we make an efficiency and scalability analysis of both the SNEL implementation in Sesame (section 12.1) and of SNEL's theoretic framework (section 12.2) and we compare the two in section 12.3. We will look at two factors: space overhead and the query evaluation time complexity. Note that if we refer to Sesame we mean the SNEL implementation in Sesame and if we refer to SNEL we refer to SNEL's theoretic framework.

12.1 Implementation in Sesame

The best way to store and query RDFS data in Sesame is by using a DBMS. The other option, in memory, is only viable if the amount of data does not exceed several dozens of MB. According to the makers of Sesame MySQL is the fastest DBMS for evaluation of their queries. Therefore the MySQL database is used for the efficiency analysis.

12.1.1 Storage overhead

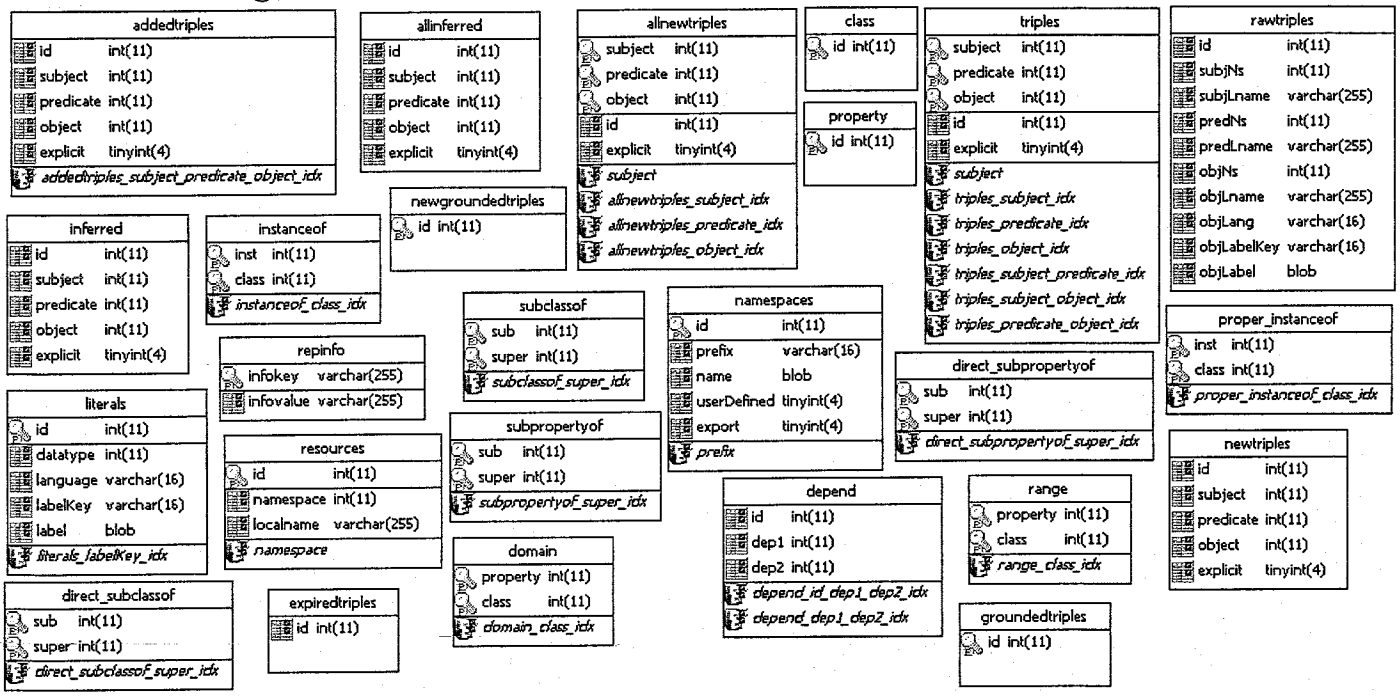

Figure 12-1: Sesame Database layout for RDFS data

Figure 12-1 depicts the tables that are created by Sesame in a MySQL database for a new RDFS repository. This is basically the ER-diagram without the relations. Relations are left out because the data has too many dependencies to properly display within one picture.

Note that a lot of tables maintain a list of triples, namely: addedtriples, allinfered, allnewtriples, triples, rawtriples, newtriples and inferred. However, only one table is really used to constantly maintain the list of all triples (namely "triples"). The others are used for administrative purposes only and act as intermediate tables for adding new triples to the repository.

The creation of a new Sesame RDFS repository automatically fills the database with 120 triples that form the framework of the RDFS data model (the 120 triples are displayed in N-triple notation in appendix F). To measure the space overhead in Sesame, the repository is filled with a variable amount of data and we observe what this means for the size of the

database. The statistics that these measurements produce also tell something about the data that is indexed. For instance, if there is not much schema information in the new inserted data this will add less data to the database than if there is much schema information. Schema information is namely duplicated for several tables like the class and property table, but also for inferring new data by using this schema information.

We create our test data with an RDFS generator that generates schema information and the accompanying instantiations. These data are alternately uploaded into Sesame and measured for a number of statistics. Table 12-1 contains the different configurations for the data we uploaded. Table 12-2 then contains the space overhead Sesame uses for the concerning configurations. Table 12-4, finally, contains the execution time for typical SNEL queries over the different repository configurations.

	# repo configuration									
Repository names / configuration	A	B	C	D	E	F	G	H	I	J (wordnet)
Number of branches per node	2	2	2	3	3	3	3	3	4	0
Depth of the classtree	2	3	4	4	5	6	6	6	8	0
Instances per class	25	25	25	25	25	25	50	500	1	0
Number of properties / class	3	3	3	3	3	3	3	3	5	0
Instances per property	25	25	25	25	25	25	50	500	1	0
Total number of triples	332	776	1.664	4.439	13.430	40.403	76.803	732.003	502.435	473.589

Table 12-1: repository configurations

Table 12-1 contains the repository configurations as we created with the RDFS generator. The configuration parameters all are obvious by their name (the first column). We refer to the different configurations with letters, as stated in the second row. For configuration A to F we only play with the parameters "number of branches per node" and "depth of the classtree" and look at what influence the variation of these parameters have. After that we look at the effect of increasing the instance information (G and H), thus having relatively little schema information and lot of instantiations. For configuration I we look at the effect of much schema information and only little instantiations. Conclusively, for configuration J, we used a schema-less configuration; not a generated file but an RDF dump of the wordnet lexical database ([24]).

Table 12-2 contains the results of the repository storage overhead measurements. The table contains for every MySQL table, which Sesame maintains for its data repository, the number of rows that are generated by Sesame per repository configuration (Table 12-1). The results for the following DB tables are left out of the measurements table because they are always empty: addedtriples, allinferred, allnewtriples, expiredtriples, groundedtriples, inferred, newgroundedtriples, newtriples and rawtriples. This is because those tables are only used as administrative tables for the addition of new data. The results of repinfo table are also left out because this table only contains the constant repository info. For all tables it is intuitively clear what information it holds considering its name, except maybe the "depend" table. Sesame uses this table to keep track of all dependencies between statements, e.g. statement B was derived from statement A, etc. This information is used to determine which sets of statements have to be removed together.

Note the results for column J. Five additional classes were found, even though we declared the wordnet database has no schema information. These classes were inferred by Sesame. In the wordnet file statements of the following type are made:<b:Noun rdf:about="&a;100001740"/>. Sesame then infers that Noun is a class and that a:100001740 is of type Noun.

	# Rows per DB table										
Column names	0	A	B	C	D	E	F	G	H	I	J
# Triples in DB / DB tables	0	332	776	1.664	4439	13.430	40.403	78.803	732.003	502.435	473.589
Triples	120	604	1388	3.164	8.532	28.620	95.202	181.727	1.921.177	1.143.353	573.383
Resources	30	117	233	465	1.190	3539	10.586	19.686	183.486	152.947	99.681
Literals	0	0	0	0	0	0	0	0	0	0	222.240
Class	13	16	20	28	53	134	377	377	377	21.859	18
Subclassof	31	39	55	95	213	699	2.400	2.400	2.400	211.204	41
Direct_subclassof	12	15	19	27	52	133	376	376	376	21.858	17
Property	14	23	35	59	134	377	1.106	1.106	1.106	109.239	18
Subpropertyof	15	24	36	60	135	378	1.107	1.107	1.107	109.240	19
Direct_subpropertyof	1	1	1	1	1	1	1	1	1	1	1
Domain	9	18	30	54	129	372	1.101	1.101	1.101	109.234	9
Range	8	17	29	53	128	371	1.100	1100	1.100	109.233	8
Instanceof	57	281	713	1.777	4.927	17.725	62.194	121.419	1.369.469	495.217	199.359
proper_instanceof	28	115	231	463	1.188	3.537	10.584	19.684	365.484	174.791	99.679
Depend	120	2435	6.467	16.219	45.244	163.261	581.707	1.112.307	11.027.107	7.689.645	1.745.251
Namespaces	3	4	4	4	4	4	4	4	4	4	5
TOTAL #rows	461	4.087	10.143	24.359	66.970	234.397	813.709	1.553.771	14.874.659	10.369.671	2.939.729
(TOTAL # rows - init) / #triples	-	10,92	12,48	14,36	14,98	17,42	20,13	19,71	20,32	20,64	6,21
File size (MB)	0	0,03	0,06	0,14	0,36	1,09	3,28	6,21	59,41	44,10	33,49
DB size (MB)	0,29	0,47	0,79	1,49	3,17	12,13	41,20	77,71	748,50	506,02	170,12
DB size – init (MB)	0	0,18	0,50	1,20	2,88	11,84	40,91	77,42	748,21	505,73	169,83
(DB size – init) / File size		6,71	7,83	8,82	7,97	10,84	12,45	12,47	12,59	11,47	5,07

Table 12-2: Sesame space overhead

In Table 12-2 we compare file size with DB size and total number of table rows with number of triples in DB. These quantities are not comparable just like that. The problem with file size and DB size is that the RDFS file size suffers from the XML syntax overhead, while the database optimizes the required triples storage overhead. However, we assume that the differences in storage are a constant factor. The comparison between total number of table rows with the number of triples in DB suffers a similar problem. The problem there is that not every table row contains the same amount of information and cannot be directly compared. However again, we assume the differences are constant in respect with the number of triples in storage. Thus the relative difference is still comparable, and the measurements do give a reasonable estimate of the order of space overhead compared to the input. The two discussed ratios, R_1 and R_2 are displayed in Figure 12-2 and Figure 12-3.

$$\frac{\text{Database rows}}{\text{Number of input triples}}$$

Figure 12-2: Ratio R_1

$$\frac{\text{Database size}}{\text{File size}}$$

Figure 12-3: Ratio R_2

The RDFS generation tool creates repositories with schema information and instantiations of that schema information (as opposed to stand alone rdf data, not related to rdfs schema data). We note that R_1 is, in this case, always about 1.6 times larger than R_2. The factor difference between these two ratios is caused by the tables that grow the most are kept down in size by using references to data (i.e. with integers), instead of repetition of the entire data. The references point to the Resources table in which the complete URI's and literals are maintained, together with a reference ID. Storing the data in this smart way may save quite some space overhead. Note that for the schema-less repository the ratio factor difference is much smaller, a factor 1,2. This is because of less (smart stored) schema overhead.

If we look at the two ratios (R_1 and R_2) we notice that both grow (even though only logarithmic, not linear) with an increase of the complexity of the schema data, especially the depth of the class tree. This is caused by Sesame pre-computing the closure of the RDFS class and property tree, amongst other inference, and store the result as extra triples in the database.

As our test data repositories like F, G and H have quite reasonable parameters, which we also expect to find in arbitrary SW data, the repository may need to maintain up to 10 or 12 times the amount of input data as space overhead.

12.1.2 Query performance

We first take a look at the time it takes to compile a SNEL query into an equivalent SeRQL-C query (or several). It appears that almost every SNEL query compiles within a millisecond, except for path queries with several distance parameters with larger (>5) values, e.g. for a path query that was compiled using two distance parameters with the respectively values 6 and 5 (this compiles into 30 SeRQL-queries) the compilation time was 23 milliseconds. In any case, query compilations takes insignificant time compared with the SeRQL-C query evaluation time.

Table 12-3 is an overview of translations from typical SNEL queries to equivalent SeRQL-C queries. And then from those SeRQL-C queries to the MySQL queries that are executed by Sesame to solve those SeRQL-C queries. The table gives a good idea of what Sesame does internally. Most translation results are exactly what one would expect. What immediately attracts attention, though, in the MySQL queries that Sesame produces, is that the "like" operator is evaluated by Sesame, not MySQL. The problem is that Sesame first executes all MySQL queries and only then execute the final processing on basis of the results of the MySQL queries. This means that a SeRQL query with a "like" statement for some variable will retrieve all possible values of that variable and then filter out the ones that not fulfil the like operator.

An example to illustrate the problem is the SeRQL-C query:

```
Construct *
From {X0} P1 {X1} P2 {X2} P3 {X3}
Where (X0 like "*term*")
```

Suppose that this "term" does not appear in the database. An immediate conclusion that can then be drawn is that the result of the whole SeRQL-C query is the empty set. Sesame, however, first computes all the results for "{X0} P1 {X1} P2 {X2} P3 {X3}", which even with a repository that contains 25 statements (and the 120 initial RDFS framework triples)

results in approximately 5700 MySQL queries and approximately the same number of results, that have to be filtered by the Sesame query engine.

SNEL query	SeRQL-C query('s)	MySQL queries
Term[*term*]	Construct DISTINCT * From {X}p{Y} Where (X like "**term***" OR p like "**term***" OR Y like "**term***")	SELECT r1.id, r1.namespace, r1.localname, r2.id, r2.namespace, r2.localname, r3.id, r3.namespace, r3.localname FROM triples t, resources r1, resources r2, resources r3 WHERE t.subject = r1.id AND t.predicate = r2.id AND t.object > 0 AND t.object = r3.id
		SELECT r1.id, r1.namespace, r1.localname, r2.id, r2.namespace, r2.localname, l.id, dt.id, dt.namespace, dt.localname, l.language, l.label FROM triples t, resources r1, resources r2, literals l, resources dt WHERE t.subject = r1.id AND t.predicate = r2.id AND t.object < 0 AND t.object = l.id AND l.datatype = dt.id
Structure[child,*uri#class*]	Construct DISTINCT * From {X}<serql:directSubClassOf>{Y} Where Y=<*uri#class*>	SELECT id FROM resources WHERE namespace = 2 AND localname = '*class*'
		SELECT r1.id, r1.namespace, r1.localname FROM direct_subclassof t, resources r1 WHERE t.sub = r1.id AND t.super = 16
Class[*uri#class*]	Construct DISTINCT * From {X}<rdf:type>{< *uri#class* >}	SELECT id FROM resources WHERE namespace = 1 AND localname = 'type'
		SELECT id FROM resources WHERE namespace = 2 AND localname = '*class*'
		SELECT r1.id, r1.namespace, r1.localname FROM triples t, resources r1 WHERE t.subject = r1.id AND t.predicate = 1 AND t.object = 16
Property[*?*,*uri#prop*,Term[*obj*]]	Construct DISTINCT * From {X}p{Y} Where (p=<*prop*>) AND (Y like **obj***")]	SELECT id FROM resources WHERE namespace = 4 AND localname = '*prop*'
		SELECT r1.id, r1.namespace, r1.localname, r3.id, r3.namespace, r3.localname FROM triples t, resources r1, resources r3 WHERE t.subject = r1.id AND t.predicate = 42 AND t.object > 0 AND t.object = r3.id
		SELECT r1.id, r1.namespace, r1.localname, l.id, dt.id, dt.namespace, dt.localname, l.language, l.label FROM triples t, resources r1, literals l, resources dt WHERE t.subject = r1.id AND t.predicate = 42 AND t.object < 0 AND t.object = l.id AND l.datatype = dt.id
Path[term[*x0*],[*3*],term[*x3*]]	Construct * From {X0} P1 {X3} Where (X0 like "**x0***") AND (X3 like "**x3***")	If the repository is empty this triggers 3955 simple SQL queries With 25 triples this triggers 5683 simple SQL queries. 99,99% of the queries are of the form: SELECT r.namespace, r.localname FROM resources r WHERE r.id = 15 The important queries are of the form: SELECT t2.predicate, t0.predicate, t2.object, t1.predicate, t0.object, t0.subject, t1.object FROM triples t0, triples t1, triples t2 WHERE t1.subject = t0.object AND t2.subject = t1.object
	Construct * From {X0} P1 {X1} P2 {X3} Where (X0 like "**x0***") AND (X3 like "**x3***")	
	Construct * From {X0} P1 {X1} P2 {X2} P3 {X3} Where (X0 like "*x0*") AND (X3 like "**x3***")	

Table 12-3: Query translations in Sesame from SNEL→ SeRQL→ MySQL

The number of MySQL queries that will be generated by Sesame grows linearly with the number of paths of length (in this case) 3. This, while the number of queries could be reduced to only a fraction by first evaluating the "like" operators. The reason that Sesame does not do this is probably because their data structure does not permit this just like that. The tables that are used to evaluate the path query contain only references to the data, not the data themselves. Thus applying the database facility for pattern matching in those tables is not possible. For longer path queries, however, evaluation could be done more efficiently by first evaluate solutions for parts of the path, and then join these sub-solutions.

Note that Sesame's like operator is only an extra facility and certainly not intended to be used as often as we do.

	Execution time (ms)										
Column names	0	A	B	C	D	E	F	G	H	I	J
# Triples in repository / SNEL query	0	332	776	1.664	4439	13.430	40.403	78.803	732.003	502.435	473.589
Term[*term*]	16	268	502	767	1.387	3.230	8.156	15.502	1.055.686	174.918	141.531
Structure[subtree,*class*]	15	16	16	16	16	15	16	16	1.658	1.488	1.142
Class[*class*]	8	15	16	15	15	16	79	109	594	74.220	266
Property[*?,,term[obj]*]	15	78	125	235	764	3.074	23.422	33.037	1.839.360	287.416	372.869
Path[term[*x1*],[*3*],term[*x3*]]	1.438	7.745	20.608	51.819	138.913	499.833	2.388.119	$> 7*10^8$	$> 7*10^8$	$> 7*10^8$	$> 7*10^8$

Table 12-4: Query execution time versus triples in the Repository

Table 12-4 (execution time table) contains measurements of the execution time for various SNEL queries for a certain repository size. We chose the queries such that the number of results per repository of some size is roughly equal. This is to enable the Sesame server to rule out communication overhead with MySQL (although on the same computer) as much as possible.

Note the outcome for the path queries for the repositories G to J. Sesame has a built in time-out that queries may not run longer than one hour. If we remove the time-out, query evaluation will not stop after an hour, but seems to go on for an indefinite amount of time. Looking only at repository sizes and the former results we expected the path query for repository G to evaluate on a timescale of about 4 hours. During our tests, however, it went on for more then 8 days ($7*10^8$ ms) and still was not finished. During all that time memory usage was maximal and processor utilization was minimal (only a few percent). The question is if the system got in some kind of livelock (e.g. by a bug in Sesame, Tomcat or MySQL) or that the query evaluation was still running. If the latter is the case continues paging from and to hard drive instead of RAM usage could be the cause. If we estimate disk access 1000 times slower than RAM access, the computation could get in order of 1000 times slower, which means more than 5 months of query evaluation instead of 4 hours. Repositories H to J would then perform even much worse. Whatever the problem eventually turns out to be, we can conclude that performance of path queries is very slow for larger data sets.

Even though it is difficult to depend on the measurements only for derivation of a function for the execution time, given the repository size, it is possible to make estimates by combining the results with the SeRQL to MySQL translations in Table 12-3 (translation table).

We will use the following variables:

- SC: Number of (sub) class declarations (so classes that appear X time in the tree count for X)
- URI: Number of URI's.
- N: Number of triples

Table 12-5: Sesame query evaluation variables

The first thing we notice for all query types is that Sesame first solves the query in the database, which results in number of results. But those results only contain references to URI's and literals, and not the URI's and literals themselves. This is because of the smart way of storing the data, as explained in the former paragraph. However, although this solution saves a considerable amount of space overhead, it will take a lookup of every reference that is in the results in the resource table. Call the results R and the resources table T. Then the overhead for every query in Sesame has a time complexity of: $O3*\|R\|*Log(\|T\|)= O\|R\|*Log(\|T\|)$ time. We will not further consider this time complexity for the rest of the queries, but only the core query solving time complexity.

Term queries in Sesame are evaluated by first retrieving all triples in the database. This takes at least O(N). Then those triples are pattern matched for the *term*, which again costs O(N). This sums to execution time O2*(N) = O(N). This also approximately appears in the query execution time table, taking into account that the first results are somewhat distorted by the overhead cost.

The structure queries in Sesame are, according the execution time table, evaluated in a logarithmic time scale. This also appears from the translation table. The class is first sought in the resources table, which takes Olog(URI) time. This roughly equals Olog(3*N) = Olog(N). Next the class is looked up in the direct_subclassof table which takes Olog(SC) time. This sums up to O3*log(N) + Olog(SC) =Olog(SC*N) time.

The class queries in Sesame are, according the execution time table, also evaluated in a logarithmic time scale. This also appears from the translation table. The "type" and "class" URIs are looked up in the resource table, which costs O2*log(URI)=Olog(N) time. Then the triple table is sought for the found predicate and object ID's, which takes Olog(N) time. This all accumulates to roughly O3*log(N)= Olog(N) time.

The property queries in Sesame are solved similar to the class queries, and thus also in a logarithmic time scale. In our property query, however, we used a term sub query. Therefore the execution is not on a logarithmic time scale. First all the triples with the property "prop" are retrieved, call these results R, which then are in linear time filtered for the concerning term. The resulting time complexity is $O3*\log(N)+O\|R\|= O\log(N)+\|R\|=\{\|R\|$ is largest factor$\}=O\|R\|$.

For the path queries we already explained that the use of term queries here had a dramatic effect on the performance. The execution time takes place, according the execution time table, on an exponential time scale. As we mentioned earlier this time is spent by querying for all possible paths in the database of the specific length and then a resource lookup for every result. In the example query this means it first searches all paths of length 3. So apparently, the

number of paths of length 3 grows on an exponential time scale in accordance with the amount of data in the repository.

12.2 SNEL's theoretic framework

In this paragraph we will analyse the efficiency in space overhead of SNEL's theoretic framework as defined in section 7 and the query performance as defined in section 8.

12.2.1 Storage overhead

We will express the storage overhead with use of a number of variables.

The following variables are of influence:

- SC: Number of (sub) class declarations (so classes that appear X time in the tree count for X)
- CI: Number of class instances (for the instance lists in the class tree)
- C: Number of distinct Classes
- AvgTerm: Average number of terms per URI/Literal
- URLi: Number of URI/Literal 's.
- Trm: Number of distinct terms
- N: Number of triples
- SP: Number of (sub) properties declarations (so properties that appear X time in the tree count for X)
- P: Number of distinct Properties

Table 12-6: SNEL repository variables

Now we count the space overhead used by the data structure we proposed in Figure 7-1 for every separate part.

Class Tree:
SC (the tree) + 2 * SC (all parents + children) + CI (instances)

Class Location Table:
C (class list) + SC (all appearances of classes)

URI/Literal Index:
AvgTerm * URLi

Sorted triple lists:
2* N

Property Tree:
N (#instance pairs + property tree) + 2 * SP (list of parents and children) *

* We neglect domain and range as insignificant

Property Location Table:
P (property list) + SP (all appearances of properties)

For the complete data structure we now accumulate all the parts:
4 * SC + CI + C + AvgTerm * URLi + 3* N + 3 * SP + P

We determine the following relations between the variables:

- SC>C
- SP = ⅓ N
- SP>P

Then the complete data structure has an upper bound size:
5 * SC + CI + AvgTerm * URLi + 4⅓* N

12.2.2 Algorithm performance

We measure the algorithm performance of the SNEL system by giving a short performance analysis of the algorithms described in section 8 and 9. The performance analysis is given in terms of the variables that were used in section 12.2.1. We use the '|' sign to denote parallel operations.

Note that that the following performance analyses are quite straightforward to compute. It is mostly summing the different algorithmic parts, which mostly contain search functions over lists. Regarding that the sorting of the data structure is known beforehand, the performance of the several steps can be written down straight away. The reading of this section may be safely skipped, but mind that its conclusions are used in the performance comparison between the SNEL theoretic framework and the SNEL implementation in Sesame.

12.2.2.1 Term queries

Searching the term in the URI/literal term index has time complexity OLog(Trm). Call the results of this search TR. Next the results of this search are searched in the corresponding tables for subject, property and object. This, then, has time complexity
O‖TR‖*(Log(N)|Log(N)|Log(P)).

This results in:
OLog(Trm)+‖TR‖*(Log(N)|Log(N)|Log(P))= O‖TR‖*Log(N)

12.2.2.2 Structure queries

For structure queries we considered five types of queries, namely:

1. *Return all classes that contain term 'Y'*
2. *Return all classes that are a child of class 'X'*
3. *Return all classes that belong to the sub-tree of class 'X'*
4. *Return all classes that are a parent of class 'X'*
5. *Return all classes that are ancestor of class 'X'*

Performance query 1:
OLog(Trm)+OLog(C)= OLog(C*Trm)

Performance query 2:
OLog(C)

Performance query 3:
OLog(C) + # Results (number of descendants)={#Results is dominant factor}=O#Results

Performance query 4:
OLog(C)

Performance query 5:
OLog(C) + # Results (number of ancestors) ={#Results is dominant factor}=O#Results

12.2.2.3 Class queries

For class queries we considered two types of queries, namely:

1. *Return all instances of class 'X'.*
2. *Return all instance of classes 'X' that contain the term 'Y'.*

Performance query 1:
OLog(C)

Performance query 2:
OLog(Trm) + Log(N)=OLog(Trm*N)

The results of term part of query 2 result in a number of classes 'X'.
These results can further be processed in parallel:
$O\left(|Log(C)|\right)^{\|X\|}$ (For instance if X contains two statements, this gives: O (Log(C) | Log(C))

Which results in:
$OLog(Trm) + Log(N) + \left(|Log(C)|\right)^{\|X\|} = OLog(Trm*N*C)$

12.2.2.4 Property queries

For property queries we considered queries of type <S|?,p|?,O|?>. Intermediate results are consequently called Ri:

Performance <S,p,O>:
$OLog(P) + Log(\|R1\|) + Log(\|R2\|) = OLog(P*\|R1\|*\|R2\|)$

Performance <S,p,?>:
$OLog(P) + Log(\|R1\|) = OLog(P*\|R1\|)$

Performance <S,?,O>:
$OLog(N) + Log(\|R1\|) = OLog(N*\|R1\|)$

Performance <S,?,?>:
OLog(N)

Performance <?,p,O>:
$OLog(P) + \|R1\| = \{\|R1\| \text{ is dominant factor}\} = O\|R1\|$

Performance <?,p,?>:
OLog(P)

Performance <?,?,O>:
OLog(N)

Performance <?,?,?>:
OConstant

Now consider a property query that contains a term query for a variable. We consider two strategies to solve these queries, which are equivalent in terms of global strategies as in 8, but do differ in specific implementation. Especially concerning their performance.

Strategy1:
First the term query part is solved in OLog(Trm) time. Call the result of this part TRi. The property query is executed for every result of this term query. This results in a time complexity multiplication of the concerning property query with a factor $O\|TRi\|$.

Example: the property query <term[1],term[2],term[3]> has time complexity $O\|TR1\|*\|TR2\|*\|TR3\|$.

Strategy2:
First the property query part is solved with the term query variable, e.g. <S,p,term[1]> is first evaluated as <S,p,?>. Call the result of this part PR. This result is then linearly searched for the term query (or queries). This adds a time complexity $O\|PR\|$

The two strategies can be considerable more efficient than the other in different circumstances. Therefore we define a final strategy using both strategies.

Final strategy:
We first compute an estimated value of the variables TRi and PR. For the term queries we look up the term in the URI/literal term index and return only the number of results. This takes OLog(Trm) time complexity. For the property queries we estimate the number of results after evaluating one constant (maximal OLog(N) time complexity), and if there are no constants we immediately know the number of results (namely N). Up to here this cost a logarithmic time complexity. Now if $\|TR1\|*\|TR2\|*\|TR3\| < \|PR\|$ we execute Strategy1 and else we execute Strategy2. Now we conclude as time complexity for term queries in a property query, in terms of time complexity of a non-term property query, which have time complexity, say, OPQ:

$$O\downarrow((\Pi_i TR_i)* PQ , PQ + \|PR\|)$$

12.2.2.5 Path expressions

For path expressions we consider the distance substitution and the query decomposition into property queries to be on an constant time scale. Then we have expressions of the form:
$<X_0 , p_0 , X_1>$
$<X_1 , p_1 , X_2>$
...
$<X_i , p_i , X_{i+1}>$
...
$<X_{n-1} , p_{n-1} , X_n>$

All those queries can be evaluated separately and parallel as property queries in a logarithmic timescale.

We call the results of subquery i Ri. We call the intermediate end result Re. The performance then is:

$Oi*(\|Re\|*Log(\|Ri\|))$

12.2.2.6 Set operations

The set operations all have the same time complexity, namely for two result sets R1 and R2:

$O(\|R1\|+\|R2\|)log(\|R1\|+\|R2\|)$

12.2.2.7 Subclass and Subproperty Expansion

The performance is similar to the performance of the structure subtree query.

O#Results

12.2.2.8 Superclass expansion

Superclass expansion has constant performance.

Superclass expansion plus expansion by computing the subtree of the superclass has performance:

O#Results

12.2.2.9 Instance membership expansion

The performance of the instance membership expansion depends on the number of results that are found thus far. Call the results of the form <a,p,O> R. The performance of computing instance membership expansion then is:

Look op instance relation (<a,rdf:type,?>):
$O\ Log(P)+ Log(\|R1\|)= O\ Log(P*\|R1\|)$

Find alternatives for <a,p,O>, namely <?,p,O> with ? one of the found classes
R2.
Compute <?,p,O> and call the results R4:
$O\ Log(P)+ \|R3\|=\{\|R3\|$ is dominant$\}= O\ \|R3\|$

Parallel with computing <?,p,O> compute all instances for every class in R2 and call the results R5:
$O\ \|R2\|*Log(C)$

Note that R4 is sorted. For every subject of R5 search if it matches with a subject in R4:
$O\ \|R5\|*Log(\|R4\|)$

This all accumulates to:

$$O(\|R\| * (Log(P) + Log(\|R1\|) + (\|R3\| \mid \|R2\| * Log(C)) + \|R5\| * Log(\|R4\|)) =$$
$$O(\|R\| * (Log(P * \|R1\|) + (\|R3\| \mid \|R2\| * Log(C)) + \|R5\| * Log(\|R4\|))$$

12.3 Comparison Sesame versus SNEL's theoretic framework

In this section we try to compare the efficiency of our implementation in Sesame versus an implementation with our own data model.

12.3.1 Storage overhead

In section 12.2.1 we accumulated the following term for the space overhead for the SNEL data model:

$$5 * SC + CI + AvgTerm * URLi + 4\frac{1}{3} * N$$

We will consider repository G of Table 12-2 as a typical repository configuration of some Web repository. Then, we can derive the following estimates:

- SC as insignificant (less than 1% of the data)
- CI as 50% of the data
- AvgTerm (arbitrarily) as 5
- URLi as $\frac{2}{3}N$

This yields less than $8\frac{1}{2} * N$, which is also $8\frac{1}{2}$ times the amount of input data. This is better than Sesame, which we estimate (based on Table 12-2, Ratio R_2) on about 12 times the input data. Note that the SNEL figure is only an estimate, and that the implementation might give rise to extra space overhead. The difference between SNEL and Sesame is in the generation and maintenance of the inferred triples. In Sesame this results in a considerable larger number of triples in the database. Further, Sesame needs to maintain the depend table for update reasons. As for our own data type, we could use a lightweight (i.e. fast) compression algorithm to further reduce storage with approximately factor 3 by using a compression library like zlib (which is used by Google).

12.3.2 Query performance

If we compare query performance of Sesame with SNEL we see that for structure, class and property queries both perform in a logarithmic time scale in relation with the amount of data in the repository. SNEL is however faster because the Sesame results consist of references to the resources instead of the resources themselves, and therefore every reference in the result set has to be looked up in the resource table. This factor is going to play a role only for queries over considerable large data sets.

The biggest difference, however, lies in term queries, and especially term queries in longer path expressions. Where SNEL solves term queries in logarithmic time scale Sesame performs on a linear time scale. And where SNEL solves path queries on a time scale that is a product function of the query path length and the number of intermediate results, Sesame needs a timescale of two times the number of possible paths of the given query path length in the entire data repository. This number of possible paths in the entire data repository is equal to the sum of all values in the adjacency matrix of the data repository raised to the power of the query path. Furthermore, Sesame lacks internal support for set operations like union, intersection and difference. The work-around implementation is certainly not optimal.

For implementation of a search engine that is available for a large user base the current architecture of Sesame is not very suitable. Sesame is clearly not built for evaluating specialized queries, with high use of term (sub) queries, over a very large amount of data. Further, Sesame has no facilities (yet) for data and query distribution by setting up a parallel network. The Sesame developers are making a distributed implementation for in-memory repositories. However, this will most likely not solve all problems we pointed out in our context of a SW search engine. We have to keep in mind though that Sesame has a different target than SNEL. Another important point for us is that Sesame does not offer support to store extra data, like the source (origin) of the triples, or a weight factor for sorting purposes. Extra functionality like set operations or aggregate and mathematical functions are viable for future releases, but also not yet implemented. These, again, are functionalities we desire for future implementation of SNEL, but likely are not within the target of Sesame. The conclusion from this, also based on our experiences with the software, is that Sesame is a very promising storage and retrieval framework, but is not intended, or equipped, to be used as the basis of a Web search engine.

13 Conclusions

The goal of this project was to investigate new techniques to improve the web search process by utilizing the Semantic Web. We already noted early on in the introduction that the Semantic Web was merely a vision, and is only partially implemented yet. We decided to focus on the existing standard RDF(S). We started with a problem description in which we outlined desirable types of queries we are not able to pose with current search engines. We also showed that these query types are inherently linked to the use of metadata. The main idea of the new types of queries we want to be able to pose, is that we want to bring together the best of both the worlds of keyword based search and database search. We want the expressive power similar to that of databases and we want the simplicity of keyword based search engines. We also investigated the possibilities to increase the recall and precision of results by utilizing semantic structures and utilizing this same structure for a better sorting and grouping of the results.

We did a literature study to investigate the current state of the art in the fields of web information retrieval and the Semantic Web. This study provided a basis for the ideas we suggested later for realizing solutions for the problems we had identified earlier. Then we formulated more concrete requirements for the queries, and query relaxation and strengthening methods, of the system we wanted to design. We proposed a data structure that is able to efficiently handle the necessary algorithms for fulfilment of the requirements. Then we presented algorithms for solving the defined queries, and query relaxation and strengthening methods, using the proposed data structure. Note that the presented algorithms are tailored specially to the specific queries. The several algorithms have quite some correspondences, however. Therefore it would be a good idea to, for the future, identify primitive actions that may be used by several (e.g. also new or user customized) algorithms. These actions could then be grouped in procedures that algorithms may use. An example of such a primitive could be "Search URI/Literal 'X' within all triples", which could then be implemented by distributing a binary search for 'X' to the several triple tables (sorted on subject and object, and the property tree), and uniting the results of those distributed searches.

Finally we made an implementation of our queries in an existing SW query engine. For this implementation we used Sesame's query primitives and its data storage. We called our Sesame module "SNEL" and implemented SNEL in such a way, that a remote application, like EROS, may access SNEL query functionality through the Sesame server.

Because Sesame does not use our data structure, which is specifically tailored to our needs, we expected the implementation in Sesame to be inherently less efficient than an implementation that uses our defined data type. We made an efficiency and scalability analysis for both systems and found that, even though its efficiency is quite reasonable, Sesame is not suited to serve as a basis for a full-featured search engine. It especially lacks efficiency for keyword based queries and inflexibility for customizing to our specialized needs (like accommodation for large scale query and data distribution). Therefore, if a concrete implementation for a Semantic Web search engine that is scalable for web sized data is to be made, SNEL's theoretic framework should be elaborated.

The query system we suggested for the Semantic Web is only a part of a search engine. In this project we did not spend time on the crawling part of a search engine nor to the GUI of a search engine. The former can probably be implemented pretty similarly to the web crawlers for current search engines. The latter, the GUI, is a bigger problem. One issue is how to present

the query language; another is how to present the query results to the user. The SNEL query language is pretty simple to use, but nevertheless users still may need some tool support to formulate their queries. Building in SNEL support in EROS may be a first step in development of such a GUI.

For the future, the data structure should be extended to accommodate for the facilities of sorting and grouping of the results. In the URI/Literal term index, for instance, a weight value could be attached to the URIs that contains some term. These weights could be used to sort the results, possibly in combination with some other heuristics. Furthermore the RDF triples could be extended with their originating source URI. This could be used to judge the relevancy of some RDF information source or an entire domain of RDF information sources.

In addition to these extensions, the data structure should be adapted so that it supports a large degree of parallelism. On the highest abstraction level this adaptation should leave the data structure as it is. But underneath this abstraction there may be need for some mechanisms on the low level for e.g.: data duplication, algorithms for dividing data over several physical locations and computers, algorithms for combining results of queries to several machines, caching mechanisms for frequently asked queries (and their results), etcetera. The query algorithms should also be adapted to a high degree of parallelism. Some user query can for instance be divided into a number of subqueries that could be divided over several machines.

Another recommendation is the extensions of the path queries so that they support branching and reified statements. This functionality is quite straightforward to implement. The only worry is how to allow for branching and reified statements without compromising to simplicity and intuitive clarity of the SNEL query language. Because of time limitations we left out this part.

Another nice feature for extending the path queries would be path queries of arbitrary length along one transitive property. Consider, for instance, a "boss_of" relation. Examples of triples that have that property are <X, boss_of,Y> and <Y, boss_of,Z>. Now we want the feature of queries like: "return all bosses of Z" or "return all subordinates of X". The functionality is similar with the structural subtree and ancestors queries for the class and property hierarchy, however now for arbitrary properties.

Another thing that we left out due to time limitations was exploiting the domain and range constraints on properties. These constraints are explicitly given and it should be possible to exploit this extra information somehow. This information might possibly be used to improve evaluation speed, for instance by restricting the search to the instances of the concerning classes. Another possibility is that some SNEL query construct could be expanded or added that enable users to integrate this RDFS part into the queries.

A future adaptation could also include other typical database constructs like mathematical operations, aggregate functions and nesting of queries. Those functions do not really have a link with the Semantic Web structure and are quite straightforward to implement, but do add some user functionality. A concern is how those constructs can be fitted in the query language without compromising the simplicity, so that they are also available for the average user.

The last recommendation for the query language is to provide a query construct that is fuzzier than the distance operator in path queries. Such a query construct, which we for instance can call "link", should be able to find all connections between some keywords. For instance the

query "Link[kees,tue]" should find all paths between "kees" and "tue", with disregard of arc direction in the RDF graph, and with some threshold for the maximum length of the path or the maximum execution time. A result for this query could be, for instance, the path:

Kees van der Sluijs <- author <- Search the Semantic Web thesis -> ownership -> IS -> part of -> TUe

The issues mentioned in this section are collected in Table 13-1. For every issue that should be treated we attached a priority between 1 (highest) and 3 (lowest). The choice of priorities is maybe disputable, but gives our view of what we think are the most important (and what we consider less urgent) issues to be handled.

To-do	Priority (1=highest)
Adapt data structure for concrete support of parallelization.	1
Branching in path queries.	1
Build data structure query primitives.	1
Build in grouping and sorting facilities in data structure.	2
Build in the "link" operator.	3
Build in Web crawler	2
Domain and range support.	3
Maintain for every triple its origin..	2
Mathematical and aggregate functions.	3
Path queries of arbitrary length along one transitive property.	1
Support nesting of queries.	2
Workout GUI.	2

Table 13-1: SNEL extension agenda

Appendices

A Bibliography

[1] R. Baeza-Yates, B. Neto (1999), *Modern Information Retrieval*, Addison-Wesley.

[2] S. Brin, L. Page (1998), *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, in Proceedings of the seventh international conference on World Wide Web 7, ACM, pp. 107 – 117.

[3] T. Berners-Lee (2000), *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*, HarperBusiness

[4] T. Berners-Lee, J. Hendler, O. Lassila (2001), *The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities*, in Scientific American, May 2001 issue.

[5] B. Berendt, A. Hotho, G. Stumme (2002), *Towards Semantic Web Mining*, in Proceedings 1st International Semantic Web Conference, Springer, pp. 264-278.

[6] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler (4 February 2004), *Extensible Markup Language (XML) 1.0 (Third Edition)*, W3C Recommendation.

[7] J. Paredaens (2003), *Databases III*, Lecture Notes, http://win-www.ruca.ua.ac.be/u/adrem/courses/DB3/DBIII.02-03.pdf.

[8] O. Lassila, R. Swick, (22 February 1999), *Resource Description Framework (RDF) model and syntax specification*, W3C Recommendation.

[9] P. Champin (2001), *RDF tutorial*, http://www710.univ-lyon1.fr/~champin/rdf-tutorial/rdf-tutorial.pdf.

[10] D. Brickley, R.Guha (23 January 2003), *RDF vocabulary description language 1.0: RDF Schema*, W3C Working Draft.

[11] M. Smith, C. Welty, D. McGuinness (18 August 2003), OWL Web Ontology Language Guide, W3C Candidate Recommendation.

[12] Z. Bar-Yossef, Y. Kanza, Y. Kogan, W. Nutt, Y. Sagiv (1999), Querying Semantically Tagged Documents on the World-Wide Web, in Proceedings of the Fourth Workshop on Next Generation Information Technologies and Systems, Springer-Verlag, pp. 2-19

[13] J. Davies, R. Weeks, U. Krohn (2002), *QuizRDF: Search Technology for the Semantic Web*, in WWW2002 workshop on RDF & Semantic Web Applications, 11th International WWW Conference WWW2002, pp. 50-59

[14] J. Mayfield, T. Finin (2003), *Information Retrieval on the Semantic Web: Integrating inference and retrieval*, in Proceedings of the SIGIR 2003 Semantic Web Workshop, ACM

[15] U. Shah, T. Finin, A. Joshi, R. Scott Cost, J. Mayfield (2002), *Information Retrieval on the Semantic Web*, 10th International Conference on Information and Knowledge Management.

[16] R. Vdovjak, P. Barna, G.J. Houben (2003), *EROS: A User Interface for the Semantic Web*, in SCI 2003, 7th World Multiconference on Systemics, Cybernetics and Informatics, pp. 485-490.

[17] G. Klyne, J. Caroll (5 September 2003), *Resource Description Framework (RDF):Concepts and Abstract Syntax*, W3C Working Draft.

[18] J. Broekstra, A. Kampman, F. van Harmelen (2002), *Sesame: A generic architecture for storing and querying rdf and rdf schema*, in The Semantic Web – ISWC 2002, volume 2342 of Lecture Notes in Computer Science, Springer, pp. 54-68.

[19] J. Broekstra, A. Kampman (2003), *SeRQL: A Second Generation RDF Query Language*, in SWAD-Europe Workshop on Semantic Web Storage and Retrieval - Position Papers, SWAD-Europe.
[20] T. Parr, *ANTLR: Another Tool for Language Recognition*, http://www.antlr.org.
[21] B. Shidlovsky, E. Bertino (1996), *A graph-theoretic approach to indexing in Object-Oriented databases*, in Proceedings of the Twelfth International Conference on Data Engineering, IEEE Computer Society, pp. 230-237.
[22] H. Stuckenschmidt, R. Vdovjak, J. Broekstra, G.J. Houben (2003), *Towards Distributed RDF Querying with Sesame (Draft)*, Internal Document.
[23] *Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/wiki/Main_Page
[24] *Wordnet, A lexical database for the English language*, http://www.cogsci.princeton.edu/~wn/

B List of figures

C Terminology

Class Query	Query type that searches for all instantiations of some class.
Conceptual Schema	A map of concepts and their relationships. ([23])
Difference Query	Query type that takes the typical set operation "Difference" on the results of two defined subqueries, where the results of the second query are subtracted from the results of the first query.
Intersection Query	Query type that takes the typical set operation "Intersection" on the results of two defined subqueries.
Metadata	Data about data.
Ontology	Formulation of a conceptual schema within a given domain, a typically hierarchical data structure containing all the relevant entities and their relationships and rules within that domain.([23])
Path	A number of successive triples, with for every two successive triples the following pattern: $<X_{i-1}, p_{i-1}, X_i>$ $<X_i, p_i, X_{i+1}>$
Path Distance	In a Semantic Web graph the path distance between a start and end node is the number of directed edges that have to traversed to get from the start node to the end node.
Path Query	A query to find all paths that fulfill some RDF triple pattern that defines conditions or variables for its components.
Property Query	A query to find all triples that fulfill some RDF triple pattern that defines conditions or variables for its components.
Ranking	A sorting algorithm to sort search results on "relevancy". The measurement of the relevancy of a document is the variable factor per algorithm.
RDF triple	RDF triples have the following pattern: <S,p,O> The three components are subject, predicate (or property) and object.
Semantic Structure	A structure that defines the meaning of concepts and the relation of concepts with other concepts.
Special Keyword	Some term in a document that is relatively rare in the complete collection of all indexed resources, but appears more frequent in the specific document.
Structure Query	Query type that searches in the hierarchical class and property structures of RDFS.
Term Query	Query type that searches for some term in resources, where that term might be part of an URI or Literal.
Union Query	Query type that takes the typical set operation "Union" on the results of two defined subqueries.

D Abbreviations and acronyms

ANTLR	ANother Tool for Language Recognition
ASCII	American Standard Code for Information Interchange
ASP	Active Server Pages
AVI	Audio-Visual Interleave
(e)BNF	(extended) Backus Naur Form
DAML	DARPA Agent Markup Language
DARPA	Defense Advanced Research Projects Agency
DB	DataBase
DBMS	DataBase Management System
DNS	Domain Name Server
ER-Diagram	Entity-Relation Diagram
EROS	Explorer for Rdf(s)-based OntologieS
GUI	Graphical User Interface
HAIRCUT	Hopkins Automated Information Retriever for Combining Unstructured Text
HTML	HyperText Markup Language
IR	Information Retrieval
JESS	Java Expert System Shell
JPEG	Joint Photographic Experts Group
MP3	MPEG1 Audio Layer 3
MPEG	Moving Pictures Experts Group
OEM	Original Equipment Manufacturer
OHTML	Objective HTML
OIL	Ontology Integration Language
OWL	Web Ontology Language
OWL DL	OWL Description Logic
PHP	PHP Hypertext Preprocessor
PNG	Portable Network Graphics
Q&A	Question and Answers
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
RDQL	RDF Data Query Language
RQL	RDF Query Language
SeRQL	Sesame RQL
SeRQL-C	SeRQL-Construct
SeRQL-S	SeRQL-Select
SNEL	Search Nearly Everything Language
SW	Semantic Web
TUE	Technische Universiteit Eindhoven
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
W3C	World Wide Web Consortium
WAV	Windows WAVeform
WWW	World Wide Web
XML	eXtensible Markup Language

2490

E SNEL syntax

E.1 Term queries

Return all resources that contain term 'Y'.

Syntax: **Term[a,X]**

X is of type term (may also be several terms seperated by white space. The a parameter is optional and indicates in which part of the triple (subject/predicate/object) the term must appear. The default value for a is '0', which means in may appear in any part of the triple. The values '1','2' and '3' respectivally indicate to only search in the subject, predicate or object.

Example query: Term[1,course]

E.2 Structure queries.

E.2.1 Class/property search

Return all classes/properties that contain term 'X'.

Syntax: **Structure[Class|Property,Z]**

This enables to search the class respectivally property tree. Z is of type URI (and should later become of type set of URIs and determines the URIs that are declared as classes. This structure query type may be used in conjunction with Term structure, i.e. Structure[Class|Property, Term[X]] finds all classes respectivily property declarations that contain the requested term.

Example query: Structure[Class,Term[Course]]

E.2.2 Child search

Return all direct children of class/property 'Z'.

Syntax: **Structure[Class|Property,child,Z]**

Z should be of type class (or property). This query will then return all the "children" direct subclasses of Z. Again, this type of query may be used in conjunction with the Term structure.

Example query:
Structure[Class,child,http://www.student.tue.nl/E/~K.A.M.v.d.Sluijs/kees.rdfs#subklasse11]

E.2.3 Subtree search

Return all children or descendants of class/property 'Z'.

Syntax: **Structure[Class|Property,subtree,Z]**

Z should be of type class (or property). This query will then return Z and all subclasses of Z including the subclasses of its subclasses, etc. Again, this type of query may be used in conjunction with the Term structure.

Example query:
Structure[Class,subtree,http://www.student.tue.nl/E/~K.A.M.v.d.Sluijs/kees.rdfs#subklasse11]

E.2.4 Parent search

Return all classes of which class/property 'Z' is a direct sub class/property.

Syntax: **Structure[Class|Property,parent,Z]**

Z should be of type class (or property). This query will then return the direct parent class(es) of Z. This type of query may be used in conjunction with the Term structure.

Example query:
Structure[Class,parent,http://www.student.tue.nl/E/~K.A.M.v.d.Sluijs/kees.rdfs#subklasse11]

E.2.5 Ancestor search

Return all classes of which class/property 'Z' is a descendant.

Syntax: **Structure[Class|Property,ancestors,Z]**

Z should be of type class (or property). This query will then return Z and all classes that contain Z in its subtree. This type of query may be used in conjunction with the Term structure.

Example query:
Structure[ancestors,http://www.student.tue.nl/E/~K.A.M.v.d.Sluijs/kees.rdfs#subklasse11]

E.3 Class queries

Return all instances of class 'X'.
Return all instance of classes 'X' that contain the term 'Y'.

Syntax: **Class[Z]**

Z should be of type class. This query will then return the instances of Z (thus: <?,rdf:type,Z>). Instead of Z also the Term construct may be used.

Example query: Class[Term[Course]]

E.4 Property queries

Return all instance pairs that have property 'X',
Return all instances that have property 'X' with object value 'Y'.
Return all instances that are the object value from a property 'X' with subject 'Y'.

Syntax: **Property[X|ε , p|ε , Y|ε]**

X stands for the subject of a property, p for the property name and Y for the object of the property. Leaving one of those empty means it is variable. In all three cases also the Term construct may be used.

Example query: Property[,Term[type],Term[Course]]

E.5 Path queries

E.5.1 Regular path queries

Return all instances of class 'X1' that have a property 'Y1' with an object value of class 'X2' that has a property 'Y2' with an object value 'Z'.

Syntax: **Path[** $X_1 |\varepsilon$ **,** $(p_i |\varepsilon , X_i |\varepsilon)^*$ **]** (i>1)

X stands for a node of the RDF graph and p for a property relation between nodes. The path constructor enables users to define path queries of arbitrary length. Leaving some position empty (which e.g. leads to two consecutive commas) the position is used as a variable. In all position also the Term construct may be used. Note that a path query with three elements is equal to property query.

Example query: Path[,serql:directSubClassOf,,serql:directSubClassOf,term[subklasse]]

E.5.2 Distance path queries

Return all instances of class 'X1' that have some property with an object value 'Y' at a distance of at most two property-object values.

Syntax: **Path[** $X_1 |\varepsilon$ **,** $(p_i |\varepsilon |$ **[(exact,)**$^?$ **int]** , $X_i |\varepsilon)^*$ **]** (i>1,int>0)

This query type is similar with the first path query type. The difference is that, instead of some property relationship , the user may define the maximum distance between two nodes in terms of connecting properties. For instance the query:
Path[term[geertjan],[3],term[tue]]
This query indicates that the distince of some result node that contains the term "geertjan" should be at most three property-object relations from the term "tue". The following fictive subject-property-object chain would thus be found: geertjan-worksat-IS-subsectionof-computerscience-facultyof-tue. The following chain would also be found: geertjan-worskat-tue, because "tue" is also at at most three property-object relations from "geertjan" (namely 1). If the optional exact directive is used before the distance parameter the query will compute the exact distance instead of the maximum distance.

Example query: Path[term[geertjan],[exact,2],term[computer science]]

E.6 Union queries

Return all instances that fulfil one of the following queries.

Syntax: **Union[**$query_1$ **(,** $query_i)^*$ **]**

Unites the results of the specified set of queries. The result of the union is the set of all triples that occur in one of the resultset of the specified queries.

Example query: Union[Path[term[geertjan],[2],term[computer science]],term[computer science],term[course]]

E.7 Intersection queries

Return all instances that fulfil all of the following queries.

Syntax: **Intersect[query1 , query2]**.

Intersects the result of the two specified queries. The result of the intersections is the set of triples that occur both in the resultset of the first query and in the resultset of the second query.

Example query: Intersect[Path[term[IS],[2],term[tue]],term[science]]

E.8 Difference queries

Return all instances that fulfil query1, but not fulfil the second query.

Syntax: **Difference[query1 , query2]**.

Takes the difference of the two specified queries. The result of the difference is the set of triples that occur in the resultset of the first query, but do not occur in the resultset of the second query.

Example query: Difference[Path[term[geertjan],[exact,3],term[tue]],term[worksat]]

F Initial inferred data in Sesame RDFS repository

The following triples are initially in a newly created Sesame RDFS repository:

<rdf:type> <rdf:type> <rdf:Property> .
<rdf:type> <rdf:type> <rdfs:Resource> .
<rdf:type> <rdfs:subPropertyOf> <rdf:type> .
<rdf:type> <rdfs:range> <rdfs:Class> .
<rdf:Property> <rdf:type> <rdfs:Resource> .
<rdf:Property> <rdf:type> <rdfs:Class> .
<rdf:Property> <rdfs:subClassOf> <rdf:Property> .
<rdf:Property> <rdfs:subClassOf> <rdfs:Resource> .
<rdf:XMLLiteral> <rdf:type> <rdfs:Resource> .
<rdf:XMLLiteral> <rdf:type> <rdfs:Class> .
<rdf:XMLLiteral> <rdf:type> <rdfs:Datatype> .
<rdf:XMLLiteral> <rdfs:subClassOf> <rdf:XMLLiteral> .
<rdf:XMLLiteral> <rdfs:subClassOf> <rdfs:Resource> .
<rdf:XMLLiteral> <rdfs:subClassOf> <rdfs:Literal> .
<rdf:subject> <rdf:type> <rdf:Property> .
<rdf:subject> <rdf:type> <rdfs:Resource> .
<rdf:subject> <rdfs:subPropertyOf> <rdf:subject> .
<rdf:subject> <rdfs:domain> <rdf:Statement> .
<rdf:predicate> <rdf:type> <rdf:Property> .
<rdf:predicate> <rdf:type> <rdfs:Resource> .
<rdf:predicate> <rdfs:subPropertyOf> <rdf:predicate> .
<rdf:predicate> <rdfs:domain> <rdf:Statement> .
<rdf:object> <rdf:type> <rdf:Property> .
<rdf:object> <rdf:type> <rdfs:Resource> .
<rdf:object> <rdfs:subPropertyOf> <rdf:object> .
<rdf:object> <rdfs:domain> <rdf:Statement> .
<rdf:Statement> <rdf:type> <rdfs:Resource> .
<rdf:Statement> <rdf:type> <rdfs:Class> .
<rdf:Statement> <rdfs:subClassOf> <rdf:Statement> .
<rdf:Statement> <rdfs:subClassOf> <rdfs:Resource> .
<rdf:Alt> <rdf:type> <rdfs:Resource> .
<rdf:Alt> <rdf:type> <rdfs:Class> .
<rdf:Alt> <rdfs:subClassOf> <rdf:Alt> .
<rdf:Alt> <rdfs:subClassOf> <rdfs:Resource> .
<rdf:Alt> <rdfs:subClassOf> <rdfs:Container> .
<rdf:Bag> <rdf:type> <rdfs:Resource> .
<rdf:Bag> <rdf:type> <rdfs:Class> .
<rdf:Bag> <rdfs:subClassOf> <rdf:Bag> .
<rdf:Bag> <rdfs:subClassOf> <rdfs:Resource> .
<rdf:Bag> <rdfs:subClassOf> <rdfs:Container> .
<rdf:Seq> <rdf:type> <rdfs:Resource> .
<rdf:Seq> <rdf:type> <rdfs:Class> .
<rdf:Seq> <rdfs:subClassOf> <rdf:Seq> .
<rdf:Seq> <rdfs:subClassOf> <rdfs:Resource> .
<rdf:Seq> <rdfs:subClassOf> <rdfs:Container> .
<rdf:List> <rdf:type> <rdfs:Resource> .
<rdf:List> <rdf:type> <rdfs:Class> .
<rdf:List> <rdfs:subClassOf> <rdf:List> .
<rdf:List> <rdfs:subClassOf> <rdfs:Resource> .
<rdf:first> <rdf:type> <rdf:Property> .
<rdf:first> <rdf:type> <rdfs:Resource> .
<rdf:first> <rdfs:subPropertyOf> <rdf:first> .
<rdf:first> <rdfs:domain> <rdf:List> .
<rdf:rest> <rdf:type> <rdf:Property> .
<rdf:rest> <rdf:type> <rdfs:Resource> .
<rdf:rest> <rdfs:subPropertyOf> <rdf:rest> .
<rdf:rest> <rdfs:domain> <rdf:List> .
<rdf:rest> <rdfs:range> <rdf:List> .
<rdf:nil> <rdf:type> <rdf:List> .
<rdf:nil> <rdf:type> <rdfs:Resource> .
<rdfs:Resource> <rdf:type> <rdfs:Resource> .
<rdfs:Resource> <rdf:type> <rdfs:Class> .
<rdfs:Resource> <rdfs:subClassOf> <rdfs:Resource> .
<rdfs:Class> <rdf:type> <rdfs:Resource> .
<rdfs:Class> <rdf:type> <rdfs:Class> .
<rdfs:Class> <rdfs:subClassOf> <rdfs:Resource> .
<rdfs:Class> <rdfs:subClassOf> <rdfs:Class> .
<rdfs:Literal> <rdf:type> <rdfs:Resource> .
<rdfs:Literal> <rdf:type> <rdfs:Class> .
<rdfs:Literal> <rdfs:subClassOf> <rdfs:Resource> .
<rdfs:Literal> <rdfs:subClassOf> <rdfs:Literal> .
<rdfs:subClassOf> <rdf:type> <rdf:Property> .
<rdfs:subClassOf> <rdf:type> <rdfs:Resource> .
<rdfs:subClassOf> <rdfs:subPropertyOf> <rdfs:subClassOf> .
<rdfs:subClassOf> <rdfs:domain> <rdfs:Class> .
<rdfs:subClassOf> <rdfs:range> <rdfs:Class> .
<rdfs:subPropertyOf> <rdf:type> <rdf:Property> .
<rdfs:subPropertyOf> <rdf:type> <rdfs:Resource> .
<rdfs:subPropertyOf> <rdfs:subPropertyOf> <rdfs:subPropertyOf> .
<rdfs:subPropertyOf> <rdfs:domain> <rdf:Property> .
<rdfs:subPropertyOf> <rdfs:range> <rdf:Property> .
<rdfs:domain> <rdf:type> <rdf:Property> .
<rdfs:domain> <rdf:type> <rdfs:Resource> .
<rdfs:domain> <rdfs:subPropertyOf> <rdfs:domain> .
<rdfs:domain> <rdfs:domain> <rdf:Property> .
<rdfs:domain> <rdfs:range> <rdfs:Class> .
<rdfs:range> <rdf:type> <rdf:Property> .
<rdfs:range> <rdf:type> <rdfs:Resource> .
<rdfs:range> <rdfs:subPropertyOf> <rdfs:range> .
<rdfs:range> <rdfs:domain> <rdf:Property> .
<rdfs:range> <rdfs:range> <rdfs:Class> .
<rdfs:comment> <rdf:type> <rdf:Property> .
<rdfs:comment> <rdf:type> <rdfs:Resource> .
<rdfs:comment> <rdfs:subPropertyOf> <rdfs:comment> .
<rdfs:comment> <rdfs:range> <rdfs:Literal> .
<rdfs:label> <rdf:type> <rdf:Property> .
<rdfs:label> <rdf:type> <rdfs:Resource> .
<rdfs:label> <rdfs:subPropertyOf> <rdfs:label> .
<rdfs:label> <rdfs:range> <rdfs:Literal> .
<rdfs:isDefinedBy> <rdf:type> <rdf:Property> .
<rdfs:isDefinedBy> <rdf:type> <rdfs:Resource> .
<rdfs:isDefinedBy> <rdfs:subPropertyOf> <rdfs:isDefinedBy> .
<rdfs:isDefinedBy> <rdfs:subPropertyOf> <rdfs:seeAlso> .
<rdfs:seeAlso> <rdf:type> <rdf:Property> .
<rdfs:seeAlso> <rdf:type> <rdfs:Resource> .
<rdfs:seeAlso> <rdfs:subPropertyOf> <rdfs:seeAlso> .
<rdfs:Datatype> <rdf:type> <rdfs:Resource> .
<rdfs:Datatype> <rdf:type> <rdfs:Class> .
<rdfs:Datatype> <rdfs:subClassOf> <rdfs:Resource> .
<rdfs:Datatype> <rdfs:subClassOf> <rdfs:Class> .
<rdfs:Datatype> <rdfs:subClassOf> <rdfs:Datatype> .
<rdfs:Container> <rdf:type> <rdfs:Resource> .
<rdfs:Container> <rdf:type> <rdfs:Class> .
<rdfs:Container> <rdfs:subClassOf> <rdfs:Resource> .
<rdfs:Container> <rdfs:subClassOf> <rdfs:Container> .
<rdfs:ContainerMembershipProperty> <rdf:type> <rdfs:Resource> .
<rdfs:ContainerMembershipProperty> <rdf:type> <rdfs:Class> .
<rdfs:ContainerMembershipProperty> <rdfs:subClassOf> <rdf:Property> .
<rdfs:ContainerMembershipProperty> <rdfs:subClassOf> <rdfs:Resource> .
<rdfs:ContainerMembershipProperty> <rdfs:subClassOf>
<rdfs:ContainerMembershipProperty> .

With rdf: ≡ http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs: ≡ http://www.w3.org/2000/01/rdf-schema#