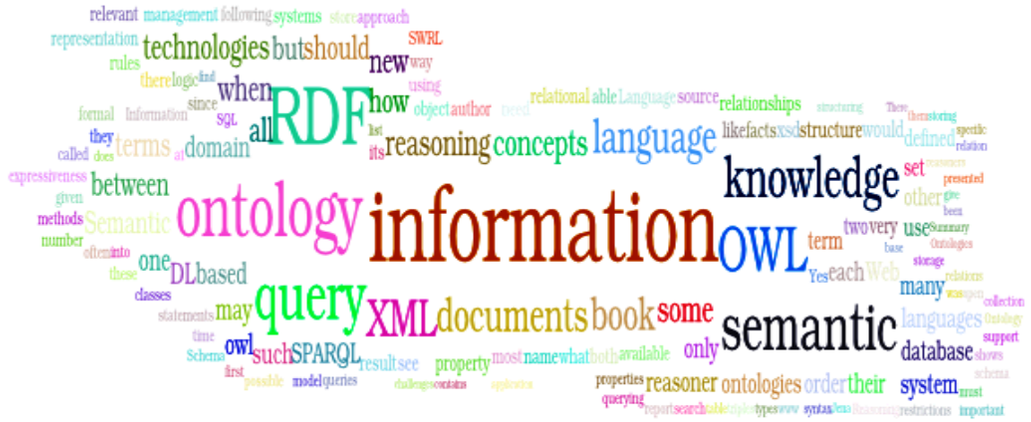
Introduction to Technologies and Methods for Semantic Information Management

HIRAD ASADI, MARIANELA GARCIA LOZANO, ANDREAS HORNDAHL,
EDWARD TJÖRNHAMMAR, KTH - KATHARINA RASCH



FOI, Swedish Defence Research Agency, is a mainly assignment-funded agency under the Ministry of Defence. The core activities are research, method and technology development, as well as studies conducted in the interests of Swedish defence and the safety and security of society. The organisation employs approximately 1000 personnel of whom about 800 are scientists. This makes FOI Sweden's largest research institute. FOI gives its customers access to leading-edge expertise in a large number of fields such as security policy studies, defence and security related analyses, the assessment of various types of threat, systems for control and management of crises, protection against and management of hazardous substances, IT security and the potential offered by new sensors.

FOI
Swedish Defence Research Agency
InformationSystems
P.O. Box 1165
SE-581 11 LINKÖPING

Phone: +46 13 37 80 00
Fax: +46 13 37 81 00

www.foi.se

FOI-R--2858--SE
ISSN 1650-1942

User report
November 2009

Information Systems

Introduction to Technologies and Methods for Semantic Information Management

The Figure on the front page was made with a tag cloud tool made by Chris Done et.al. A tag cloud is a cloud of words that have a size and position according to the frequency with wich they appear in a text. This cloud is the result of the text found in this report.

FOI-R--2858--SE

Hirad Asadi, Marianela García Lozano, Andreas Horndahl,
Edward Tjörnhammar, KTH - Katharina Rasch

Introduction to Technologies and Methods for Semantic Information Management

Titel	Introduktion till teknologier och metoder för informationshantering
Title	Introduction to Technologies and Methods for Semantic Information Management
Rapportnummer / **Report no**	FOI-R--2858--SE
Rapporttyp / **Report type**	Användarrapport / User report
Utgivningsår / **Year**	2009
Antal sidor / **Pages**	90
Kund / **Customer**	
Forskningsområde	2. Operationsanalys, modellering och simulering
Research area	2. Operational Research, Modelling and Simulation
Delområde	21. Modellering och simulering
Sub area code	21. Modelling and Simulation
Projektnummer / **Project no**	E53079
Godkänd av / **Approved by**	Martin Rantzer Head, Information Systems
ISSN	ISSN-1650-1942

FOI Swedish Defence Research Agency
Information Systems
P.O. Box 1165
SE-581 11 LINKÖPING

Abstract

Data stored in different information systems can easily grow to a large quantity over a period of time. Information management, such as retrieving the relevant data, quickly becomes a complex and tedious task. Hence, we want to be aided by the inherent structure of our data through automation. However, computer aided information management gives rise to many questions, e.g., where and how the information is stored, how the information is structured and clustered, how we retrieve the information and can algorithms help us gain any new knowledge from the current information.
In this report we bring up semantic technologies as a way to interpret information, we also give concrete examples. We will also talk about the challenges that occur when using the semantic information and how they could be dealt with. The report is intended for an audience with an interest in technologies and methods for information management. It could be used as a reference for the reader who wishes to learn more on the subject.

Keywords

information management, knowledge representation, semantic web, language stack, ontology, reasoning, storage, logic, querying

Sammanfattning

Den mängd information som lagras i informationssystem växer sig ofta stor efter en tid och informationshantering, så som att hitta relevant information, kan snabbt bli en komplicerad och tidsödande uppgift. Därför vill vi ta hjälp av datorer. Emellertid ger datorstödd informationshantering upphov till många frågor, så som t.ex. var och hur man kan lagra informationen, hur informationen bör struktureras och grupperas, hur vi hittar relevant information och om algoritmer kan hjälpa oss att finna ny kunskap baserat på det vi redan vet.
I denna rapport diskuterar vi semantiska tekniker och metoder. Vi kommer att diskutera ämnen så som informationssökning, strukturering av information, ontologier, språkformalismer för kunskapsrepresentation, inferens och inferensmotorer och slutligen tekniker för lagring samt frågespråk. Vi kommer även att ta upp vilka utmaningar man står inför om man vill utnyttja dessa tekniker och hur dessa kan hanteras. Rapporten är avsedd för läsare med intresse för tekniker och metoder för informationshantering och kan användas som utgångspunkt för den som vill lära sig mer om ämnet.

Nyckelord

information management, knowledge representation, semantic web, language stack, ontology, reasoning, storage, logic, querying

Executive Summary

This report is meant to be a reference manual for a reader that is interested in learning more about technologies and methods for semantic information management. Even though some chapters may require previous knowledge to understand all details, we believe that the reader lacking this knowledge still can get at least a general understanding of the topics. After each chapter in this report, except the last two, we give a short summary of the highlights found in each chapter. We, the authors, hope that the reader enjoys this report and find it a rewarding experience.

In the first chapter we introduce the report and describe the vision of information systems we have had when writing this report. Here the scope and reading instructions are also given.

In the second chapter we begin from the end user's perspective and discuss what happens when querying for information. We describe information retrieval metrics, nomenclature and the fundamental structuring methods used when indexing and recollecting unstructured information. `MapReduce` is also introduced to the reader as it is one of the more successful and widely deployed information retrieval algorithms of today.

Since it is difficult to manually retrieve information from vast amounts of documents, we would like to employ the help of computers. This requires some kind of structuring (at least syntactical) of the information. In the third chapter we thus discuss structured languages, their impact on the semantic web and their internal hierarchy. The main languages presented in this chapter are; XML, RDF and OWL.

However, syntactical structuring is often not enough, and more semantic information is needed. *Ontologies* describe concepts that exist in a domain and how these relate to each other. This is the topic of the fourth chapter. In an ontology all concepts have a formal definition to make them interpretable by computer software. This makes it also easier to exchange information, reuse domain knowledge and even perhaps make it possible to infer new facts.

In the fifth chapter we go further in the discussion on how to represent knowledge in a way that it can be used to draw new conclusions i.e. reason about it. One of the more widely used formalisms for knowledge representation languages is *Description Logics* (DL). It is, for example, the basis of OWL. DL is actually a family of languages with different levels of expressiveness. Related to this, and also discussed in the chapter, is the *knowledge base* which is the collection of rules and statements that we wish to use for reasoning.

A language's reasoning capabilities is decided by its expressiveness i.e. the types of concepts, instances, relationships and restrictions that can be modeled with it. In the sixth chapter we discuss reasoning engines and algorithms. The aim of the chapter is to give the reader an understanding of what reasoning is, what a reasoner engine does, and what its defining features are. The supported levels of logic expressiveness decides whether the reasoner can guarantee *decidability* and *termination*. Decidability means whether an arbitrary statement

can be said to be true in the context of a particular knowledge base and termination refers to whether this answer is possible to obtain in finite time or not. We will also briefly discuss two of the most popular reasoning algorithms, i.e. Tableau and RETE, as well as present a reasoner comparison of some of the more popular reasoners, e.g., Pellet and JESS.

A knowledge base is another word for information repository. In the seventh chapter we focus on semantic data storage and querying. Three different approaches for efficiently storing semantic data are introduced: the triple store approach, modeling to relational database systems and vertical partitioning. A comparison of these three approaches shows, that their performance is largely dependent on the type of queries that are asked, i.e. if we have a need for explicit or implicit information. SPARQL has been identified as the de-facto standard for querying RDF data. The chapter finishes with a short introduction to different semantic storage software packages, such as, Jena, Sesame, Mulgara and Oracle.

Throughout the chapters we give indications that semantic techniques are not always purely beneficial and therefore, in chapter eight, these challenges are discussed in more detail. It is important that developers are aware of those problems and that they consciously decide whether their applications or parts of their applications need semantic support. For example, for high performance applications it is generally advisable to use established systems like relational databases instead. A good thing to remember is that the technologies and methods mentioned in this chapter are still in development and some are not yet as mature as established ones. Also, no critical mass of users has adopted them yet. It is to be expected that, when more people start to use semantic technologies some of the challenges presented in the report will be resolved.

In the final chapter we present the project that has sponsored this report. We discuss its focus and conclude with some of the research challenges that the project aims to tackle in future. The reader that would like to contact us is welcome to do so at `infoMoS [at] foi.se`

Contents

1 Introduction

Most organizations collect vast amounts of data over time, and they usually store it in an unstructured form such as text documents, power point presentations, excel files, etc. To manually search though this kind of information is usually not very efficient and therefore we try to use the aid of computers for information management. Computer software can be used to help us navigate, search and even reason about what we know. But, if we want machines to be able to help us, we need to encode our knowledge in a structured and formal way.

The knowledge within documents, often available as natural language text, is normally not available in a formally structured form, thus making it hard for machines to understand the meaning. Therefore they have a limited ability to do anything useful with the information.

Popular search tools like Google can execute a search query very fast but it is not always the case that the search result contains what you are looking for. Furthermore, the result set may not be presented in the most suitable format. Traditionally, the search result for a query like *"Which countries are members of the European Union and have a population greater than 10 million"* would be a result set of documents that hopefully contains the answer. To get the final result further processing of the documents is required.

If this type of knowledge (i.e. the answer to the question) is structured and formally encoded, a search engine may answer the question with a list of countries directly. Normally this information is explicitly stored (e.g. in a relational database), but what happens if we wish to retrieve implicit information? For example, if two persons have the same father and mother, then we would like to know that they are siblings without having to explicitly state this fact.

Today, information and content management systems often use a relational database as storage. Relational databases rely on a *"static"* schema (one that does not change easily) which needs to be updated when a new type of document is introduced or a new attribute is added. We need to find technologies that let us expand our information model with our new data, a so called dynamic model.

1.1 Vision

We believe that new technologies and methods, such as semantic technologies, can be used to realise our vision of a future information system. Our vision is that an information system should contain the following features:

- It should be easy for both man and machine to find relevant information. The result of a search should be able to be presented in a suitable form such as graphs, tables, timelines, diagrams etc. where applicable.

- It should be easy to find related information.
- The system should be able to infer new facts and reveal complex relationships.
- The system should be adaptive to changes so that a new type of document or a new concept can easily be introduced.
- Data, information and knowledge should be stored in a formal way so that it can be understood by both man and machine.
- The system should be scalable.
- The system should promote collaboration.
- The system should utilize new information about how the users actually use the system.

To build systems that have all the features described in the vision, many technologies from different areas needs to be used. Ideas and technologies from research areas such as Semantic Web, Information Retrieval, Knowledge Management, Text Mining, Database Management may be suitable. No technique alone can solve the problem.

1.2 Scope

In this report we describe some of the key technologies that the next generation systems are likely to be based upon. We also point out some of the challenges that arise. The scope of this report is to give an introduction to information management methods and technologies. We will talk about the semantic web and its enabling technologies. Later we will discuss the general concept of ontologies and semantic data storage as well as querying semantic data. We will also talk about reasoning and information retrieval.

The report aims to answer the following questions:

- Why do we structure information?
- How can we structure information?
- How do we store the structured information?
- How do we reason using the stored information?
- When should we reason about the information?
- How do we retrieve the needed information?
- What are the challenges with semantic technologies?

1.3 Reading Instructions

The report aims at giving an introduction to information system technologies and methods for semantic information management. Intended readers are people interested in learning about information management methods and technologies. The report requires

The structuring of the report should allow a reader to read each chapter individually in no special order. However it is recommended to start reading from the first chapter to the end chapter since we sometimes refer back to earlier chapters. At the end of each chapter, except Chapters 1, 8 & 9, there is an individual summary describing the chapter and the main issues raised in it. This is a good starting point for the reader who does not have the time to read the entire chapter.

Chapter 2 (Information Retrieval): Describes methods for clustering and retrieving unstructured information. It is a good starting point since it gives an introduction to efficiency measures used when evaluating information systems.

Chapter 3 (Structural Languages): Describes what the Semantic Web is, which languages are involved and how it evolved.

Chapter 4 (Ontologies): Describes the ontology concept, methodologies and practical tools for ontology building.

Chapter 5 (Knowledge representation and Reasoning Languages): Describes knowledge representation formalisms and languages. It gives an introduction to the expressiveness of different language classes and their respective computational issues.

Chapter 6 (Reasoning Engines and Algorithms): Gives a glance at different reasoning technologies, that is, how we can draw logical conclusions from given facts. The chapter also compares the implementations of different reasoning engines.

Chapter 7 (Semantic Data Storage and Querying): Describes how semantic data can be stored and talks about scalability concerns when building large semantic information systems. It also describes different semantic query languages.

Chapter 8 (Challenges): Describes challenges that can occur when using semantic systems and discusses some possible solutions.

Chapter 9 (Research Project): Presents the research project behind this report, its focus and the research challenges that will be investigated in future.

For the interested reader; the bibliography at the end of this report gives pointers for where to find more information about the discussed technologies and methods.

2 Information Retrieval

Information Retrieval (IR) is the science of searching for unstructured documents, for information within documents, and for metadata about documents [54]. When comparing different information retrieval technologies it is important to have a common method to measure the results, and thus know why one technology is better or worse than another. We can e.g. use IR methods to compare semantic technologies to ordinary index-based technologies used by search engines.

In this chapter we discuss what happens when querying for information and present IR metrics, nomenclature, and fundamental structuring methods used when indexing and recollecting unstructured information.

2.1 Nomenclature

IR is an interdisciplinary field combining diverse topics such as; mathematics, computer science, linguistics and statistics. The focus is to study and develop efficient information models, i.e. meta data procedures which yield a high hit rate for different query methods.

When talking about information systems we must also talk about different indexing methods and their impact upon query evaluation. This impact is captured via analyzing different efficiency metrics related to these systems. To have a common interpretation and language on the nature of unstructured data we often refer to the following entities or data concepts:

Index: An index is any data structure which improves the performance of lookup. There are many different data structures used for this purpose, and in fact a substantial proportion of the field of Computer Science is devoted to the design and analysis of index data structures. There are complex design trade-offs involving lookup performance, index size, and index update performance [53].

Collection: Defines the multiset[1] of documents upon which we operate. For example, Google's collection would be almost the entire world wide web.

Document: The smallest retrievable content entity. The query result set, i.e. our *"hits"*, can only encompass references to this entity.

Corpus: Is a set of terms collected as an operative entity i.e., an entity we might internally produce and hold indexing information for. Corpora might be split up over multiple nodes to increase throughput.

[1] In mathematics a set requires that its members are unique whereas a multiset allows for duplicates.

Term: The smallest entity upon which we operate. This can be defined as many words, single words or parts of words. This is usually defined as a single word, by which we mean a varied sized literal string prefixed and suffixed by white space.

These (except *Index*) concepts are of course related to each other in an hierarchical fashion where a collection of document may hold many different corpora which in turn may hold many different terms.

2.2 Metrics

IR is best explained by studying the difference between relevant and retrieved documents. In Figure 2.1 we see that when we search with a query we could end up with lots of retrieved documents. Some may be relevant, and some irrelevant. In most search engines it is unlikely that we would retrieve exactly all relevant documents. Somehow we would like to measure the relationship between relevant and irrelevant documents. In order to do so we need metrics and more detailed information about document collections.

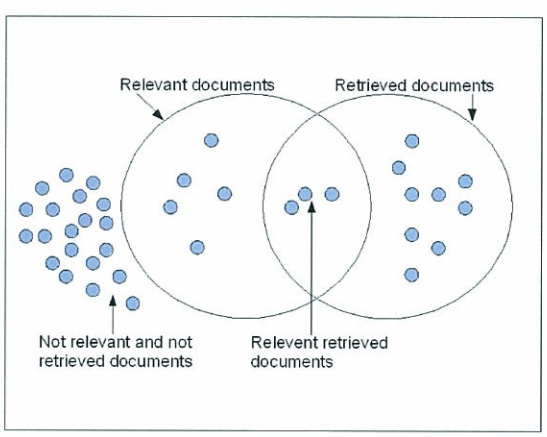


Figure 2.1: Retrieved and relevant documents.

Figure 2.2 is used as a frame of reference for the symbols used in the representation of the different metrics. We define efficiency metrics to analyze the impact of different models and queries upon different collections. The following measurements are of interest [56]:

Fall-Out: Measures how many of the non relevant documents in the collection we retrieve for a specific query. Given the sets from Figure 2.2 we get the fall out as a ratio defined as

$$FO(\alpha, \beta, \gamma, \delta) = \frac{\delta}{\alpha + \beta + \gamma + \delta}$$

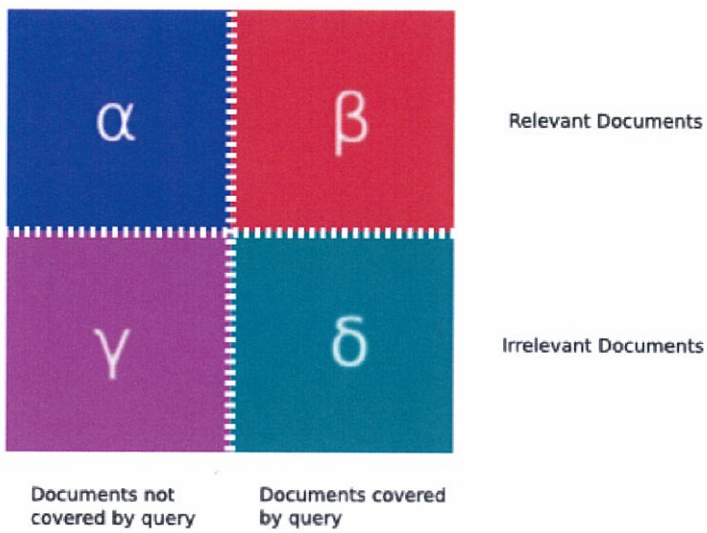


Figure 2.2: Information retrieval categories over a document collection.

Precision: A fraction of the relevant documents in the query result set to the number of retrieved documents, i.e. the fraction of good *"hits"*.

$$Pr(\beta, \delta) = \frac{\beta}{\beta + \delta}$$

With perfect precision the query retrieves no irrelevant documents from the collection and with non perfect precision we retrieve some documents which are irrelevant.

Recall: The fraction of relevant documents which were in the query result set to the number of relevant documents, i.e. how many of the relevant documents we found.

$$Re(\alpha, \beta) = \frac{\beta}{\alpha + \beta}$$

With perfect recall the query retrieves all of the relevant documents in the collection and with non perfect recall some documents which are relevant are left in the collection. Since both precision and recall are orthogonal a high recall does not guarantee that a specific query does not return any bad hits, just that all relevant hits were found.

F-Measure: Also called *"Pythagorean Harmonic Mean"* in euclidean math or *"F1 Score"* in statistics. It is used to calculate a combined *"Precision"* and *"Recall"* measurement. Since *"Precision"* and *"Recall"* by their nature give measurement upon the same query set and are orthogonal their

mean must be treated accordingly.

$$FM(\alpha, \beta, \delta) = \frac{\beta(2\alpha + \beta + \delta)}{2(\alpha + \beta)(\alpha + \delta)}$$

This measurement is hard to develop an intuitive feel for. It should simply be seen as a metric to cover the correct combined measure of both the precision and recall aspects for an IR system.

2.3 Indexing Models

In order to be able to search through different types of unstructured documents we need to generate an index (a model) for the vast collections of data. This can be done in different ways, where some technologies are good and some less good.

There is a plethora of models, i.e. classification methodologies or indexing mechanisms, within information retrieval. An information system might implement one or many of the different indexing mechanisms.

2.3.1 Boolean

The boolean model holds an index table with all unique terms in the collection [58]. This implies a fairly large index since the naive solution would be to hold a matrix with documents on one axis and terms on the other.

The matrix representation is also not very space, or memory, efficient. A collection of 10 documents containing 20000 unique words[2] would yield a matrix with 200000 cells

A typical matrix would look like:

	The Bible	On the Origin of Species	Horses	...
breed	1	1	1	...
destroy	1	1	0	...
development	0	1	1	...
murder	1	0	0	...
whore	1	0	0	...
slaughter	1	0	0	...
species	0	1	1	...
struggle	1	1	0	...
survival	0	1	0	...
⋮	⋮	⋮	⋮	⋱

This is called a *"Term-Document incidence matrix"*. This data model allows us to do boolean query evaluation. A typical boolean query might look like

[2]Average number of words known by a typical Englishman. And, Yes he wrote all 10 documents.

"slaughter or *murder* and *breed"*. Using the above section of the data model would yield [The Bible] whereas the query *"struggle* or *survival"* would yield [The Bible, On the Origin of Species]. We denote a list of items with square brackets, i.e. [List Item_0, List Item_1, ...].

The evaluation is performed using boolean algebra, since we have a bit vector for each term. Constructing the query *"murder* or *struggle* and not *development"* would be performed by joining the bit vectors for murder and struggle, complementing the vector for development and joining the result, i.e.

```
10 | 11 & !01 = 10 [The Bible]
```

One problem with this model is that a document is either in or out of a query set. What we are looking for is a system which can tell us the difference, or rather, a metric distance between competing documents.

2.3.2 Inverted Index

Looking at the data model above, one realizes that there is typically some skewness in the distribution of terms. Cursory inspection would suggest that *"The Bible"* contains far more violence than *"On the Origin of Species"*. Other works are likely to show the same behaviour, i.e. a medical text will contain terms not typically found in a motorcycle manual and vice versa, since these belong to different domains and cultures. The incidence matrix will therefore become inherently sparse as more documents enter the collection. As such the matrix is not very space efficient, as implied earlier, and it is also a tedious task to increase and expand it.

Instead it is common to store references to occurring terms mainly by keeping an ordered list of documents for each occurring term. A typical record is denoted by:

```
Record :: Term -> [DocumentID]
```

An inverted index consists of a dictionary, the set of all unique words, and a postings list [55]. This data structure is faster, since we can look directly at a terms postings list and it does not require us to maintain long bit vectors when we locate new terms or add new documents. This is exemplified in Table 2.1

We still have a problem with this model: we can't relate query results so we have no way to exclude uninteresting results from the set. Since we cannot relate results it may very well be that, perhaps in a result set of a hundred thousand documents, the most relevant result is presented last. Browsing through such a collection is not an alternative. If there was a rank for each hit, relative to the query, we could present only the relevant hits.

Dictionary		Postings Lists			
breed	→	0	1	2	...
destroy	→	0	1		...
development	→	1	2	...	
⋮					

Table 2.1: A typical inverted index. Where 0 = *"The Bible"*, 1 = *"On the Origin of Species"* and 2 = *"Horse Breeding"*.

2.3.3 Vector

An even better method, than inverted index, to relate query results with would be to record each term encounter in the postings list. In this way we could count the number of times a term frequents a document [60]. This time the representation for a record is given as:

```
Record ::  Term -> [(DocumentID,TermIndex)]
```

Where each term collects a list of document, term index tuples. The term index is the position of the term in the document, or corpus. This is exemplified in Table 2.2

Dictionary		Postings Lists					
breed	→	(0,77)	(0,2344)	(1,135)	(1,140)	...	
destroy	→	(0,65)	(0,67)	...	(1,564)	(1,567)	...
development	→	(1,42)	(1,64)	(1,68)	(1,78)	(1,79)	...
⋮							

Table 2.2: We can now order documents after their relevance to a query. A query for *"breed* or *development"* will rank *"On the Origin of Species"* higher than *"The Bible"* since it contains far more breeding and development than the latter.

2.3.3.1 Term Frequency

A document's term frequency, $tf_{i,j}$, is the number of times a term, t_i, occurs in a document, D_j [59]. At a first glance one might assume that a high term frequency would indicate a more relevant document. It is, however, the case

that a document which contains a hundred occurrences of a term whereas another only one, does not necessarily indicate that the first document is a hundred times more relevant.

In order to remedy this, the $tf_{i,j}$ is normalized to the number of other term occurrences within a document. For any given term t_i and any given document d_j, the normalized term frequency, $TF_{i,j}$, is defined as the number of that occurrence, i.e. $tf_{i,j}$, divided by the sum of the total number, k, of term occurrences within the document, i.e. $\sum_{k=1}^{|D_j|} tf_{k,j}$.

$$TF_{i,j} = \frac{tf_{i,j}}{\sum_{k=1}^{|D_j|} tf_{k,j}}$$

Normalization is sometimes done by only taking the sum of the most frequent terms. It should be clear that when we refer to term frequency we refer to the normalized TF.

2.3.3.2 Inverse Document Frequency

A problem however, is that many languages contain words which occur quite often and are likely to be irrelevant to a query, e.g., in English the words *"the, and, is,"* etc. An inverse document frequency (IDF) [60] is added as a weight to differentiate interesting keywords from those which occur often and are non relevant, like the term *"the"*. As such it is effectively used to measure the importance of a single term. IDF can be defined as some weight which suppresses frequently used terms.

Formally, IDF is defined as the logarithm of the number of documents in a collection divided by the number of documents which contain the term.

$$IDF_i = log\frac{N}{n_i}$$

So if a term occurs in all 10 documents in a collection we get a weight of 0 effectively eliminating the term from further evaluation[3]. If the term instead only occurs in one document we get amplification of that term[4]. For a more in-depth analysis of IDF see [37].

2.3.3.3 Winning combination?

The combined weight given from multiplying TF with IDF filters out many non-relevant terms. Since the weight for a given term is maximized by obtaining a

[3] since $\log 10/10 = \log 1 = 0$.
[4] since $\log 10/1 \simeq 2.3$.

high term frequency within a document while not being present in many other documents. This is commonly referred to as $tf - idf$ or TF*IDF.

2.4 MapReduce

`MapReduce` is an algorithm for indexing and recollecting documents in large sets of data. The algorithm was recently popularized by Google [11].

The baseline for `MapReduce` is an inverted index, as described earlier, of terms mapping to documents on which the algorithm is run. As such it strictly operates on key-value pairs. Despite its name `MapReduce` consists of three major operations, `Map-Group-Reduce`. We will ignore `Group` and assume that `Map` is combined with `Group`. The programming model then becomes:

`Map` takes data from one domain structured in a key-value pair, i.e. a tuple (key, value), and `Map`:s it to a new pair in another domain consisting of a key and a list of values:

```
Map :: (k1,v1) -> (k2,[v2])
```

`Reduce` takes the output of `Map` and groups the newly found pairs with the same key and `Reduce`:s that result into a single value for each unique key:

```
Reduce :: (k2,[v2]) -> (k2,v3)
```

Both need to be implemented by the user for any specific purpose. One of the benefits of deploying a `MapReduce` is that both functions are required to be pure and as such may be parallelized and distributed [25]. This means that it scales well on an increasing number of nodes.

In order to be more concrete consider the task of counting the number of times each term is encountered in a collection of documents. For this instance we get:

`Map` processes the documents in the collection, produces a list and groups the document identifiers belonging to each term. This produces:

```
[(Term,DocumentID)] -> [(Term,[DocumentID])]
```

`Reduce` groups all document identifiers for a specific term and then `Reduce`:s the list to a word count:

```
[(Term,[DocumentID])] -> [(Term,WordCount)]
```

The strength of the `MapReduce` algorithm resides not from the fact that the combination of `Map` and `Reduce` is beneficial, but in the fact that both functions are purely functional [11].

2.5 Summary

This chapter discusses fundamental structuring methods when indexing and recollecting unstructured information. It is focused on introducing the reader to IR in order to give a better understanding of involved issues when developing scalable information systems and when querying for information in unstructured documents.

The chapter describes a basic nomenclature and the different units and metrics commonly used in Information Retrieval (IR). Examples of simple measurement metrics used in evaluation of queries are *"Precision"* and *"Recall"*. The first indicates how relevant a query result set is i.e. if many irrelevant documents were returned the result set has not been very precise. The second term indicates how good a search has been, i.e. how many of the relevant documents a query recollects.

Then we discuss how basic retrieval models are created and how to reason about their operation. The inefficiencies of the boolean indexing model, both in terms of an exhaustive query result set but also in its in-memory representation, are presented. We also discuss the benefits of the vector based indexing model. This model tries to rate documents in relation to some quality measurement, e.g. the frequency of interesting words.

Further, we introduce the reader to one of the more successful and deployed IR algorithms as of today i.e. `MapReduce`. It is an algorithm used in large sets of data for indexing and recollecting documents.

3 Structural Languages

In the previous chapter we saw how indexing and querying is handled when we have unstructured data. If we wish to become more efficient and be able to retrieve richer and more precise information we need to start pre-processing the information.

In recent years the World Wide Web has started to evolve from a source of plain facts to an intelligent information source. This development is driven by the vision of a *"Semantic Web"* [26], where not only humans but also machines can understand and process the information presented in the Web. For realizing this vision, new technologies are being developed that facilitate the structuring, presentation and interpretation of information. In this chapter we will present the essential technologies that the Semantic Web is built upon, including for example XML, RDF and OWL. The knowledge presented in this chapter is a basis for the understanding of the rest of this document.

3.1 Language Stack

The advancement of the Semantic Web is steered by the World Wide Web Consortium [51] (W3C). The W3C is responsible for the standardization of matured Semantic Web technologies, but also gives recommendations for the usage of promising new technologies.

Figure 3.1 shows the Semantic Web Language stack [66]. It can be seen, that the technologies of the Semantic Web form a hierarchy; technologies from the upper layers build upon the basic, lower-layer technologies. The stack is still evolving and the top layers contain some technologies that have not yet been standardized and some which are only ideas. The bottom layers contain technologies (URI, Unicode, XML) that are used in the classical hypertext web[1]. The middle layer technologies (RDF, RDFS, OWL, SPARQL) are standardized technologies used for enabling Semantic Web applications.

3.2 XML - eXtensible Markup Language

XML is a language for structuring data[2]. It is widely used on the World Wide Web and has been considered a standard for many years [68]. Because of the wide adoption of XML, parsers and libraries for manipulating XML data are available for most programming languages and platforms. As can be seen in Figure 3.1, XML is the basic foundation of the Semantic Web and all other technologies build on it.

[1] Note that the Semantic Web works as an extension of the classical hypertext web, not a replacement.

[2] Data structuring does not imply logical structuring.

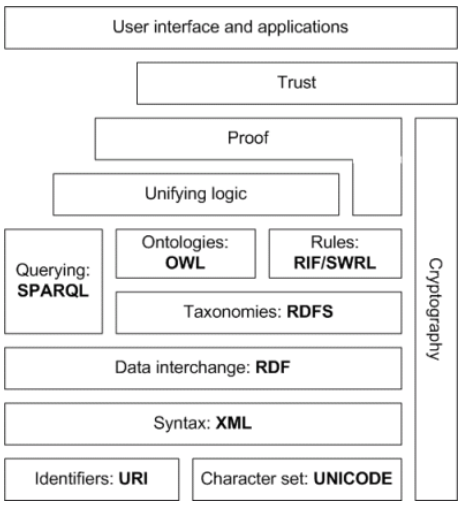


Figure 3.1: Semantic Web Language stack.

XML divides information into markup and content. Markup is the means to add structure to the data, by annotating the actual content with hints on how to process and interpret it. Figure 3.2 shows an example XML document. Markup is encoded in XML in so-called elements[3] enclosing the actual content, for example *<"book">...content...<"/book">*. Attributes like *"currency=SEK"* can be used to further specify the content. In order to uniquely name the entities an XML namespace, *"xmlns="http://www.example.org/bookShop"*, is used. This is done in order to avoid ambiguity between identically named elements or attributes in different XML documents [69].

```
<?xml version="1.0" encoding='UTF-8'?>
<book xmlns="http://www.example.org/bookShop">
  <title>On the Origin of Species</title>
  <author>Charles Darwin</author>
  <price currency="SEK">95</price>
</book>
```

Figure 3.2: XML example.

The main advantage of XML is the division between markup and content, that allows to present information without having to care about how it is actually processed in an application. Further advantages are that it allows hierarchical structuring of data and is easy to validate. Forward and backward

[3]Elements are also called tags.

compatibility between different XML versions is easy to achieve [68]. XML is also platform independent. The major disadvantage of XML is its data redundancy. It can be seen in Figure 3.2, that the content only makes up about half of the document, while the rest is markup. This problem will be picked up again later on in Chapter 7, when we talk about the efficient storage of semantic data, and in Chapter 8, where we discuss some open challenges.

3.3 XML Schema

While XML is used to structure information, an XML schema is a description of the structure of an XML document. The schema specifies for example which type of elements the document contains, how many of them and their data types [71]. The parser/validator uses the XML schema to check if syntax and other constraints are fulfilled in the XML document. If two documents use the same schema, then this implies that they must *"play by the same rules"*. There are different languages available to express XML schemas, e.g. DTD (Document Type Definition), RELAX NG, XML Schema, the two latter being more expressive [71]. XML Schema is a W3C recommendation and currently the most commonly used XML schema language.

Figure 3.3 shows an example of an XML Schema document. In this document a data structure for books is defined. If we compare this document to the XML document in Figure 3.2, we can see that our XML document conforms to the data structure defined in the XML Schema document.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.example.org/bookShop">

 <xsd:element name="book">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:element name="title" type="xsd:string"/>
       <xsd:element name="author" type="xsd:string"/>
       <xsd:element>
         <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="price" type="xsd:double"/>
          </xsd:sequence>
          <xsd:attribute name="currency" type="xsd:string"/>
         </xsd:complexType>
        </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>

</xsd:schema>
```

Figure 3.3: XML Schema example.

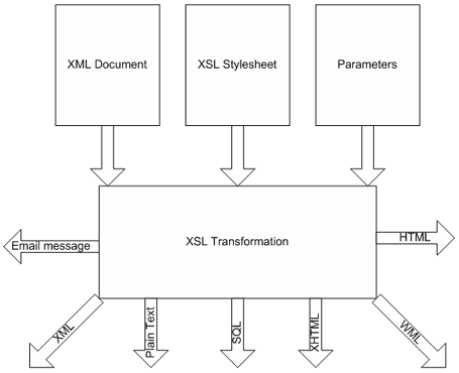


Figure 3.4: The eXtensible Stylesheet Language Family Transformations.

3.4 XSLT - The eXtensible Stylesheet Language Family Transformations

XSLT is a language for transforming XML documents. A so-called XSLT stylesheet transforms the XML tree from the source document into an output document in the desired output format [62]. Patterns are used for matching different parts of the source tree, extracting those parts and storing the final output. XSLT uses XPath [70] (see also Section 7.2.2) for selecting different parts of an XML document for processing. The selection can of course be conditional.

Figure 3.4 shows how XSLT can be used for creating different types of output [52]. The input is the source XML document and an XSLT stylesheet with parameters. The output that is produced is basically a *"subset"* of the input, allowing the data to be used and expressed in other formats, e.g. SQL, a new XML document, HTML, XHTML or plain text.

3.5 RDF - Resource Description Framework

RDF is a general method for the conceptual description or modeling of information in the Semantic Web [65]. RDF is based on the idea to make statements about resources. Statements are in the form of triples, consisting of subject, predicate and object. For example, consider the following statement: *"the cat jumps on Bob"*. We can break it down to its atoms, which are: *"the cat"* = subject, *"jumps on"* = predicate and *"Bob"* = object. By creating statements and breaking them down to their atoms we can make human knowledge available for machines in a formal way. If machines can understand human knowledge, then they can e.g. process it many times faster than us. Machines can also help us discover new knowledge from an existing data set.

There are different ways to express RDF statements. The most common is through the XML notation, but there is also other ways, e.g. by using Notation3 [65]. Because the statements can be considered as resources, RDF uses URIs to identify them. The URIs basically provide a grouping ability for statements, putting them in context.

RDF data is visualized through graphs. The graphs contain nodes which represent subjects and object. The predicates among them are visualized by lines, indicating the relationships. Figure 3.5 shows an example of RDF and Figure 3.6 shows the graphical representation of this example. It can be seen that the document contains a subject, the horse *"Mary"*, connected to an object, the rider *"John"*, by the predicate *"hasRider"*. Further statements are made for both *"Mary"* and *"John"*. The data in RDF documents can be queried through different query languages, e.g. SPARQL, Versa and RDQL. Querying semantic data will be described further in Chapter 7.

```
<?xml version="1.0"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:ns="http://www.example.org/horses/">

   <ns:Rider rdf:about="http://www.example.org/#john">
      <ns:hasAge>53</ns:hasAge>
      <ns:livesIn>Utah</ns:livesIn>
   </ns:Rider>

   <ns:Horse rdf:about="http://www.example.org/#mary">
      <ns:hasRider rdf:resource="http://www.example.org/#john"/>
      <ns:hasColor>black</ns:hasColor>
   </ns:Horse>

</rdf:RDF>
```

Figure 3.5: RDF example.

3.6 RDF Schema and OWL - Web Ontology Language

An ontology is a formal representation of a set of concepts within a domain and the relationships between those concepts [64]. An ontology provides the vocabulary needed to build a domain containing different concepts and relationships [64]. In Chapter 4 we will look more deeply into the ontology concept.

RDF Schema is a basic ontology language providing the means to define classes and the relationships between them. RDF Schema is similar to XML Schema in the sense that it describes the constructs that can be used in an RDF document. The RDF Schema language is not very expressive, for example it is not possible to set cardinalities on the relationships between classes (*"Each horse can only have one rider"*).

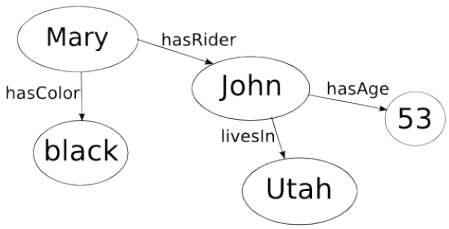


Figure 3.6: Graph for the RDF example.

The Web Ontology Language OWL is a more expressive ontology language, partly based on RDF Schema. For example, it contains support for cardinalities, hierarchical properties and capabilities of properties (e.g. transitive, symmetric). OWL has three dialects, with different expressivity: OWL-Lite for simple classifications, OWL-DL based on Description Logics (see Chapter 5) and OWL-Full with very high complexity.

Figure 3.7 shows a simple example of OWL. Here we create a class *"Person"* and two subclasses of this class: *"AmericanCitizen"* and *"SwedishCitizen"*. We also create two properties *"hasPersonalNumber"* (an ID number applicable only to Swedes) and *"hasSocialSecurityNumber"* (an ID applicable only to Americans). If we then create two persons: a Swede named Ola and an American named Jennifer, we could infer that Jennifer does not have a personal number. In Chapter 6 we will talk more about reasoning and the methods behind it.

3.7 SWRL - Semantic Web Rule Language

SWRL is a proposal for a Semantic Web rule language [47]. It is a high-level abstract syntax for Horn-like rules [63]. The rules created are used in order to infer information that is not explicitly stated (which is the case in a relational database). For example, you can have a rule that says: *"if person A and person B both have mother C and father D, then they are siblings"*. In a relational database you would need explicitly to state that A and B are siblings, thus storing more data.

SWRL combines OWL (DL + Lite) and RuleML subset (Unary and Binary Datalog) [47]. There are different SWRL implementations available (see also the reasoner comparison in Chapter 6), here are some examples [67]:

- SWRLTab is an extension to Protege that supports editing and execution of SWRL rules.
- R2ML (REWERSE Rule Markup Language) supports SWRL.
- Hoolet, an implementation of an OWL-DL reasoner that uses a first order prover supports SWRL.

```
<rdf:RDF xmlns="http://www.example.org/people.owl#"
     xml:base="http://www.example.org/people.owl"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:people="http://www.example.org/people.owl#">
    <owl:Ontology rdf:about=""/>

    <owl:Class rdf:about="#Person"/>

    <owl:Class rdf:about="#AmericanCitizen">
        <rdfs:subClassOf rdf:resource="#Person"/>
    </owl:Class>

    <owl:Class rdf:about="#SwedishCitizen">
        <rdfs:subClassOf rdf:resource="#Person"/>
    </owl:Class>

    <owl:DatatypeProperty rdf:about="#hasPersonalNumber">
        <rdfs:domain rdf:resource="#SwedishCitizen"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>

    <owl:DatatypeProperty rdf:about="#hasSocialSecurityNumber">
        <rdfs:domain rdf:resource="#AmericanCitizen"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>
</rdf:RDF>
```

Figure 3.7: OWL example.

- Pellet, an open-source Java OWL DL reasoner has SWRL-support.
- KAON2 is an infrastructure for managing OWL-DL, SWRL, and F-Logic ontologies.
- RacerPro, supports processing of rules in a SWRL-based syntax by translating them into nRQL rules.

Figure 3.8 shows an example of SWRL rules using the RDF/XML syntax. You can see that there are three variables declared: $x1$, $x2$, $x3$. There are also four different properties: *"hasParent"*, *"hasSibling"*, *"hasSex"* and *"hasUncle"*. The rules create the following relationships: $x1$ *"hasParent"* $x2$ and $x2$ *"hasSibiling"* $x3$ and $x3$ *"hasSex"* *"male"*. Then from this we know that $x1$ *"hasUncle"* $x3$. An example would be if Peter has a parent, e.g. a father named John. If John has a sibling (a male sibling, a brother) named Jonas, then Peter has an uncle (Jonas).

```
<swrl:Variable rdf:ID="x1"/>
<swrl:Variable rdf:ID="x2"/>
<swrl:Variable rdf:ID="x3"/>
<ruleml:Imp>
  <ruleml:body rdf:parseType="Collection">
    <swrl:IndividualPropertyAtom>
      <swrl:propertyPredicate
        rdf:resource="&eg;hasParent"/>
      <swrl:argument1 rdf:resource="#x1" />
      <swrl:argument2 rdf:resource="#x2" />
    </swrl:IndividualPropertyAtom>
    <swrl:IndividualPropertyAtom>
      <swrl:propertyPredicate
        rdf:resource="&eg;hasSibling"/>
      <swrl:argument1 rdf:resource="#x2" />
      <swrl:argument2 rdf:resource="#x3" />
    </swrl:IndividualPropertyAtom>
    <swrl:IndividualPropertyAtom>
      <swrl:propertyPredicate
        rdf:resource="&eg;hasSex"/>
      <swrl:argument1 rdf:resource="#x3" />
      <swrl:argument2 rdf:resource="#male" />
    </swrl:IndividualPropertyAtom>
  </ruleml:body>
  <ruleml:head rdf:parseType="Collection">
    <swrl:IndividualPropertyAtom>
      <swrl:propertyPredicate
        rdf:resource="&eg;hasUncle"/>
      <swrl:argument1 rdf:resource="#x1" />
      <swrl:argument2 rdf:resource="#x3" />
    </swrl:IndividualPropertyAtom>
  </ruleml:head>
</ruleml:Imp>
```

Figure 3.8: SWRL example.

3.8 Summary

In the previous chapter we saw different information retrieval techniques that can be used when dealing with unstructured information. The problem is that machines do not really understand the information or comprehend what is important and what is not. Hence, if we wish to be able to retrieve richer and more precise information we need to start pre-processing the information and thus indicate how the information should be interpreted. In this chapter we talked about the Semantic Web and the different technologies which work as a foundation for it.

The *"Semantic Web"* is based on technologies such as XML, RDF and OWL that help represent data in a machine readable way. This allows us to process knowledge faster and also gain new knowledge from it. XML is an enabling syntax technology for the Semantic Web and is also used by RDF (Resource

Description Framework) to describe objects, relationships and properties. With RDF we can make statements about our world like *"Anna owns a Horse"* and *"John has rider Anna"*. Even though RDF is useful in many situations it is not very expressive and does, for example, not allow expressing restrictions.

OWL is a web-based ontology language serialized mainly in the form of RDF and XML. An ontology is a domain description providing the vocabulary needed to build a domain containing different concepts and relationships. It is much more expressive than RDF and comes in different expressiveness flavors. As we are able to express more and more information it quickly becomes tedious to state every fact and therefore we would like to be able to infer implicitly stated information[4]. SWRL is a proposal for a Semantic Web rule language in order to create rules which are used to reason around ontologies and draw new facts.

[4]In Chapters 5 and 6 we will discuss inferencing and reasoners further.

4 Ontologies

The rapid growth of the Internet and the huge amount of information available today has increased the need for technologies and tools that can be used to organize and structure information. Ontologies may play an important role when trying to address this problem. This chapter describes what an ontology is, construction methodologies and practical tools that can be used.

4.1 What is an Ontology?

Ontologies describe knowledge about a certain domain by specifying concepts, relations between concepts, and axioms. In [42] an ontology is defined as:

> *Ontologies are explicit formal specifications of a shared conceptualization.*

The terms are explained as follows:

Explicit: The type of concepts used, and the constraints on their use are explicitly defined.

Formal: The ontology should be machine readable.

Shared: An ontology captures consensual knowledge, that is, it is not private to some individual but accepted by a group.

Conceptualization: A conceptualization is an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon.

Another attempt to define what an ontology is resulted in

> *An ontology may take a variety of forms, but necessarily it will include a vocabulary of terms and some specification of their meaning. This includes definitions and an indication of how concepts are inter-related which collectively impose a structure on the domain and constrain the possible interpretations of terms. An ontology is virtually always the manifestation of a shared understanding of a domain that is agreed between a number of agents. Such agreement facilitates accurate and effective communication of meaning, which in turn leads to other benefits such as inter-operability, reuse and sharing* [46]

Figure 4.1 shows the relationship that may exist between two concepts, person and pet, defined in an ontology. A person may have a dog as a pet, but

the opposite relation may not be valid. The example is trivial but captures what an ontology construction is all about, namely identifying concepts and the kind of relationships that may exist between entities.

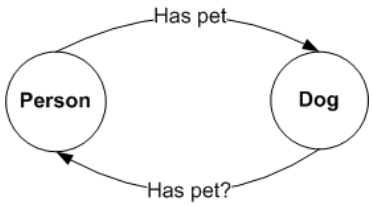


Figure 4.1: An ontology defines which kind of relationships can exist between two concepts. A person can have a dog as a pet, but the opposite relation may not be valid.

According to [27] the following requirements must be fulfilled in order for something to be considered an ontology:

- A controlled (extensible) vocabulary.
- Unambiguous interpretation of classes and term relationships.
- Strict hierarchical subclass relationships between classes.

A controlled vocabulary means that there is a finite set of terms and that every term has an explicit definition. Formal definitions of terms make it possible for machines to interpret the information in a correct manner. The relationship between terms should also have explicit and unambiguous definitions. In practice, this means that relationships must be defined using a formal language interpretable by a machine.

Strict hierarchical subclass relationships are necessary for type deduction. If A is a superclass of B then the relationship between A and B is said to be strictly hierarchical if an instance x of type B entails that instance x is also of type A. A hierarchy where elements do not follow this kind of "is-a" structure can consequently not be said to have a strict hierarchical subclass relationship between classes. A book about Physics might be placed in a category named Science. It is wrong to say that a book about Physics is a Science, consequently is not a strict is-a hierarchy. The hierarchy is a valid is-a hierarchy if the category changed named to "Science book".

Figure 4.2 shows an example class hierarchy from the PROTON ontology developed by Ontotext [33]. The ontology is used in many applications such as the KIM semantic annotation platform [2]. A screenshot showing the relations available for persons is available in Figure 4.3.

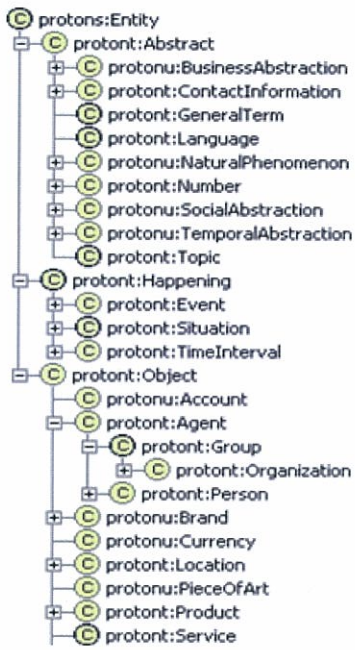


Figure 4.2: Ontology class hierarchy used by the KIM annotation platform.

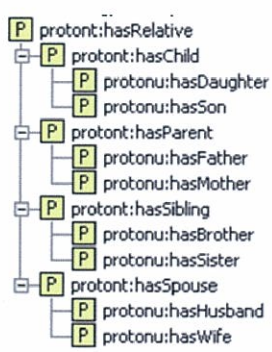


Figure 4.3: The screenshot shows a part of the relation hierarchy used by the KIM annotation platform.

4.2 Classifying Ontologies

There are different kinds of ontologies. Among else, the expressiveness and generality may differ. In [27] the following list of ontology features is presented. The list can be used to classify ontologies with respect to expressiveness.

Controlled vocabulary: List of terms.

Thesaurus: Information describing relationships between terms such as synonyms.

Informal taxonomy: A hierarchy exists but inheritance is not strict.

Formal taxonomy: Hierarchy with strict inheritance.

Frames (classes): Classes with sets of properties that can be inherited are supported.

Value restrictions: Property values can be restricted.

Limited logic constraints: Values of properties can be restricted by logic rules.

First Order Logic (FOL) constraints: Very expressive constraints with an extended support for logic rules. FOL with detailed restrictions such as disjoint classes or transitive relationships is supported.

Figure 4.4 provides a graphical representation of the list above list. Classifications of some well-known ontologies are included in the figure. Ontologies with a high level of expressiveness fall under the category heavy-weight ontologies whereas less expressive ontologies fall under the category light-weight ontologies.

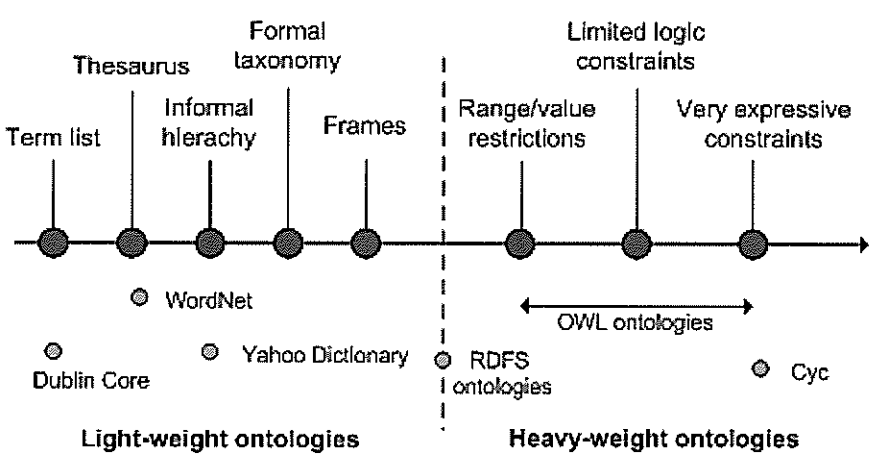


Figure 4.4: Classification of ontologies according to their expressiveness (adopted from [3]).

Ontologies can be classified based on their generality [18] as shown in Figure 4.5.

Top-level ontology: Describes very general concept like space, time, matter, object, event, action etc. Independent of a particular problem or domain.

Domain & Task ontology: Describe the vocabulary related to a generic domain (medicine, cars) or a activity (selling).

Application ontology: Depends on both domain and task ontologies. These concepts often correspond to roles played by domain entities while performing a certain activity.

In order for an application to be useful in practice, an ontology must cover the terms of the specific domain/problem to be solved. A too general ontology might not cover all concepts needed but can act as candidate to use as a top-level ontology.

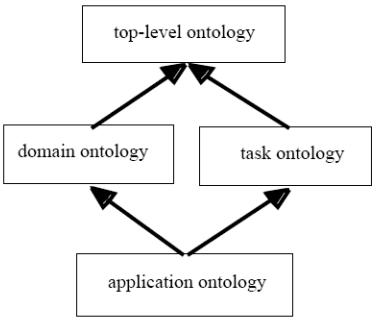


Figure 4.5: Ontology classifications according to Guarino [18].

4.3 Why Develop an Ontology?

The benefits of using ontologies can be summarized as follows:

- It provides a shared understanding about concepts in a domain by formal definitions.
- It provides a way to reuse knowledge about the domain.
- It provides a way to encode knowledge in a way that is understandable by both man and machine.

A shared understanding about concepts and formal definitions makes it easier to exchange information and reuse domain knowledge. Both man and

machine benefits from this. By encoding knowledge, machines may get a better understanding of how the information can be used and may even be able to infer new facts. Potential benefits of using ontologies are explained in further detail in the following subsections.

4.3.1 Reuse of Knowledge and Information Sharing

Ontologies can improve information exchange by reducing the possibility of misinterpretation. Axioms and rules defined in an ontology can be used to find contradicting information. However, it is not always necessary to agree on and share all the details of an ontology in order to benefit from it. It is often enough to agree on some top-level concepts to be able to exchange information. By agreeing on the structure of information among people and software, information can more easily be extracted and aggregated from different sources.

Ontologies does not only make it easier to exchange information, it may also be easier to reuse knowledge. If a public ontology already exist which covers the concepts needed it can be reused with ease. One example of concepts that can be reused is time and space since there is often a need to be able to make statements about time-intervals and points in space.

A goal when developing an ontology might be to make domain assumptions explicit. If domain assumptions hard coded in a programming language are moved to an ontology, the assumptions become more clear and manageable [32].

The formal specifications of terms in an ontology makes it easier to analyze domain knowledge. A formal description of terms also is of great help if you want to find out if you can reuse existing ontologies [27].

4.3.2 Moving from Unstructured to Structured Data

Ontologies can be used to transform unstructured information to structured information in various ways. It can be used to add valuable metadata both on document and content level. The latter is known as semantic annotation. Another possible application where an ontology can be used is in relation extraction from text. If an entity extraction algorithm is able to find names of persons and locations an ontology can be used to give suggestions of what relationship may exist between the entities.

4.3.3 Improved Search

An ontology can be used t, improve search capabilites by find equivalent words describing the same concept in a search query and hopefully by exploiting this get more search hits. This is known as parallelism. Ontologies can also be used to generalize search terms. A result set for a search on boats can include both sailing boats and motorboats since both of the terms are more specific variants of boats.

Since each term has a specific meaning, an ontology based information

system may answer very precise search queries. In theory, an ontology based information system would be able to answer a search query like *"How many people live in Sweden"* with the actual number instead of links to documents.

4.3.4 Revealing Implicit Facts

The formal specification of terms and relationships can be used to infer new statements. Consider the following example.

The following is specified in an ontology:

- The relation $sonOf$ is defined as a sub-relation to the more general $bloodRelativeTo$.
- The relation $fatherTo$ is defined as a sub-relation to the more general $bloodRelativeTo$.
- The relation $bloodRelativeTo$ is specified to be transitive[1].
- The $bloodRelative$ relation is only valid between humans. In other words, we set the domain and range of the relation to be a class $Human$.

The two following statements are added:

- John is $sonOf$ Edward.
- Edward is $fatherTo$ Lisa.

By utilizing the formal definitions of terms and relationships defined in the ontology the following facts can be inferred:

- John and Lisa are blood relatives since $bloodRelativeTo$ is a transitive relation.
- Lisa and John are humans since the relation are only valid between instances of $Human$.

Figure 4.6 provides a graphical representation of the example. As seen in the example, new facts can be inferred that must be true based on the new information and the rules defined in the ontology. Principally, if one or more facts are added to the knowledge base, it is likely that this information can be used to infer new facts. In the example, two facts were added and we got two for free. The fundamental problem that ontologies can be used to solve is how to make the most use of the available information. The example shows that ontologies can not only be used to improve the structure of information, they can also be used to reveal implicit facts.

[1] A *"transitive"* property asserts that if the property P exists between instances x and y, and between y and z, then the property also exists between x and z.

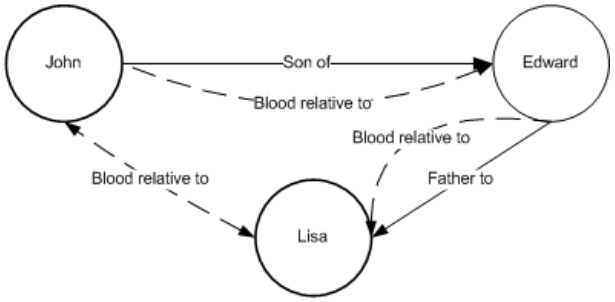


Figure 4.6: New facts can be inberred based on a transitive relationships. Implicit facts are represented as dashed lines.

4.4 Methodologies for creating an evaluating Ontologies

Several methodologies for evaluating existing ontologies. These are [10, 14, 32, 23]. One thing that they all have in common, is that they all point out the importance of defining what the ontology should be used for. They also point out that before one starts to build an ontology one should consider reusing the work of others. There are many public ontologies that can be extended in order to match the requirements of a specific domain or context.

The methodologies provide guidelines on how to identify concepts and relations. However, before identifying concepts and relations, we need to know what kind of problem that the ontology is supposed to solve. Does the application, let's say a movie recommendation system, require a definition of a car? Most of the methodologies also provide guidelines on how to validate the ontology. In [14] the authors point out that a methodology for creating ontologies should not only provide guidelines for how to identify concepts and relations, it should also take the life cycle into consideration. In most cases, an ontology should be seen as a natural part of the system that it is used in and be documented as such. As an software engineering, it is hard to get it right the first time. Even if a lot of research has been done in beforehand, it is hard to cover all things from start.

Regardless of which approach is chosen, one should keep the following fundamental rules in mind [32]:

1. There is no single correct way to model a domain.

2. Ontology development is an iterative process.

3. Concepts in the ontology should be close to objects (physical or logical) and relationships in your domain of interest. They are most likely to be nouns (objects) or verbs (relationships) in sentences that describe your domain.

4.4.1 Motivating Scenarios

Many well-known methodologies agree that it is important to motivate the existence of the ontology and provide scenarios and/or use-cases where the ontology is needed.

Common questions to answer when determining the role of an ontology in an application are:

- Why do we need it?
- Why not use existing ontologies?
- How will it be used?
- Who are the targeted users?

A useful technique introduced by [17] and adopted by other methodologies are the use of so called competency questions. A competency questions is a question that a system, by utilizing the ontology, should be able to answer. In a movie recommendation system a competency question might be formulated as: Which action movies have won an Oscar. Competency questions can be grouped by importance. Some may be a requirement and some just nice to have.

4.4.2 Identifying Terms and Concepts

The next step, after having motivated the need for an ontology, is to think about what concepts need to be introduced and what kind of relationships that may exist. The first step is often to organize and structure the knowledge about the domain. The following steps are suggested by [14]:

1. Create a glossary of terms; natural language definition, synonyms, acronyms.
2. Create a taxonomy; identify classes and subclasses.
3. Create a concept dictionary; specify relations that can be used.
4. Specify rules; specify rules that can be used for inference.

Different methodologies propose different strategies how to identify and specify how concepts relate to each other.

A bottom up approach starts with defining the most specific classes. The next step is to find more general terms. An ontology built with a bottom up approach may result in a very high level of detail, which may increase overall effort. Using this approach it may be difficult to spot commonalities between related concepts which can increase the risk of inconsistency [45].

A top-down approach starts with the definition of the most general concepts in the domain and subsequent specialization of the concepts. Consider a Wine

ontology. In a top down approach one would start with creating the Wine and Food classes, then you specialize the wine class by creating some of its subclasses like White wine, Red wine etc.

A combined approach starts with important concepts first, and then tries to find more general and specialized concepts. [45] argues that a combined strategy produces an ontology with the best balance in terms of the level of detail.

Finally, the following rule of thumb can be helpful when deciding if the ontology definition is complete.

> *The ontology should not contain all the possible information about the domain: you do not need to specialize (or generalize) more than you need for your application (at most one extra level each way).* [32].

4.4.3 Evaluation

An important step when creating an ontology is to find out if the ontology meets the specified requirements. Important things to consider when evaluating an ontology is:

Consistency: Can any contradictory knowledge be inferred?

Completeness: Is all that is supposed to be in the ontology explicitly stated or can it be inferred?

Conciseness: Does it include unnecessary definitions?

Finally, if one adopts the idea of competency questions, the system should be able to answer these.

4.5 Ontology Editors

There are many tools available today which can be used for creating, visualizating and manipulating ontologies. Most of them can visualize ontology information like the class/relationship hierarchy. An important feature supported by many tools is the possibility to use a reasoner in order to infer statements.

The most widely used open-source ontology developing tool is Protégé. Protégé has a large set of available plugins. Protégé can be used with Pellet, FACT++ and any DIG compliant reasoner. A reasoner is a piece of software that is able to infer new facts from a set of know facts or axioms. A screenshot of Protégé is shown in Figure 4.7.

Another application, originally developed by MIND lab at University of Maryland, is SWOOP. It is at current date available as an open-source project.

SWOOP has support for reasoning via the OWL-DL reasoner Pellet. A screenshot of SWOOP is shown in Figure 4.8.

A commercial alternative is TopBraid Composer developed by TopQuadrant. TopBraid Composer is implemented as an Eclipse plugin in and has support for reasoning. TopBraid Composer integrates reasoners such as OWLIM, Pellet and Jena Rules. A screenshot of TopBraid Composer is shown in Figure 4.9.

The online web-based ontology editor Knoodl has a wiki approach. It emphasizes the collaborative aspect of ontology engineering. Another web-based application is OwlSight developed by Clark & Parsia. As of today OwlSight can only be used to view an ontology and it has no editing capabilities. OwlSight integrates with Pellet and can consequently infer implicit statements.

UML can also be used to model a domain. In [23] the authors discuss the possibilities and benefits of using UML to construct ontologies since UML is more widespread than other ontology languages such as OWL. A UML file can then automatically be transformed to an OWL file using a transformation algorithm.

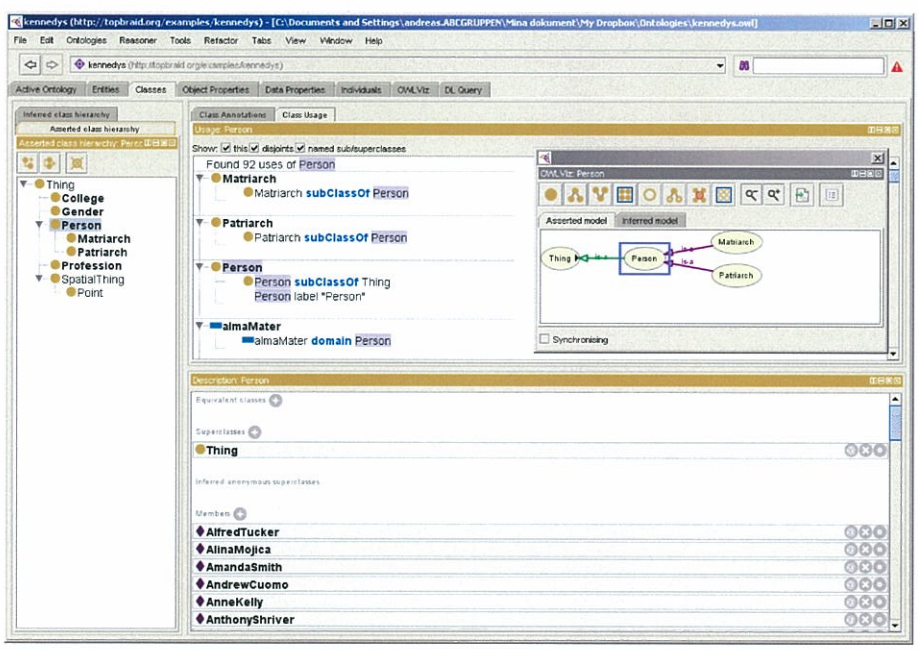


Figure 4.7: A screenshot of the Protégé user interface.

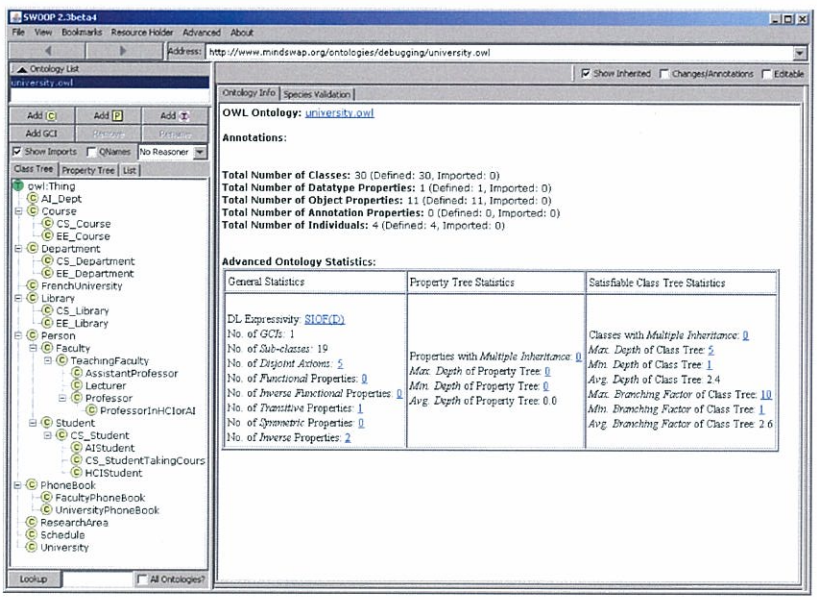


Figure 4.8: A screenshot of SWOOP user interface.

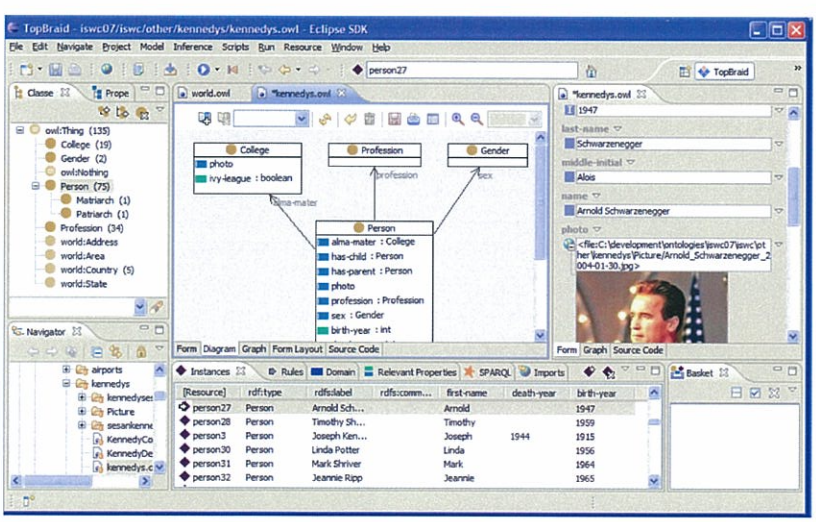


Figure 4.9: A screenshot of TobBraid Composer user interface.

4.6 Summary

An ontology describes concepts that exist in a domain and how these relate to each other. Each concept have a formal definition which makes it interpretable by computer software. A shared understanding of concepts and formal definitions makes it easier to exchange information and reuse domain knowledge. Both man and machine benefit from this. By encoding knowledge, machines can get a better understanding on how the information can be used and may even be able to infer new facts.

Ontology construction is a research area of its own and several methodologies exist. It is recommended that you follow a well established methodology since the chances of missing some important aspect decreases. The most important thing is perhaps to involve domain experts and end-users during the construction phase. One good practice is to formulate scenarios and/or use-cases where the ontology is needed. Another useful technique that can be utilized when developing an ontology is to formulate competency questions. A competency question is a question that a system, by utilizing the ontology, should be able to answer.

5 Knowledge representation and Reasoning Languages

In order to achieve intelligent behavior in a system it needs to have knowledge, a way to represent it and reason about it. In essence, the goal of knowledge representation is to represent knowledge in a way so it can be used to draw new conclusions. This chapter presents the formalisms that outline the expressiveness of a language and some of the more commonly used reasoning languages. By formalism we mean a way to abstract the information using different symbols and rules previously defined and therefore known.

5.1 Knowledge Representation Formalism

To represent knowledge we need a language that not only consists of a syntax, with which to model knowledge, but that also can be used to reason with. The reasoning capability is decided by its expressiveness, i.e. the types of concepts, relationships and restrictions that can be modeled. Let's take a look at these modeling building blocks.

Classes: *"Classes"* (or *"concepts"*) are interpreted as sets that contain individuals and they may be organized into a superclass-subclass hierarchy (a taxonomy). Subclasses specialize (*"are subsumed by"*) their superclasses.

Individuals: *"Individuals"* (or *"instances"*) can be referred to as being *"instances of classes"*.

Properties: *"Properties"* are binary relations on individuals. Properties are also known as slots (in Protégé/Frames), as roles in description logics and relations in UML and other object oriented notions. In other formalisms they are called attributes. Properties can also be organized in hierarchies where a property can be a more specialized variant of another property.

Properties are usually divided into three different types: data, object and annotation properties.

- Data type properties link an individual to an XML-Schema data type value or an RDF literal, Figure 5.1 shows an example of a data property.
- Object type properties link an individual to an individual and can be called a binary relationship.
- Annotation properties can be used to add metadata to classes, individuals and object/data type properties.

```
<owl:DatatypeProperty rdf:ID="currency">
   <rdfs:domain rdf:resource="#Money"/>
   <rdfs:range
    rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
```

Figure 5.1: Data type property example

Object properties can be further classified into different types; functional, inverse functional, transitive and symmetric.

- A *"functional"* property is a property that can have only one (unique) value i for each instance x, i.e. there cannot be two distinct values i_1 and i_2 such that the pairs (x, i_1) and (x, i_2) are both instances of this property. For example, lets have the concept person that can have a functional property called *"sex"* with the value male or female. We thus do not allow a person x to have both the property value i_1 male and i_2 female.
- An *"inverse functional"* property asserts that a property value i can only have the value for a single instance x, i.e. there cannot be two distinct instances x_1 and x_2 such that both pairs (x_1, i) and (x_2, i) exist. For example, in Sweden each person is given a personal identity number with a unique value i. There cannot be two persons that have the same personal identity number. For those who have knowledge in relational databases, an inverse-functional property resembles the notion of a unique key in a table.
- A *"transitive"* property asserts that if the property P exists between instances x and y, and between y and z, then the property also exists between x and z. For example, let's say that we have the property $ancestor$. If x is an $ancestorOf(y)$, and y is an $ancestorOf(z)$, then x will also be an $ancestorOf(z)$.
- A *"symmetric"* property is a property that if the pair (x, y) is true, then the pair (y, x) is also true. For example, lets assume that x has the property $isSiblingOf(y)$, then y will thus also have the property $isSiblingOf(x)$.

Properties are used to create restrictions on the individuals that belong to a class. Restrictions can be of three main types: quantifier, cardinality and value.

- Quantifier (or *"Value"*) restrictions, see Figure 5.2 for an example.
  - The existential quantifier $\exists$ can be read as *"at least one"*, or *"some"*.

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#hasParent" />
  <owl:hasValue rdf:resource="#Clinton" />
</owl:Restriction>
```

Figure 5.2: Value restriction example.

  - The universal quantifier $\forall$ can be read as *"all values from"*.

- Cardinality restrictions. The number of relationships that an individual may participate in for a given property. Figure 5.3 shows an example of this.
  - Minimum cardinality restrictions ($\leq$).
  - Maximum cardinality restrictions ($\geq$).
  - Cardinality restrictions($=$).

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#hasParent" />
  <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">2
  </owl:minCardinality>
</owl:Restriction>
```

Figure 5.3: Cardinality restriction example.

5.2 Knowledge Representation Languages

In this section we will present some of the more common knowledge representation languages and give a short historical overview. We will pay special attention to description logics since it is the foundation for semantic web reasoning.

5.2.1 Semantic Networks

In the 1950's *"Semantic Nets"* for computers were introduced by Richard H. Richens. They were meant to be used as an *"interlingua"* for machine translation of natural languages.

A semantic network is a network that represents semantic relations between concepts. It can be viewed as a directed or undirected graph consisting of vertices, representing concepts, and edges representing relations, see Figure 5.4. It is often used as a form of knowledge representation but cannot really be called a knowledge representation language since it is not ruled based.

An example of a semantic network is Wordnet [72]. It is a large, lexical database that groups English words into sets of synonyms, provides short

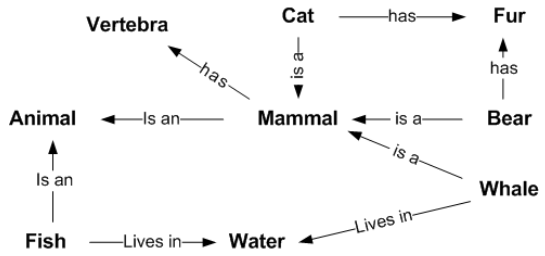


Figure 5.4: Example of a semantic network from [57].

general definitions and records the various semantic relations between these sets. Some of the most common semantic relations defined are meronymy[1], holonymy[2], hyponymy[3], hypernymy[4], synonymy[5] and antonymy[6].

5.2.2 Frame Language

A frame is a data structure used for knowledge representation. It was introduced by Marvin Minsky in the 1970:s when working in the field of Artificial intelligence [28].

Frames can be viewed as chunks of information, which usually contain properties called attributes or slots. Each frame has several kinds of information attached to it.

> *Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is about what to do if these expectations are not confirmed. We can think of a frame as a network of nodes and relations. The "top levels" of a frame are fixed, and represent things that are always true about the supposed situation. The lower levels have many terminals - "slots" that must be filled by specific instances or data. Each terminal can specify conditions its assignments must meet. (The assignments themselves are usually smaller "sub-frames.")* [28]

Thus the frames usually contain data and object type properties (called attributes or slots) together with different types of restrictions.

Two well known examples of frame- or frame-based- languages are OIL and F-logic.

[1]Meronymy: A is part of B.
[2]Holonymy: B is part of A.
[3]Hyponymy: A is subordinate of B.
[4]Hypernymy: A is superordinate of B.
[5]Synonym: A denotes the same as B.
[6]Antonym: A denotes the opposite of B.

5.2.3 Description Logic

Description logic (DL) was designed as an extension to semantic networks and frames (which were not equipped with formal logic-based semantics) and got its name in the 1980's (previously it was called *"terminological systems"*, and *"concept languages"*). DL is not only a knowledge representation language but a family of them, where each has different levels of expressiveness. They have many application areas but are perhaps best known as the basis for commonly used ontology languages such as OIL[7], DAML+OIL and OWL [19].

A DL language has formal semantics and is a decidable fragment of first-order logic (FOL) with reasonable expressive power.

5.2.3.1 DL Operators and Expressivity

Each description logic language has a label, e.g. $\mathcal{SHIQ}$, that follows an informal naming convention[8]. The label roughly describes the allowed operators and the expressivity of the language. Here follows a list of them [61] (for explanations of the meaning of the formalisms, see Section 5.1). Note that the purpose of this section is only to give the reader an orientation in the terms. These DL operator terms are of interest when we discuss the expressivity of reasoners in Chapter 6:

- $\mathcal{F}$ Functional properties.
- $\mathcal{E}$ Full existential quantification (Existential restrictions that have fillers other than $owl:thing$).
- $\mathcal{U}$ Concept union.
- $\mathcal{C}$ Complex concept negation.
- $\mathcal{S}$ An abbreviation for $\mathcal{ALC}$[9] with transitive roles.
- $\mathcal{H}$ Role hierarchy (subproperties - $rdfs:subPropertyOf$).
- $\mathcal{R}$ Limited complex role inclusion axioms; reflexivity and irreflexivity; role disjointness.

[7] OIL has its basis in both Frame Languages and DL.

[8] The naming conventions aren't fully systematic. The letters might be permuted so that the logic $\mathcal{ALCOIN}$ can also be referred to as $\mathcal{ALCNIO}$ or $\mathcal{SNIO}$.

[9] The base language is $\mathcal{AL}$ (attributive language). Which allows:

- Atomic negation (negation of concepts that do not appear on the left hand side of axioms).
- Concept intersection.
- Universal restrictions.
- Limited existential quantification.

$\mathcal{ALC}$ is $\mathcal{AL}$ but with negation of any concept allowed, not just atomic concepts. $\mathcal{ALC}$ is also the equivalent of $\mathcal{ALUE}$ but is used much more often.

- $\mathcal{O}$ Nominals. (Enumerated classes of object value restrictions - $owl : oneOf$, $owl : hasValue$).
- $\mathcal{I}$ Inverse properties.
- $\mathcal{N}$ Cardinality restrictions ($owl : Cardinality$, $owl : MaxCardinality$).
- $\mathcal{Q}$ Qualified cardinality restrictions (available in OWL 1.1, cardinality restrictions that have fillers other than $owl : thing$).
- ($\mathcal{D}$) Use of data type properties, data values or data types.

5.2.4 Reasoning Languages for the Semantic Web (OWL)

In Chapter 3 we reviewed the language stack and discussed OWL. We saw that there are three flavors, Lite, DL and Full, but we didn't really discuss what it means. The three are sublanguages of each other where each successively obtains more and more expressive powers.

- OWL Lite is suitable for situations where only a simple class hierarchy and simple constraints are needed e.g. thesauri. OWL-Lite is based on $\mathcal{SHIF}^{(\mathcal{D})}$.
- OWL DL is as the name suggests based on DL. Since it is based on $\mathcal{SHOIN}^{(\mathcal{D})}$ it is suitable for automated reasoning. It is possible to automatically compute the classification hierarchy (subsumption) and check for inconsistencies in an ontology that conforms to OWL-DL[10].
- OWL Full is not actually a sublanguage [50]. OWL Full contains all the OWL language constructs and provides free, unconstrained use of RDF constructs. It is used where very high expressiveness is more important than being able to guarantee the decidability or computational completeness of the language. It is not possible to perform automated reasoning on OWL Full ontologies since it contains all the OWL language constructs and provides free, unconstrained use of RDF constructs. For example, in OWL Full the resource $owl : Class$ is equivalent to $rdfs : Class$, which is different from OWL DL and OWL Lite, where $owl : Class$ is a proper subclass of $rdfs : Class$[11].

5.2.4.1 OWL 2

At the time of writing W3C has produced a new OWL recommendation called OWL 2 has more formal syntax and semantics [48]. The primary exchange syntax for OWL 2 is RDF/XML and the semantics resemble that of DL as

[10] The popular ontology editor Protege supports $\mathcal{SHOIN}^{(\mathcal{D})}$.
[11] This implies that not all RDF classes are OWL classes in OWL DL and OWL Lite.

OWL 2 provides the expressiveness of $\mathcal{SROIQ}^{(\mathcal{D})}$. It has three *"profiles"* i.e. sublanguages (EL, QL and RL [49]), which somewhat resemble the intended purposes of OWL Lite, DL and Full.

- OWL 2 EL is intended for ontologies that contain very large numbers of properties and/or classes. The EL acronym reflects the profile's basis in the EL Description Logics family[12].
- OWL 2 QL is aimed at applications that use very large volumes of instance data, and where query answering is the most important reasoning task. The QL acronym reflects the fact that query answering in this profile can be implemented by rewriting queries into a standard relational Query Language e.g. SQL.
- OWL 2 RL is aimed at scalable reasoning without sacrificing too much expressive power. OWL 2 RL reasoning systems can be implemented using rule-based reasoning engines[13]. The RL acronym reflects the fact that reasoning in this profile can be implemented using a standard Rule Language e.g. SWRL.

5.3 Summary

In this chapter we have discussed different languages used for reasoning and formalisms for knowledge representation. A formalism is a structured set of rules with wich we can explain a domain. One of the goals of knowledge representation is to represent knowledge in such a way that it can be used to draw new conclusions i.e. reason about it. To be able to do this the language we use needs to have a formal syntax and semantics. The reasoning capabilities of a language is decided by its expressiveness i.e. the types of concepts, instances, relationships and restrictions that can be modeled with it.

We also discussed some of the more common knowledge representation languages like Description Logics (DL) and the reasoning Languages for the Semantic Web i.e. OWL. DL has its roots in Semantic nets and Frames and is a decidable fragment of First Order Logic. DL is actually a family of languages with a varying degree of expressive power.

W3C has produced a new OWL recommendation called OWL 2 that is more formal. The semantics remind of DL and it has three profiles (sublanguages with different expressive powers) that are aimed at different types of applications with different requirements.

[12] $\mathcal{EL}$ provides intersection and full existential quantification.
[13] For more information on reasoning and reasoning engines, see Chapter 6.

6 Reasoning Engines and Algorithms

Reasoning is the operation that we perform when we draw logical conclusions from given facts (i.e. deductive reasoning). Nowadays there are a multitude of software that help us reason when we have large numbers of facts gathered in knowledge bases. In this chapter we will first see what a knowledge base is and then review some of the currently available reasoning engines, compare them and look more closely at the more common reasoning algorithms and the advantages and disadvantages of each.

6.1 Knowledge Base

A (machine readable) knowledge base is an information repository, usually denoted by KB or Δ. The information is stored for the purpose of having automated deductive reasoning applied to it and thus it has to be structured in a logically consistent manner. An ontology can define that structure and its statements are divided into two groups TBox and ABox.

A bit simplified a TBox stands for *"terminological box"* and contains the statements that describe the classes and properties. ABox is an *"assertional box"* and contains the statements associated with instances of the TBox statements. Using a database analogy, the TBox contain the schema and the ABox the data.

ABox statements typically have the form: *"A is an instance of B"* or *"Mary is an Officer"*. While TBox statements are typically of the form: *"All Officers are Persons"* or *"There are two types of Persons: Privates and Officers"*.

Usually, when talking about DL and reasoners there is sometimes also a mention of the RBox, *"role"* or *"property"* box. It "exists" in any DL that has property axioms e.g. *"subPropertyOf"*, *"transitive"*, *"functional"*, etc.

Together all of these statements (ABox, TBox, RBox) make up a knowledge base.

6.2 Reasoning Engines

A reasoner's task is to check the consistency of the knowledge base. Most of the more widely used reasoner engines of today use variations of description logic as a base.

Depending on the algorithm or the type of data in the KB a reasoner processes the KB Boxes separately. This is a way to optimize the reasoning work since certain key inference problems are related to one but not the other ones. For example the *"classification"* task is related to the TBox and the *"instance checking"* task to the ABox. Since consistency and satisfiability checking is done against the TBox its language compliance and structure can greatly affect the performance of these procedures.

A crucial issue when choosing a reasoner is whether one is guaranteed decidability (in logics called satisfiability) or termination in finite time. Depends on the choice of knowledge representation language and its expressiveness. Decidability is whether an arbitrary statement can be said to be true in the context of a particular KB and termination refers to whether this answer is possible to obtain in reasonable time or not. For example First Order Logic is not decidable in general and therefore Description Logics (which is a subset) is used instead.

Reasoning or inferencing commonly proceeds by forward chaining (modus ponens, data-driven) and backward chaining (goal-driven). Modus ponens (MP) is a simple argument form sometimes referred to as *"affirming the antecedent"* or *"the law of detachment"*. It has the form of *"If P then Q"* meaning that if P is true then so is Q (also called a true-false statement). Backward chaining starts with a goal and works backwards from the consequent to the antecedent to see if there is data available that will support any of the consequents. For example, lets have the goal to decide which color Napoleon's pet has, giving it whinnies and eats grass, and that the knowledge base contains the following rules:

1. If X whinnies and eats grass Then X is a horse.
2. If X chirps and sings Then X is a canary.
3. If X is a horse Then X is white.
4. If X is a canary Then X is yellow.

This KB would be searched and the third and fourth rules would be selected, because their consequents (Then X is green, Then X is yellow) match the goal (to determine the pet's color). It is not yet known that the pet is a horse, so both the antecedents (If X is a horse, If X is a canary) are added to the goal list. The KB is searched again and this time the first two rules are selected, because their consequents (Then X is a horse, Then X is a canary) match the new goals that were just added to the list. The antecedent (If X whinnies and eats grass) is known to be true and therefore it can be concluded that X is a horse, and not a canary, and thus it is white.

In the following sections a description of the more common reasoning algorithms are given.

Recently there have also begun to appear probabilistic reasoners[1] which can reason over KBs containing uncertain knowledge. For example they can process statements like *"After rain comes sunshine with a probability greater than 80%"*. To express these kinds of statements Bayesian networks are used.

[1]Examples of probabilistic reasoners: Pei Wang's non-axiomatic reasoning system, Novamente's probabilistic logic network, Pronto - probabilistic Description logic reasoner.

Many reasoners are commercial, open source or available as both but with different levels of functionality or support. There are many synonyms to "reasoning engine" such as "inference engine", "rule engine", "semantic reasoners" or simply "reasoner". The differences are due to the algorithms or the types of data they reason about, but the main functionality is still the same i.e. drawing new logical conclusions from provided data.

In the Section 6.4 a comparison of currently available reasoner implementations is made.

6.3 Reasoning Algorithms

The two most often used reasoning algorithms in today's reasoners are Tableau and RETE, but there are a number of other algorithms in use.

6.3.1 Tableau

The tableau methodology was invented in the 1950's by Beth and Hintikka. It was later perfected by Smullyan and Fitting, and is today one of the most popular proof theoretical methodologies. By the end of 1980 Schmidt-Schaußand Smolka described the first complete tableau-based subsumption[2] algorithm for a non-trivial DL i.e. ALC.

The tableau method is used to determine the satisfiability of finite sets of formulas of various logics in our case statements in a KB. It works by transforming a formula into subformulas until all constraints are satisfied or an obvious contradiction is detected. The transformation is done by following tableau calculus rules. All ABox assertions[3] are viewed as constraints.

6.3.2 RETE

The RETE algorithm was developed by Charles Forgy in the late 1970:s and is designed to sacrifice memory for increased speed. Expert systems like JESS [21] and SOAR use the algorithm today. The name is taken from the latin word for "net".

In a Rete-based system a network is built where each node (except the root) corresponds to a pattern occurring in the left-hand-side (the condition part) of a rule. Remember the statement we made earlier *"If P then Q"*. The path from the root node to a leaf node defines a complete left-hand-side rule. Each node has a memory of facts (i.e. ABox assertions) which satisfy that pattern. When new facts are asserted or modified, they propagate to the nodes that match the pattern and are annotated accordingly. For a given rule to be triggered a fact or combination of facts must cause all of the patterns for the given rule to be

[2]Subsumption is the action of deciding whether a concept is a subclass of something else. For example, if mammal is a subclass of animal, then mammal is considered to be an animal.

[3]An assertion is a true-false statement that is viewed as true, thus is a fact.

satisfied. This also implies that a leaf node is reached.

6.4 Reasoner Comparison

When choosing which reasoners to compare we have chosen those that have had some form of development since 2008. The reasoner list is in no way exhaustive. The information in the tables come from the reasoners own pages. The Tables 6.1 and 6.2 compare the reasoners from a functionality and a feature perspective respectivly.

Here follows an explanation of the columns in the reasoner comparison table:

Reasoner: This is the name of the reasoner.

OWL-DL Entailment: This tells whether the reasoner can infer new OWL assertions from the given data in the knowledge base.

Reasoning expressiveness: This was explained in chapter 5.

Reasoning algorithm: These have been previously explained in this chapter.

Query language support: This tells whether the reasoner supports a query language.

Consistency checking: Tells whether the OWL data are consistent to avoid contradictory data.

Interface Support: The types of interfaces the reasoner supports.

Rule Support: Tells whether the reasoner supports reasoning with rules.

Programming language: The programming language the reasoner has been developed in.

Version: The current reasoner version.

License: The reasoner's license type.

Table 6.1: Reasoner comparison; language support.

Reasoner	**OWL-DL Entailment**	**Supported expressivity for reasoning**	**Reasoning algorithm**	**Query language support**	**Consistency checking**
CEL[4]	Yes	EL+	Tableau	Unknown	Yes
FaCT++[44]	Yes	SROIQ(D)	Tableau	Unknown	Yes
FuzzyDL[41]		Fuzzy SHIF	tableau + Mixed Integer Linear Programming Optimization	Own syntax	
HermiT[29]	Yes	SHIQ with description graphs	Hypertableau	Unknown	Yes
Hoolet[5]	Yes	Unknown	First-order prover	Unknown	Unknown
Jena[43]	No complete reasoner included with standard distribution	varies by reasoner (incomplete for nontrivial description logics)	Rule-based	SPARQL	Incomplete for OWL DL
Jess[21]	No	Horn clause	RETE	Horn clause	Yes
KAON2 [30]	Yes	SHIQ(D)	Resolution & Datalog	SPARQL[4]	Unknown
OWLIM[34]	No	R-entailment	Rule-based	SPARQL	No
Pellet[8]	Yes	SROIQ(D)	Tableau	SPARQL	Yes
RacerPro[24]	Yes	SHIQ(D)	Tableau	nRQL, OWL-QL	Yes
SweetRules[16]	No	Unknown	Rule-based	Unknown	No

[4]queries with variables at predicate positions are currently not supported

Table 6.2: Reasoner comparison; development aspects.

Reasoner	**Interface Support**	**Rule Support**	**Programming language**	**Version**	**License**
CEL	DIG,OWL-API	No	Lisp	1.1.2	Free/ open-source
FaCT++	DIG, OWL-API, Lisp-API	No	C++	1.3.0	Free/ open-source
FuzzyDL		No	Java/C++	Unknown	Free/ open-source
HermiT	KAON2-API	No	Java	1.0	Free/ open-source
Hoolet		Yes (SWRL)		Unknown	Free/ open-source
Jena	DIG	Yes (Own rule format)	Java	2.5.7	Free/ open-source
Jess		Yes (Own rule format)	Prolog-like	7	Non-Free/ closed-source
KAON2	DIG	Yes (SWRL – DL Safe Rules)	Java	Unknown	Free/ closed-source
OWLIM		Yes (Own format)		2.x/3.x	Free/ open-source & Non-Free/ closed-source
Pellet	DIG, OWL-API, Jena	Yes (SWRL – DL Safe Rules)	Java	2.0 RC7	Free/ open-source & Non-Free/ closed-source
RacerPro	DIG, Java-Api, OWL-API, Lisp-API	Yes (SWRL – not fully support SWRL)	Lisp	1.9.2	Non-Free/ closed-source
SweetRules		Yes (SWRL, RuleML, Jess)		2.1	Free/ open-source

6.5 Summary

In this chapter we have discussed reasoning engines and algorithms. The aim of the chapter is to give the reader an understanding of what reasoning is, what a reasoner engine does and what its defining features are.

Reasoning is the process of drawing logical conclusions from a set of given facts. This process is also called inference. The facts are stored in a knowledge base and the reasoning engine's task is to check its consistency. A reasoner uses some variation of logic (Description logic, Horn clause logic, etc) as a knowledge management basis. The expressiveness of the used logic decides whether the reasoner can guarantee decidability and termination. Decidability is whether an arbitrary statement can be said to be true in the context of a particular knowledge base and termination refers to whether this answer is possible to obtain in reasonable time or not.

Two of the most popular reasoning algorithms are Tableau and RETE. Of today's available reasoning engine implementations two good examples of the repective algorithms are Pellet and Jess.

7 Semantic Data Storage and Querying

The vision of a semantic web also entails having to deal with a huge amount of semantic data. Possibly millions or even billions of RDF (see Chapter 3) triples need to be stored in semantic data stores on multiple sites. Such a data store should of course provide an efficient access to the data. And all of this should be fitted to the specifics of semantic data, for example to enable semantic querying and integrated reasoning. This chapter will try to present an overview of problems and solutions for semantic data storages. In Section 7.1, different approaches to efficient storage of RDF data are presented. Section 7.2 deals with the semantic querying of such data storages and different semantic query languages are introduced. An overview of existing solutions for RDF storages follows in Section 7.3. This chapter requires basic knowledge about database management system. An introduction to this area is given in [9].

7.1 Storing RDF Data

In the following different approaches to storing semantic data, or more specifically RDF data, will be presented. Important to remember here is that basic RDF data, especially when compared to typical relational data, is rather unstructured data consisting simply of a collection of triples. This lack of structure is imposed by the very nature of the semantic web: data coming from multiple resources produced by many different people can not be expected to have a common, agreed-upon structure. For higher-level applications of course a need for at least some structure arises and RDF-Schema and OWL are powerful languages to describe the underlying concepts of the semantic data. While any improvements drawn from using such schema knowledge are of course nice to have, the minimal requirement for a successful RDF storage solution is the efficient storage of basic RDF data triples.

7.1.1 Triple Store

	triples	
subject	**predicate**	**object**
book1	writtenBy	author1
book1	publishedBy	publisher1
book2	writtenBy	author2
book2	publishedIn	2001

Table 7.1: Database table for RDF triple store.

triples		
subject	**predicate**	**object**
1	2	3
1	4	5
6	2	7
6	8	9

dictionary	
id	**literal**
1	book1
2	writtenBy
3	author1
4	publishedBy
5	publisher1
6	book2
7	author2
8	publishedIn
9	2001

Table 7.2: Database tables for RDF triple store with dictionary.

The straightforward approach to RDF storing is creating a single *triples* table in a relational database management system. This table has three columns, one each for subject, predicate and object (see Table 7.1). An additional dictionary table can be used to store the literal values for subjects, predicates and objects (see Table 7.2). The *triples* table must then only store references to the dictionary table, avoiding the possible duplicate storage of long literal values.

While the triple store approach offers a very simple and intuitive storing of RDF data, potential performance issues can easily be seen. Even simple queries like *"Give me all books written by author1 in 2001"* involve joining the triples table with itself (see Figure 7.1). Each entry needs to be paired with each of the other entries and each pair must then be tested for whether it fits the query. And more complicated queries will need even more such self-joins. If the triples table is very large, queries like this can put a huge strain on the query processor and especially on the main memory.

```
SELECT t1.subject FROM triples AS t1
WHERE t1.predicate='writtenBy' AND t1.object='author1'
INNER JOIN triples AS t2 ON t1.subject=t2.subject
WHERE t2.predicate='publishedIn' AND t2.object='2001'
```

Figure 7.1: Self joins in RDF triple table.

7.1.2 Mapping onto Relational Database

Another straightforward approach would be to directly model the RDF data into a relational database schema. This could for example mean to model each class in an RDFS schema as a table in the relational database and the relationships between classes as foreign keys between these tables (see Figure 7.2). This approach requires the data to be very structured, as pointed out before,

this can not be assumed for RDF data. In fact, if one has structured data, a *"normal"* relational database approach should be preferable to store such data. This solution does not fit for typical RDF data.

```
<rdfs:Class rdf:ID="author">
  <rdfs:subClassOf rdf:resource="#Thing"/>
</rdfs:Class>
<rdfs:Class rdf:ID="book">
  <rdfs:subClassOf rdf:resource="#Thing"/>
</rdfs:Class>
...
<rdf:Property rdf:ID="writtenBy">
  <rdfs:domain rdf:resource="#book"/>
  <rdfs:range rdf:resource="#Literal"/>
</rdf:Property>
```

book		
id	**name**	**writtenBy**
1	book1	1
2	book2	2

author	
id	**name**
1	author1
2	author2

Figure 7.2: Modeling RDFS to relational database.

7.1.3 Vertical Partitioning

Vertical partitioning has been proposed as another approach for storing RDF data [1]. This approach is inspired by the research of vertical fragmentation of data which has been a recent topic in database research and has been implemented for example in MonetDB [20]. For RDF data it means that for each predicate in the datastore, a table consisting of two columns for subject and object is set up (see Table 7.3). These tables contain all subject-object pairs linked through the respective predicate. The data is thereby much more distributed than in the triple store approach, leading to smaller tables and therefore less performance issues when joining over the tables. However this statement only holds when the predicate is known for a query. If it is a predicate that is queried for a subject/object pair, a join over all predicate tables in the system may be necessary.

writtenBy	
subject	**object**
book1	author1
book2	author2

publishedBy	
subject	**object**
book1	publisher1

publishedIn	
subject	**object**
book2	2001

Table 7.3: Database tables for vertical partitioning.

7.1.4 Performance Comparison

In [38] the triple store and the vertical partitioning approaches are benchmarked and compared with modeling the data in a traditional relational database system (non RDF). The benchmark uses a set of queries specifically chosen to show the weaknesses and strong sides of the different approaches. The authors show that the traditional approach outperforms both forms of RDF stores in orders of magnitudes, e.g. queries taking 1s in the relational database usually take 100 to 1000 seconds in the RDF stores. When comparing the triple store with the vertical partitioning, it can be seen that both approaches produce very similar results on average. The triple store has slightly better results for some queries, the vertical partitioning for other queries. Because of these performance issues of RDF storages, it is important during the application design to carefully consider, whether the interoperability and openness of semantic data stores are really needed, or if a conventional relational database design would be a better fit. One thing can be noted from this test though, and that is, that querying for implicit information is not possible in the traditional solutions. When this is a requirement a semantic data store has to be the choice[1].

7.2 Semantic Query Languages

This section introduces different approaches for querying semantic data. First a short overview of query languages will be given. Since many of the proposed semantic query languages are built upon the design principles and syntax of SQL, this section will focus mainly on giving a short introduction to SQL. Afterwards different semantic query languages will be described, including examples showing how to use these languages practically.

7.2.1 Introduction to Query Languages

Relational database management systems have been present since the 1970s and are today one of the most widely used approaches to database management. Relational database managemant systems present data in relations, meaning collections of tables. An in-depth presentation of database management systems is out of scope of this document, interested readers are pointed to read

[1] We have a proposal for a scalable RDF store based on MapReduce at FOI called SDR[15].

further, for example in [13].

SQL (Structured Query Language) is the standard query language for relational databases, based on the relational algebra. The relational algebra provides a subset of the expressiveness of first-order logic. At its center, six operators are defined (examples are given based on the table structure of Figure 7.2):

The selection operator: $\sigma_c(R)$ selects tuples from a relation R for which the condition C holds, e.g. $\sigma_{name='book1'}(book)$.

The projection operator: $\Pi_{a_1 \ldots a_n}(R)$ selects attributes $a_1 \ldots a_n$ from a relation R, e.g. $\Pi_{writtenBy}(book)$.

The Cartesian product: $R \times S$ is the direct product of the relations R and S, e.g. $Book \times Author$.

Set union: $R \cup S$ is the union of two set-compatible relations R and S, e.g. $bookTable_1 \cup bookTable_2$.

Set difference: $R \backslash S$ is the difference of two set-compatible relations $R \wedge S$, e.g. $bookTable_1 \backslash booktable_2$.

Rename: $p_{\frac{a}{b}}(R)$ is used to avoid duplicate attribute names in the result set, e.g. $p_{name/authorname}(author)$.

With these six operators, all other query operators, for example joins, can be expressed. The following expression retrieves the names of books and the authors that wrote them in one single table with two columns aname (authorname) and bname (bookname):

$$\Pi_{aname,bname}(\sigma_{aname=bname}((p_{name/aname}author)\ x\ (p_{name/bname}book)))$$

The SQL language defines keywords for phrasing database queries using these operators. Figure 7.3 gives an exemplary SQL query for the above expression. It is important to note that every SQL query always returns the result in a relation, allowing nesting subqueries in SQL. Also it should not be forgotten that the SQL standard does not only define data manipulation (query and updating) functionality, but also a Data Definition Language to set up database schema and a Data Control Language to control the access to data. However only the data manipulation sublanguage is relevant for the following sections.

As has been already addressed at the beginning of this section, other approaches to database management systems often provide a query language very similar to SQL. Object orientend database management, for example, represents information as objects, basically extending an object-oriented programming language with, amongst others, persistency and query functionalities. The Object Query Language (OQL) is an attempt to define a query language

```
SELECT author.name AS aname, book.name AS bname
FROM author, name
WHERE author.name=book.name
```

Figure 7.3: Simple SQL query.

very similar to SQL for accessing object oriented databases; opposed to the native access based on the pointers between the objects. A reason for the need to define such a query language can probably be found in the huge success of SQL and the programmers' familiarity with it.

The following sections will now define query languages developed specifically for handling semantic data. Much of the concepts and syntax presented of SQL in this section will be seen again there.

7.2.2 XPath

RDF files can be serialized for example as XML documents. A naive approach to semantic data querying would be to simply employ XML query languages to query the XML tree describing RDF data. XPath is a W3C standard language for selecting nodes from an XML tree. In [12] XPath is described in detail. One exemplary XPath query to select the name from a book could be *"//book//name"*. This query selects all nodes 'name' that are descendants of nodes 'book'. It can easily be seen that using XPath to query RDF documents implies that a specific serialization into a specific XML tree is necessary. However one RDF document could have many possible serialization. The obvious conclusion is that XPath querying is not the right solution for semantic data.

7.2.3 SPARQL

As was pointed out in the previous section, the syntactic level querying of XPath should not be used for semantic data. Instead, structure level querying seems more promising. Instead of using the actual document (or rather serialization), structure level querying is based on the graph representation of RDF triples, whatever their serialization. This approach is in fact used by most of the currently common RDF query languages. Two sub-approaches can be identified here, an SQL-based approach that is employed by SPARQL (SPARQL Protocol And RDF Query Language) to be described in this section and the XPath approached employed by Versa described in the following one.

SPARQL is the official W3C recommendation for RDF querying ([39]). Its syntax is very similar to SQL. Just as in SQL, the results of a SPARQL query can be modeled as relations. SPARQL has emerged as a de-facto standard for RDF querying and is supported by all major semantic database solutions. Further work to extend the capabilities of SPARQL is ongoing.

```
PREFIX lit: <http://ex.org/literature>
SELECT ?book,?author
WHERE { ?book lit:writtenBy ?author.
        ?book lit:publishedIn "2001" }
```

Result	
book	**author**
book2	author2

Figure 7.4: SPARQL query for books published in 2001 and their authors.

Figure 7.4 shows a SPARQL query for the names and authors of books published in 2001. The result of this query is a table with two columns for book and author name and rows for the fitting values. The SPARQL query begins with setting the correct namespace for the RDF file. In the SELECT clause, the attributes to be selected are specified. When instead of the SELECT keyword, the CONSTRUCT keyword is used, the result is returned as an RDF graph, thereby allowing nesting subqueries. The WHERE clause gives the conditions that the result should conform to. In the example, the first line of the where clause defines as interesting triples those that link a subject *book* with an object *author* by the predicate *writtenBy*. The second line further specifies the condition by saying that the desired subject *book* should also have a predicate *publishedIn* with the object *2001*. Querying for a predicate is not shown in this example but also possible in SPARQL.

```
PREFIX lit: <http://ex.org/literature>
SELECT ?book,?year
WHERE {?book lit:writtenBy ?author.
    OPTIONAL {?book lit:publishedIn ?year}.
    FILTER regex(?author,"^auth.*")}
```

Result	
book	**year**
book1	
book2	2001

Figure 7.5: SPARQL query for authors with optional publishing year.

Figure 7.5 shows two other functionalities possible with SPARQL. The first line of the WHERE clause selects all book-writtenBy-author triples from the RDF graph. The second line defines the optional retrieval of the publishing year of the book. If this optional query is not successful, the respective cell of the resulting table is simply left empty. In the third line of the WHERE clause, the result is additionally filtered according to a regular expression such that the name of the author has to start with 'auth'. Since that is true for all two books in our RDF graph, the result contains both books. The publishing year is only given for *book2*, because there is none defined for *book1*.

7.2.4 SeRQL

Another interesting development is SeRQL [7] for the Sesame application which will be presented later in Section 7.3.2. Like SPARQL it uses a syntax very similar to SQL. While not officially endorsed by the W3C like SPARQL is, it nonetheless comes with some very interesting features currently missing in SPARQL. One of those is the language's awareness of the underlying schema of an RDF document, as defined for example in an RDF schema. While inferencing of new information based on an ontology (see Chapter 4) is supported by many semantic storage solutions, it is an actual requirement for solutions wanting to implement SeRQL. SeRQL also defines specific schema-aware constructions, for example *direct-SubClassOf*, which is only true for classes that are a direct subclass of a given class.

Figure 7.6 shows an example of a SeRQL query for all books published in 2001 and their authors. Here the keyword SELECT is used, so the query would return the same result as the SPARQL query in Figure 7.4. If instead CONSTRUCT was used, the query would return an RDF graph containing only the nodes matching the query. In the first line of the query, the attributes to be selected are defined. Unlike SPARQL, the SeRQL query then describes the set of nodes to be selected in the FROM clause. All names in curly brackets refer to nodes in the graph, the others to predicates. In the WHERE clause the result is then restricted to match only books written in 2001. The USING NAMESPACE clause is similar to the PREFIX in SPARQL. Optional path expressions and of course subclauses are also defined in the SeRQL language.

```
SELECT DISTINCT book, author
FROM ({book} lit:writtenBy {author},
     {book} lit:publishedIn {year})
WHERE year=2001
USING NAMESPACE lit=<http://ex.org/literature>
```

Figure 7.6: SeRQL query for books published in 2001 and their authors.

7.3 Overview of RDF Storage Solutions

This section will try to give a quick overview of some available RDF storages. The selection is restricted to applications that are still being actively developed. This list is by no means conclusive. Readers interested in using one of these applications are encouraged to further investigate details of the different solutions to find the one best fitting their needs.

7.3.1 Jena

Jena [43] is one of the most well known semantic web frameworks. Using Jena, data can be written to and queried from RDF graphs. The data can come from files, databases, URLs or a combination of them. Data queries to Jena are made with the SPARQL query language. Jena provides internal reasoners but also has an interface for plugging in external and more powerful reasoners, e.g. OWL reasoners. Please see Chapter 6 for an introduction to semantic reasoners. For storing RDF graphs, Jena uses a triple store and different database management systems are supported e.g. MySQL and PostgreSQL. Jena is released under a Hewlett-Packard specific open-source license.

7.3.2 Sesame

Sesame [40] is an RDF framework with support for RDF Schema inferencing and querying. Like in Jena, the data can come from different data sources, for example files or relational databases. Sesame supports SPARQL as an RDF query language but also introduces SeRQL as a powerful alternative. Sesame includes an RDF Schema reasoner and adds inferenced data to the data store. Sesame contains an abstraction layer called SAIL, which provides all database dependent methods to the upper layers through an API. This means that different implementations of the actual data storing and retrieving can easily be plugged in. Most commonly the data is stored in a triple store. Sesame is released under a BSD-style license.

7.3.3 Mulgara

Mulgara [31] does not call itself an RDF store but a metadata store using similar technologies as RDF. RDF data can however be imported into Mulgara. Mulgara has its own query language called iTQL but also supports SPARQL. Mulgara does not use a relational database system for storing the data, it does however employ a similar technique as triple stores. Inferencing from RDF Schema is supported. Mugabe is released under the Open Software License.

7.3.4 Oracle Spatial 11g

Oracle Spatial 11g [35] is an optional component of the Oracle 11g Enterprise Edition. It supports RDF, RDF-S and OWL, including a native inference engine. Oracle 11g does not natively support SPARQL, however full SPARQL support can be added with a Jena plug-in. Oracle reports to have loaded 1.1 billion triples, inference performed on this triple store resulted in additional 500000 inferred triples. The three open source solutions presented before have each been reportedly filled with maximally a few hundred thousand triples [36]. However no independent benchmarks of the different solutions have been performed.

7.4 Summary

This chapter dealt with the storage and querying of semantic data. Three different approaches for efficiently storing semantic data were introduced: the triple store approach, modeling to relational database systems and vertical partitioning. A comparison of these three approaches showed, that their performance is largely dependent on the type of queries that are asked, i.e. requesting explicit or implicit information; a result which should be taken into consideration when deciding for a semantic data storage.

This comparison was followed by a presentation of different semantic query languages. Of those the W3C recommended language SPARQL was identified as the de-facto standard for querying of RDF data. The chapter finished with a short introduction to different semantic storage software packages: Jena, Sesame, Mulgara and Oracle.

8 Challenges

In this chapter we will discuss the challenges that can arise when using the technologies that were presented in the previous chapters. We will also try to give directions how to best handle these and to point out alternatives where possible.

8.1 Structuring Information

In order to use data efficiently it is beneficial to have a coherent data structure. The challenge here is that the information that is being produced is often ambiguous, subjective, contradicting and/or incomplete. Adding to this, are the difficulties of creating sound and complete ontologies (further discussed in Section 8.3). It can easily be seen that structuring the available information in a satisfying way can be a very difficult task.

Choosing the correct methods and technologies for structuring information is often the first hurdle. It is important to keep in mind how and for what purpose the information is going to be used later on.

Some aspects to take into consideration when structuring information are:

- End purpose (how structured does the information need to be?).
- Scalability requirements (how much information is to be handled and how responsive to queries the system must have to be).
- Quality requirements (how many types of information are to be handled and how complex queries and result sets do we want to be able to handle?).
- Model extensibility (how easy is it to modify the data model?).
- The information structuring process (e.g. how is input structuring supported?).

In Chapter 3 and 7 we discussed different standards for structuring information (e.g. XML, RDF, OWL) and their use.

8.2 Data Overhead

XML and other XML based extensible languages offer a clearly defined data exchange format. Such a format is crucial for the upper layers in the information system stack, enabling communication and data exchange between heterogeneous components. These advantages of XML unfortunately come with drawbacks. XML adds overhead data to the data itself. If the XML document is not properly designed, it can become seriously inflated compared to

the original data. This can become a problem not only for the actual exchange of data, for example over a network, but applications also need to serialize the data objects they want to exchange into XML documents, which the receiving application then needs to parse again into a data object. Therefore to avoid performance issues, special care should be taken when defining the schema of XML documents.

8.3 Ontology Building and Using

It may be easier for a user to find information in an information system if the data in the system is tagged with terms defined in an ontology. Several things must be solved in order to benefit from such a system. First, a suitable ontology that meets the information usage requirements must be developed. Second, the information within the knowledge base must be tagged with terms defined in the ontology. Third, the ontology must be up to date with respect to the users' intentions and requirements.

Ontology construction is a research area of its own and several methodologies exist. It is recommended that you follow a well established methodology since the chances of missing some important aspect decreases. An important step is to involve domain experts and end-users during the construction phase. Otherwise, there is a chance that the ontology is useless from the end user's point of view.

Manual semantic annotation is a time consuming activity. However, Named Entity Recognition (NER) algorithms can be used to automatically find things like people and places in a text. Such algorithms might be used to automate the process of semantic annotation or, at least, be deployed in order to give the user well-founded suggestions.

Another possible problem which needs to be addressed is change management. Using ontologies enables the inference, or discovery, of new facts. However, by changing the ontology you change the axioms and rules that new facts are based upon. Such a change may have a major impact on the whole knowledge base.

8.4 Storage

Information storage brings forth many challenges, especially when storing semantic information. Scalability in terms of parallel query evaluation is the main one, this has been addressed in section 7.1. Existing storage solutions for RDF data generally perform worse than conventional relational database systems, because of limitations inherent to semantic query evaluation. Performance differences often lie in the order of magnitudes. This is the drawback of storing generically structured triples instead of data structures specific to the data of one application.

Research in this area is ongoing[1] and hopefully there will be a wide variety of efficient solutions for semantic storing and query processing soon.

However, for now when developing an application storing semantic data, it should be considered, whether the interoperability and openness of semantic data stores outweigh their performance issues or if a conventional relational database design would be a better choice.

8.5 Reasoning

In the semantic web stack, RDF-S and OWL provide the means to define ontologies and building knowledge bases in which new knowledge can be inferred from existing information. Inference engines have been presented in this document in Chapter 6. As mentioned there, RDF-S and OWL are subsets of first order logic systems. Full first-order logic systems have been proven to be computationally intractable, meaning that the time required for the worst case grows exponentially with the size of the knowledge base. Using only a subset of the logic system somewhat reduces the time complexity but also the expressiveness. However reasoning with OWL ontologies still have a very high complexity.

In [22] it is pointed out that there are two opposing demands on ontologies:

- A high fidelity description with low or no inference support without decidability guarantees.
- A low fidelity description with strong inference support with decidability guarantees.

It is therefore important to select an ontology language and a reasoner that handle the actual demands of an application in regards to expressiveness and time complexity. In [6] an evaluation of different reasoning algorithms is performed, trying to give support for deciding on a specific reasoner for a given application. In the paper the reasoners KAON2, OWLIM, Pellet and RacerPro were benchmarked on different A-Box sizes, using RDFS, OWL-Lite, OWL-DLB and ODL-DL as ontology languages. In the evaluation OWLIM performed very well for ontologies with little expressiveness, regardless of the A-Box size. For ontologies with high expressiveness, RACER can be recommended if the A-Box is rather small, and KAON2 for bigger A-Boxes.

8.6 Final Thoughts

In this chapter we discussed that semantic technologies are not purely beneficial, they come with their own set of challenges. It is important that developers are aware of these and that they consciously decide whether their applications or parts of their applications need semantic support. For example, for high

[1] SDR[15] supports parallel execution of queries, which makes it one such scalable effort.

performance applications it is generally advisable to use established systems like relational databases instead.

It also needs to be stressed that the semantic technologies are still in development and some are not yet as mature as established ones. Also, no critical mass of users has adopted them yet. It is to be expected that when more people start to use semantic technologies some of the challenges presented here will be resolved.

9 Research Project

In this report we have tried to give the reader an introduction to the promises and problems with semantic information management. Naturally, all aspects could not be covered here in this report. In this chapter we give information about the project that has sponsored this report, i.e. "Efficient Information Management of M&S Resources in Command and Control Systems" (InfoM&S) and conclude with the research goals for the coming years.

9.1 InfoM&S Project Description and Research Goals

The "Efficient Information Management of M&S Resources in Command and Control Systems" project is a three year project within the FOI Modeling and Simulation Research and Development program. The project started in 2009 as a spin-off project from the "Semantic Distributed Repository" project [15].

When handling different types of documents (M&S models, reports, protocols, simulations, etc.), it is important to be able to quickly find the correct information[1]. "Correct information" means several things such as, being able to

- Identify entities in the form of people, objects and places.
- Identify the context.
- Find links to other information.

For us humans, this is often an easy thing to do, but with increasing number of documents, it soon becomes untenable and computers are needed[2].

Computers are very capable in managing structures (syntax) but are not as adept in understanding and managing the content (semantics)[3]. Thus, a necessary component in computerized information management work is often semantic technologies and methods.

[1] Preferably as quickly and with as high precision and recall as possible, (see Chapter 2).

[2] Together with:

- Structured languages and methods with which to express and structure the information (see Chapters 3 and 4).
- Formalisms to express the knowledge with and draw logical conclusions from it (see Chapters 5 and 6).
- Ways to query and store the structured information (see Chapter 7).

[3] There are many challenges (see Chapter 8).

Here are some examples of the challenges that the project will look at in the coming years:

Entity extraction: Is necessary when you want to structure information in order to be able to automatically identify entities such as people and places.

Context relating of entities and relationships: Is necessary since in non-complete data, it is sometimes difficult to determine which person or place is being referred to.

Semantic relatedness: Often when information is received from different parties, different terms have been used for the same things, or vice versa. Being able to determine the semantic relatedness (are they synonyms, antonyms, or related at all?) between terms and expressions is crucial for the task of interconnecting information i.e. determining which entities are related and which are not.

Semantic cleaning: When there are large amounts of data coming in from different sources it needs to be stored in a consistent way. This storing results in a merging of the data. Redundant, missing, ambiguous and contradictory data, requires normalization, consolidation and cleansing of the data. It is not always obvious how to do this.

Automatic generation of ontologies: Creating an ontology is often a very time consuming, but unavoidable, process. When ontologies are finished and introduced into an information management system they are also usually quite static in nature. Automating all or parts of the process could mean big profits in both time and money.

The project will mainly focus on supporting the construction and management of ontologies for both the FMKE EBAONet project and the EU SM4All project. The goal is to compare, implement and evaluate some of the available methods for the information management issues that were previously described. In cases where there aren't any methods and technologies already available, the project will develop its own. The project intends to keep an open dialog and actively cooperate with their users and projects within the Swedish Armed Forces and the EU, to disseminate the obtained results.

The listed challenges are those that recur in many areas and can be applied in most domains that handle data in large quantities. For example, the sensor, modeling and simulation, command and control, intelligence and business domains.

Bibliography

[1] D. J. Abadi, A. Marcus, S.l R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[2] Ontotext AD. The kim platform: Semantic annotation. Last viewed 2009-11-02. `http://www.ontotext.com/kim/semanticannotation.html`.

[3] V. Alexiev, M. Breu, J. Debruij, D. Fensel, Lausen R., R. Lara, and H. Lausen. *Information Integration with Ontologies*. John Wiley and Sons Ltd, March 2006.

[4] F. Baader. Cel. Last viewed 2009-11-02. `http://lat.inf.tu-dresden.de/systems/cel/`.

[5] S. Bechhofer. Hoolet. Last viewed 2009-11-02. `http://owl.man.ac.uk/hoolet/`.

[6] J. Bock, P. Haase, Q. Ji, and R. Volz. Benchmarking owl reasoners. June 2008.

[7] J. Broekstra. SeRQL: Sesame RDF query language. In M. Ehrig, editor, *SWAP Deliverable 3.2 Method Design*, pages 55–68. 2003.

[8] LCC Clark & Parsia. Pellet: The open source owl reasoner. Last viewed 2009-11-02. `http://clarkparsia.com/pellet`.

[9] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[10] A. De Nicola, M. Missikoff, and R. Navigli. A software engineering approach to ontology building. *Inf. Syst.*, 34(2):258–275, 2009.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] S. DeRose and J. Clark. XML path language (XPath) version 1.0. W3C recommendation, W3C, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[13] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems (5th Edition)*. Addison Wesley, March 2006.

[14] M. Fernandex, A Asimction Gomez-Perez, and Juristo N. Methontology: from ontological art towards ontological engineering. In *Proceedings of the AAAI97 Spring Synompsium Series on Ontological Engineering*, 1995.

[15] M. García Lozano, F. Moradi, and E. Tjörnhammar. Slutrapport för sdr - semantikbaserat distribuerat resursbibliotek. Användarrapport/User Report FOI-R–2608–SE, FOI, Stockholm, 2008.

[16] B. Grosof. Racerpro. Last viewed 2009-11-02. `http://sweetrules.projects.semwebcentral.org/`.

[17] M. Grunninger and M. Fox. Methodology for the design and evaluation of ontologies. In *Proceedings of the Int. Conf. AI 1995, Workshop on Basic Ontological Issues in Knowledge Sharing*, 1995.

[18] N. Guarino. Some ontological principles for designing upper-level lexical resources. In *the First International Conference on Language Resources and Evaluation*, pages 527–537, 1998.

[19] I. Horrocks, P. F. Patel-Schneider, and F. Van Harmelen. From shiq and rdf to owl: The making of a web ontology language. *Journal of Web Semantics*, 1:2003, 2003.

[20] Centrum Wiskunde & Informatica. Monetdb. Last viewed 2009-11-07. `http://monetdb.cwi.nl/`.

[21] Jess. Java expert system. Last viewed 2009-11-09. `http://jessrules.com`.

[22] C.M. Keet and M. Rodriguez. Comprehensiveness versus scalability: guidelines for choosing an appropriate knowledge representation language for bio-ontologies, 2007.

[23] M. Keshk and S. Chamblessn. Model driven ontology:a new methodology for ontology development. In *OIC 08 Proceedings*, 2008.

[24] Racer Systems GmbH & Co. KG. Racerpro. Last viewed 2009-11-02. `http://www.racer-systems.com/`.

[25] R. Lämmel. Google's MapReduce Programming Model – Revisited. Accepted for publication in the Science of Computer Programming Journal; Online since 2 January, 2006; 42 pages, 2006–2007.

[26] Berners T. Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[27] D. McGuinness. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*. MIT Press, March 2006.

[28] M. Minsky. A framework for representing knowledge. *MIT-AI Laboratory Memo 306*, June 1974. `http://web.media.mit.edu/~minsky/papers/Frames/frames.html`.

[29] B. Motik. Hermit reasoner. Last viewed 2009-11-02. http://hermit-reasoner.com.

[30] B. Motik. Kaon2. Last viewed 2009-11-02, Semtemper 2009. http://kaon2.semanticweb.org.

[31] Mulgara. Mulgara. Last viewed 2009-11-07. http://mulgara.org.

[32] N. F. Noy and D. L. McGuinness. Ontology development 101: A guide to creating your first ontology. Technical report, 2001.

[33] Ontotext. Ontotext semantic technology lab. Last viewed 2009-11-02. http://www.ontotext.com/kim/semanticannotation.html.

[34] Ontotext. Owlim semantic repository. Last viewed 2009-11-02. http://www.ontotext.com/owlim/index.html.

[35] Oracle. Oracle semantic technologies. Last viewed 2009-11-04. http://www.oracle.com/technology/tech/semantic_technologies/index.html.

[36] Oracle. Oracle semantic technologies presentation. Last viewed 2009-11-04. http://www.oracle.com/technology/tech/semantic_technologies/pdf/oracle\%20db\%20semantics\%20tech\%20talk\%2020080722.pdf.

[37] S. Robertson. Understanding inverse document frequency: On theoretical arguments for idf. *Journal of Documentation*, 60:2004, 2004.

[38] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, and C. Pinkel. An experimental comparison of rdf data management approaches in a sparql benchmark scenario. pages 82–97. 2008.

[39] A. Seaborne and E. Prud'hommeaux. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[40] Aduna software. Sesame. Last viewed 2009-11-07. http://www.openrdf.org/.

[41] U. Straccia. Fuzzy dl. Last viewed 2009-11-02, Semtemper. http://gaia.isti.cnr.it/~straccia/software/fuzzyDL/fuzzyDL.html.

[42] R. Studer, V. Benjamins, and D. Fensel. Knowledge engineering: Principles and methods. *Data and Knowledge Engineering*, 25, 1998.

[43] Jena Team. Jena - a semantic web framework for java. Last viewed 2009-11-02. http://jena.sourceforge.net/.

[44] D. Tsarkov. Fact++. Last viewed 2009-11-02. http://owl.man.ac.uk/factplusplus/.

[45] M. Uschold. Ontologies: Principles,methods and application. *Knowl. Eng. Rev.*, 11(2), 1996.

[46] M. Uschold. Knowledge level modelling: concepts and terminology. *Knowl. Eng. Rev.*, 13(1):5–29, 1998.

[47] W3. SWRL. Last viewed 2009-11-02, 2009. http://www.w3.org/Submission/SWRL/.

[48] W3C. OWL 2 overview. Last viewed 2009-11-02. http://www.w3.org/TR/owl2-overview/.

[49] W3C. OWL 2 profiles. Last viewed 2009-11-02. http://www.w3.org/TR/owl2-profiles/.

[50] W3C. OWL reference. Last viewed 2009-11-02. http://www.w3.org/TR/owl-ref/.

[51] W3C. World Wide Web Consortium. Last viewed 2009-11-02. http://www.w3.org.

[52] Webucator. XSLT Tutorial. Last viewed 2009-11-02, 2009. http://www.learn-xslt-tutorial.com/.

[53] Wikipedia. Index - ir. Last viewed 2009-11-02. http://en.wikipedia.org/wiki/Index_(information_technology).

[54] Wikipedia. Information retrieval. Last viewed 2009-11-02. http://en.wikipedia.org/wiki/Information_retrieval.

[55] Wikipedia. Inverted Index. Last viewed 2009-11-02. http://en.wikipedia.org/wiki/Inverted_index.

[56] Wikipedia. Precision and Recall. Last viewed 2009-11-02. http://en.wikipedia.org/wiki/Precision_and_recall.

[57] Wikipedia. Semantic Network Figure. Last viewed 2009-11-02. http://en.wikipedia.org/wiki/File:Semantic_Net.svg.

[58] Wikipedia. Standard Boolean Model. Last viewed 2009-11-02. http://en.wikipedia.org/wiki/Standard_Boolean_model.

[59] Wikipedia. Term Frequency. Last viewed 2009-11-02. http://en.wikipedia.org/wiki/Term_frequency.

[60] Wikipedia. Vector Model. Last viewed 2009-11-02. http://en.wikipedia.org/wiki/Vector_space_model.

[61] Wikipedia. Description Logics. Last viewed 2009-11-02, 2009. http://en.wikipedia.org/wiki/Description_logics.

[62] Wikipedia. The Extensible Stylesheet Language Family Transformations. Last viewed 2009-11-02, 2009. http://www.w3.org/TR/xslt.

[63] Wikipedia. Horn Clause. Last viewed 2009-11-02, 2009. http://en.wikipedia.org/wiki/Horn_clause.

[64] Wikipedia. Ontology. Last viewed 2009-11-02, 2009. http://en.wikipedia.org/wiki/Ontology_(computer_science).

[65] Wikipedia. RDF. Last viewed 2009-11-02, 2009. http://en.wikipedia.org/wiki/Resource_Description_Framework.

[66] Wikipedia. Semantic Web Stack. Last viewed 2009-11-02, 2009. http://en.wikipedia.org/wiki/Semantic_Web_Stack.

[67] Wikipedia. SWRL. Last viewed 2009-11-02, 2009. http://en.wikipedia.org/wiki/Semantic_Web_Rule_Language.

[68] Wikipedia. XML. Last viewed 2009-11-02, 2009. http://en.wikipedia.org/wiki/XML.

[69] Wikipedia. XML Namespace. Last viewed 2009-11-02, 2009. http://en.wikipedia.org/wiki/XML_namespace.

[70] Wikipedia. XML Path Language. Last viewed 2009-11-02, 2009. http://www.w3.org/TR/xpath.

[71] Wikipedia. XML Schema. Last viewed 2009-11-02, 2009. http://en.wikipedia.org/wiki/XML_schema.

[72] Wordnet website at princeton university. Last viewed 2009-11-02. http://wordnet.princeton.edu/wordnet/.