Abstract

The main idea of this survey paper is to explore current state of the Semantic Web with the major focus on search. The paper includes introductory sections on the SW and its stack, a review of several papers on semantics and ontology-based search, sections on query languages and knowledge base systems that enable Semantic Web search. The paper complements existing surveys with new entries (systems), more detailed and recent specifications for some systems.

1. Introduction

The main idea of this survey paper is to explore current state of the Semantic Web (SW) with the major focus on search. The paper includes introductory sections on the SW and its stack, a review of several papers on semantics and ontology-based search, sections on query languages and knowledge base systems that enable SW search. Our survey is by no means complete. There have been a lot of work published on the above topics, we could include only a small portion of them.

There exist several other surveys on query languages (e.g. [40] and [44]) and knowledge base systems (e.g., [14], [16] and [44]). We do not follow their style to describe as many languages/tools as possible, but rather tend to give more detailed overview of used architectures, ideas, etc. For example, our section on knowledge base systems complements those surveys with new entries such as Semagix Freedom, TAP, OWLJessKB, and DLDB, as well as with more recent specifications for Sesame, Jena and KAON.

Organization. Sections 2 and 3 introduce the Semantic Web and its stack. Section 4 describes three types of semantics for the Semantic Web. Types of search and ontology-based information retrieval model are explained in Sections 5 and 6 respectively. Section 7 provides a brief overview of query languages for the Semantic Web and a primer on RQL. Section 8 presents a survey on seven knowledge base systems for the Semantic Web. Finally, Section 9 concludes the paper.

2. What is the Semantic Web

The most cited definition of the SW is given in the following:

“The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.” (Tim Berners-Lee et al.) [20]

The main idea of the SW, proposed by Tim Berners-Lee, is to enhance existing data on the Web with machine-interpretable metadata to enable better automation, integration, discovery and reuse across various applications.

“The Semantic Web is a web of data, in some ways like a global database.” and “Leaving aside the artificial intelligence problem of training machines to behave like people, the Semantic Web approach instead develops languages for expressing information in a machine processable form.” (Tim Berners-Lee) [17]

There is some confusion about the term “Semantic Web”:

“The fact that the programmer and the interpreter of the computer output use the symbols to stand for objects in the world is totally beyond the scope of the computer. The computer, to repeat, has a syntax but no semantics.” (John Searle) [48]

“Developing XML as a richer version of HTML was generally a good idea. But what botched the Semantic Web is that promoting a universal syntax does nothing to pro-
mote semantics. To avoid further confusion, it would be a good idea to rename it the syntactic web.” (John Sowa) [30]

We do not go into philosophical discussion about correctness of term “Semantic Web”, but rather suggest that the usefulness of the SW idea does not depend on the title we use for it. Tim Berbers-Lee has noted that the semantic in Semantic Web means machine processable [29].

3. The Semantic Web Stack

The SW stack, which describes its components and their relationships, is shown in Figure 1 (copied from [30]).

3.1. Layer 1 - URIs and Unicode

Each object on the SW is identified by a unique URI (Uniform Resource Identifier) [19] assigned to it.

There are different subclasses of URIs, such as Universal Resource Locators (URLs) and Uniform Resource Name (URNs).

It is important to realize, that the SW will include not only resources like Web pages, images, audio or video files, but also objects like people, events or places. Resources on the Web have unique URLs, however there is no standard way to assign URIs to people or events.

Unicode is a character set that can deal with multiple human languages.

3.2. Layer 2 - XML and Namespaces

The SW metadata uses XML syntax.

Extensible Markup Language (XML) [24] is a standard text format for serializing data using tags. XML has been around for about a decade and has many technologies and tools available for XML data processing, such as DOM and SAX parsers, DTD and XML Schema validation, XPath and XQuery query languages, XML databases, etc.

XML Namespaces [23] are extensions to XML, which provide the mechanism to uniquely identify the element in the vocabulary, where the vocabulary consists of XML element types and attribute names. In multiple XML documents, vocabularies can overlap, leading to problems of recognition and collision. In simple words, the namespace mechanism defines a URI to indicate the vocabulary, and an element name to indicate the element in the vocabulary.

3.3. Layer 3 - RDF

Resource Description Framework (RDF) [15] is a general-purpose language for representing information in the Web. RDF model is the RDF graph, whose nodes are represented by RDF URI references, blank nodes or plain literals, and arcs are labeled with RDF URI references. (Note that a RDF URI is a URI with restrictions on allowed characters.)

An example of an RDF graph is shown in Figure 2 (copied from [30]). The RDF graph consists of triples, where each triple consists of a subject, a predicate and an object. Each triple translates into a statement about a resource. For example, the triple shown in Figure 3 (copied from [30]) has the following interpretation: the creator-predicate identified by URI http://purl.org/dc/elements/1.1/creator of the resource (object) identified by URI http://www.example.org/index.html is the object identified by URI http://www.example.org/staffid/85740.

In the following we give a brief description of each SW stack layer.

### Figure 1. The Semantic Web Stack

| Trust | Proof |
| Logic Framework | Rule |
| Ontology | RDF Schema |
| RDF M&S | XML Namespaces |
| URI | Unicode |

### Figure 2. An RDF Graph Example
3.4. Layer 4 - RDF Schema

RDF Schema (RDFS) [25] is a language to describe RDF vocabularies (ontologies).

RDFS allows describe class and property hierarchies; give labels to URIs; constrain domain and range of properties; etc.

Although RDF Schema and XML Schema [34] are both “schemas”, they have different purposes: RDFS is for inference and XMLS is for validation.

3.5. What is Ontology

One of the frequently cited definitions of an ontology is given by Gruber [36]: an ontology is a “formal specification of a conceptualization”, and is shared within a specific domain. In other words, an ontology is a document, defined in a formal language (like RDF Schema), that describes a vocabulary of terms or concepts (and their relationships) used for a specific domain.

3.6. Layer 5 - Ontology Languages

Two ontology languages, DAML+OIL [32] and OWL [13], are more complex than RDFS and provide more capabilities to define ontologies.

Web Ontology Language (OWL) is the successor of DAML+OIL, therefore we focus our discussion on the former only. OWL extends RDFS with the following features: property characteristics (TransitiveProperty, SymmetricProperty, FunctionalProperty, inverseOf, InverseFunctionalProperty), property restrictions (allValuesFrom, someValuesFrom, cardinality, hasValue), ontology mapping (equivalence between classes and properties – equivalentClass, equivalentProperty; identity between individuals – sameAs; different individuals – differentFrom, AllDifferent), complex class definitions (set operators – intersectionOf, unionOf, complementOf; enumerated classes – oneOf; disjoint classes – disjointWith), etc.

The OWL language provides three increasingly expressive sublanguages: OWL Lite (primarily, a classification hierarchy and simple constraint features), OWL DL (maximum expressiveness with guarantee of reasoning completeness and decidability) and OWL Full (maximum expressiveness and the syntactic freedom of RDF with no computational guarantees).

3.7. Layer 6 - Rules

Rules are statements that can be used to infer (discover) expressions (knowledge). There are no standards how to create rules for the SW.

The idea of rule statements is not new. For example, in logical programming languages (in particular, Prolog) the rule statement “male(X) :- son(X,Y)” might mean that an object (atom) is a male if this object is a son of some other object. Prolog uses backward chaining (top-down resolution) for reasoning.

Also, the similar idea is used in “deductive databases”, where reasoning rules are stated in datalog languages with Prolog-like syntax.

The example of a data processor for the SW is the Closed World Machine (cwm) [1]. It is a forward chaining reasoner which can be used for querying, checking, transforming and filtering information. Its core language is RDF, extended to include rules, and it uses RDF/XML or RDF/N3 serializations [1]. The above rule in Prolog can be rewritten using RDF/N3 (N3, Notation 3, is a language to express RDF in a different notation/syntax):

\[
\{ \text{x a :Male} \} \implies \{ \text{x :son y} \}
\]

The successful experience of using ontologies and rules on the Web is the Simple HTML Ontology Extensions (SHOE) [10], which uses special HTML tags for ontology definition as well as for creating inference rules.

Current version of OWL does not contain special structures to define rules, however some simple “rules” can be designed. For example, Prolog rule “male(X) :- man(X)” is similar to OWL definition

\[
<\text{owl:Class rdf:ID="Male"}>
<\text{owl:equivalentClass rdf:resource="Man"}/>
</\text{owl:Class}>
\]

which means that classes “Male” and “Man” are equivalent terms or any instance (object, atom) of “Male” is also an instance of “Man” and vice versa. However, for the above Prolog rule, the reasoner, knowing that an object is a male, can not infer that the very object is a man (unless we also state “man(X) :- male(X)”).

3.8. The Other Layers

To finish the SW stack discussion, we briefly describe the purpose of the rest layers (Logic Framework, Proof, Trust)
and components (Signature, XML encryption).

Logic Framework specifies a formalism for SW reasoning (e.g. Description Logic).

Signature, XML encryption (as well as Proof and Trust) are related to data security issues of the SW.

“As an open and distributed system, the Semantic Web bears the spirit that "anybody can say anything on anybody". People all over the world might assert some statements which can possibly conflict. Hence, one needs to make sure that the original source does make a particular statement (proof) and that source is trustworthy (trust).” (Shiyong Lu et al.) [43]

4. Semantics for the Semantic Web

Sheth et al. [50] described three types of semantics for the SW: the implicit, the formal and the powerful.

The implicit semantics is not stated explicitly and extracted from the patterns in data. For example, keyword occurrences, hypertext links, position in concept hierarchy, etc. This kind of semantics allows finding the relevance of data (document) to some semantic context, however it is not machine-processable – it is not possible to name a relationship between concepts.

The formal semantics is presented in some well-formed syntactic language. The formal language should include the following features: (1) the notions of model and model theoretic semantics – language expressions are interpreted in models which reflect “structure of the world”, and (2) the principle of compositionality – expression meaning is a function of the meanings of expression’s parts and of the way they are syntactically combined. Examples of such languages are RDF, OWL, Description Logics. This type of semantics is machine-processable. The major drawback of the formal semantics is that it becomes impractical as knowledge base size increases or knowledge is added from different sources.

The powerful (soft) semantics can exploit implicit and formal semantics (probably “incomplete”) to derive relationships using statistical analysis (e.g., probabilistic and fuzzy knowledge). The derived relationships are associated with likelihoods of being valid. The major drawback of the powerful semantics is prior assignments of probabilities to deal with uncertainties.

In summary, the current Web mostly exploits the implicit semantics (search engines like Google), the major focus of the SW is on formal and powerful semantics.

5. Types of Search

Guha et al. [37] identify two kinds of searches:

- **Navigational Searches**: In this class of searches, the user provides the search engine a phrase or combination of words which s/he expects to find in the documents. There is no straightforward, reasonable interpretation of these words as denoting a concept. In such cases, the user is using the search engine as a navigation tool to navigate to a particular intended document.
- **Research Searches**: In many other cases, the user provides the search engine with a phrase which is intended to denote an object about which the user is trying to gather/research information. There is no particular document which the user knows about that s/he is trying to get to. Rather, the user is trying to locate a number of documents which together will give him/her the information s/he is trying to find.

Example: A search query like “W3C track 2pm Panel” does not denote any concept. The user is likely just trying to find the page containing all these words. On the other hand, search queries like “Eric Miller” or “Dublin Ohio”, denote a person or a place. The user is likely doing a research search on the person or place denoted by the query (copied from [37]).

Both types of searches can be enhanced by exploiting relevant domain ontology and annotations.

6. Ontology-based Information Retrieval Model

In this section, we explore changes that the SW brings to IR model. The view of the classical IR model is shown in Figure 4 (copied from [35]). In the IR model, a query, formulated from an information need, is matched over document representations (e.g. index structures).

![Figure 4. A Classical IR Model](image_url)

Figure 5 (copied from [35]) illustrates the general ontology-based IR model. Information need and queries are refined and formulated based on an ontology structure. Resources (documents) are represented by annotations; ontologies are intended for inference. A matching process is replaced by exploration process, which can further use an ontology for navigation and logical reasoning. Finally, resource annotations are searched for relevant resources.
7. Query Languages for the Semantic Web

In this section, we describe a few RDF(S) query languages as there is extensive research in this area, as well as practical applications in RDF(S) storage and querying systems.

7.1. The Need for RDF Query Languages

Our discussion of the need for RDF query languages is based on [28]. We consider RDF(S) querying on three levels of abstraction:

1. Syntactic level (XML documents).
2. Structure level (a set of triples).

At the syntactic level, it seems possible to query RDF using XML query languages such as XPath [31] and XQuery [21]. However, the RDF data model is a graph, not a tree, and moreover, both its edges (properties) and its nodes (subjects/objects) are labeled. Relationships in the RDF data model that are not apparent from the XML tree structure become very hard to query. Finally, there are many different ways to serialize the same content using RDF syntax [28].

At the structure level, RDF documents are represented as a set of triples (Subject, Predicate, Object). The advantage of such a query is that it directly addresses the RDF data model, and that it is therefore independent of the specific syntax that has been chosen to represent the data. However, a disadvantage of any query language at this level is that it interprets any RDF model only as a set of triples, including those elements which have been given a special semantics in RDFS [28]. For example, information of class and property hierarchies described in RDFS is not exploited in such query languages (e.g., SquishQL [6] and RDQL [7]).

Finally, at the semantic level, semantics of RDF descriptions is also used for querying. For example, a query which retrieves all instances of some class, will also retrieve instances of that class subclasses. An example of an RDF query language at this level is RQL [41].

7.2. Overview of RDF Query Languages

There is no a standard language to query ontologies and annotations. Recently, there have been proposed several RDF query languages, such as RQL [41], SeRQL [26], Versa [46], N3 [18], TRIPLE [12], RDQL [7], SquishQL [6], etc. Due to space constraints of this paper, we only give a very brief description of these languages, however provide better introduction to RQL.

RQL is a OQL-like typed functional language, which defines a set of basic functions (queries) and iterators and relies on functional composition to build more complex queries. One of the RQL distinguishing features is internal support of schema (RDFS) queries and smooth combination of schema and data querying. Internally, RQL is based on interpretation of the RDF graph.

SeRQL is an attempt to design more powerful and easy-to-use querying and transformation language based on existing ideas of RQL, RDQL, N3, etc. Syntactically, it is similar to RQL, however it is based on interpretation of the RDF Model Theory.

Versa takes an interesting approach in that the main building block of the language is a list of RDF resources. RDF triples play a role in the so-called traversal operations [40].

N3 provides an optional notation (text-based syntax) to express RDF triples, as well as capability to state rules and queries. N3 does not distinguish between rules and queries. N3 query language is based on the RDF data model.

TRIPLE is derived from F-Logic, such that RDF triples (Subject, Predicate, Object) are represented as F-Logic expressions Subject[Predicate− Object]. TRIPLE is similar to N3, as they both “rule-oriented”.

RDQL has a SQL-like syntax similar to RQL and SeRQL. RDQL does not provide support for schema (RDFS) queries. RDQL is based on RDF graph.

SquishQL is similar to RDQL in its SQL-like syntax, interpretation of data model and capabilities. Research on SquishQL is discontinued.

The comparison of six RDF query languages is presented in [40]. Authors explore expressiveness, closure, adequacy, orthogonality, and safety properties of chosen languages. Additionally, authors construct 28 queries to evaluate each language: count 0.5 of a point if language succeeds in for-
mulating and evaluating a single query. The resulting ranking (maximum score is 14) is as follows:

- RQL – 10.5
- SeRQL – 8.5
- Versa – 7.5
- N3 – 7.0
- TRIPLE – 5.5
- RDQL – 4.5

More research on standardization and improvement of query languages should be done, as even the leading language, RQL, was not suitable for some query types. Finally, authors [40] include grouping, aggregation, sorting, etc. to the “wish list” of RDF query language features.

In [44], authors similarly provide detailed theoretical evaluation of expressive power for seven RDF query languages including RQL, SquishQL, Versa, TRIPLE, etc.

7.3. RQL Primer

RQL [41, 5] is a OQL-like typed functional language, which defines a set of basic functions (queries) and iterators and relies on functional composition to build more complex queries.

We present the informal description of RQL basic features by examples over data given in Figure 6 (copied from [41]).

Basic RQL queries:
- Class and Property – return a bag with all classes and properties respectively.
- subClassOf – returns a bag with transitive subclasses of an argument class.
- subClassOf’ – returns a bag with direct subclasses of an argument class.
- subPropertyOf and subPropertyOf’ – return a bag with transitive and direct subproperties of an argument property respectively.
- domain and range – return domain and range of an argument property respectively.
- Name of a class (e.g. Artist or Painting) – returns a bag of URIs that represent resources of class type.
- Name of a property (e.g. creates) – returns a bag of ordered pairs of resources (URIs) that appear as a subject and an object of a property.
- typeof – returns a class (type) of an argument resource (URI).
- bag and seq – construct a bag and a sequence respectively.
- union, intersect, minus – set operators.

- =, <, > and like – Boolean predicates, that can be applied on literals or URIs. Additionally, class and property names can be compared respectively (e.g. Painter < Artist returns true since Painter is a subclass of Artist).
- min, max, avg, sum and count – aggregate functions.
- select – from – where – filters (select) RDF annotation and schema collections (from) based on conditions (where). Variables are introduced with this construct.

In the following, we provide several examples of RQL queries.

Basic queries
Example 1.
- Query: subClassOf(Artifact)
- Result: Bag(Painting, Sculpture)

Example 2.
- Query: creates
- Result: Bag(< &r5, &r6 >, < &r1, &r2 >, < &r1, &r3 >)

Example 3.
- Query: typeof(www.culture.net#picasso132)
- Result: Bag(Painter)

Schema queries
Example 5.
- Information need: Which classes can appear as domain of the property creates?
- Query: select $C from { $C } creates
- Result: Bag(Artist, Painter, Sculptor)

Example 6.
- Information need: Find all properties (and their range) that are applicable on class Painting.
- Query: select @P, range(@P) from { $C } @P where $C = Painting
- Result: Bag(< technique, string >, < exhibited, Museum >)

Data queries
Example 7.
Figure 6. An Example of RDF Resource Descriptions

- Information need: Find the Museum resources that have been modified after year 2000.

- Query:
  
  ```
  select X, Y  
  from Museum{X}.last_modified{Y}  
  where Y>=2000-01-01  
  ```

- Result: \( Bag(< &r4, 2000-06-09 >, < &r7, 2000-02-01 >) \)

Example 8.

- Information need: Find the names of Artists whose Artifacts are exhibited in “Rodin Museum”.

- Query:
  
  ```
  select F, L  
  from {X}creates.exhibited{Y}.title{Z}, {X}fname{F}, {X}lname{L}  
  where Z like “Rodin Museum”  
  ```

- Result: \( Bag(< Rodin, August >) \)

Combination of schema and data queries

Example 9.

- Information need: Find all things (and their types) created by “Sculptors”.

- Query:
  
  ```
  select Y, $Y from Sculptor{X}.creates{Y : $Y}  
  ```

- Result: \( Bag(< &r5, Sculpture >) \)

For more examples and more complex queries, please refer to [41, 5].

8. Knowledge Base Systems for the Semantic Web

There exist many RDF(S), DAML+OIL and OWL knowledge base systems, that can be used as a foundation for SW repositories. We provide brief introduction into several such systems, describing their general functionality and architecture, further details on storage design, querying and reasoning support.
8.1. Semagix Freedom

Semagix Freedom [8, 49] is the commercial system which provides support on all levels of SW application development. In particular, Semagix Freedom functionality includes:

- automatic classification of content,
- ontology-driven metadata extraction,
- support for complex query processing involving metadata and ontology,
- ontology design,
- content aggregation,
- knowledge aggregation and creation,
- metadata extraction,
- content tagging and querying of content and knowledge,
- exporting of ontology in RDF/RDFS with some constraints that can not be expressed in RDF/RDFS.

The system’s architecture is presented in Figure 7 (copied from [8]). Semagix Freedom provides a modelling tool to design an ontology. Knowledge Agents (KA) automatically maintain the ontology. Metadata is stored in metabase and extracted by Content Agents (CA) from structured, semi-structured and unstructured sources of different formats. Both KA and CA are created without programming using special toolkit based on domain requirements. The extracted content is further “enhanced” by Semantic Enhancement Server: identification of relevant document features (e.g. currencies, dates), entity disambiguation, content annotation (tagging). The metabase is stored in a relational database and its snapshot resides in main memory to facilitate fast querying. Semantic Query Server provides querying mechanism through HTTP and Java APIs, returning results in XML with published DTDs.

A few facts about Semagix Freedom performance are listed bellow (copied from [49]):

- Typical size of an ontology schema for a domain or task ontology: 10s of (entity) classes, 10s of relationships, few hundred property types.
- Average size of ontology population (number of instances): over a million of named entities.
- Number of instances that can be extracted and stored in a day (before human curation, if needed): up to a million per server per day.
- Number of text documents that can be processed for automatic metadata extraction per server per day: hundreds of thousand to a million.
- Performance for search engine type keyword queries: well over 10 million queries per hour with approx. 10ms per query for 64 concurrent users.
- Query processing requirement observed in an analytical application: approx. 20 complex queries (involving both Ontology and Metabase) to display a page with analysis, taking a total of 1/3 second for computation (roughly equivalent to 50+ query over a relational database with response time over 50 seconds).

8.2. Semantic Search and TAP

Semantic Search [37] is an application of the Semantic Web to search based on TAP [11, 37]. TAP is intended to be an infrastructure for applications on the Semantic Web. TAP provides the following features:

- Simple query interface GetData, which should be supported by SW repositories to be used with Semantic Search.
- Scrapping (e.g. HTML scrapers). TAP provides an infrastructure for interpreting a GetData request, mapping it to an appropriate scraping task and executing the scraping so that a client can pretend that the site offers a GetData interface to its data.
- Publishing. On the server side, an Apache HTTP server module, TAPache, is used for exposing data via the GetData interface. Data is published in form of RDF annotations, TAPache compiles RDF files into memory-mappable graph structures to avoid RDF parsing when querying.
Registries and caching. Registries (separate servers) keep track of which URL has values for which properties about which classes of resources. A client can direct the query to the registry which then redirects the query to the appropriate sites. Registries also cache the responses to GetData requests for efficiency.

GetData is a simple querying interface which is built on top of SOAP (SOAP is a protocol for performing Remote Procedure Call). By means of GetData, a client program can access the values of one or more properties of a resource from an RDF graph. Each GetData query is a SOAP message addressed to that URL. The message specifies two arguments: the resource whose properties are being accessed and the properties that are being accessed [37]. An abstract syntax of GetData query looks like:

\[ \text{GetData}(\text{resource}, \text{property}) = \text{value} \]

For example, query GetData(www.museum.es, title) on data presented in Figure 6 will return “Reina Sofia Museum”.

GetData also allows reverse (bottom-up) traversal of RDF graph arcs. Additionally, TAP provides two other interfaces to help graph exploration (details can be found in [37, 11]).

8.3. Sesame

Sesame [27, 9, 28] is an RDF framework with support for RDF Schema inferencing. Its main features include querying in three languages (SeRQL, RDQL, RQL), parsing and writing in several serialization syntaxes, support for MySQL, PostgreSQL, Oracle and SQL Server as well as in-memory. It can be deployed as an RDF database, with persistence in an RDBMS, or as a Java library for embedded use in applications [27].

The architecture of Sesame is presented in Figure 8 (copied from [27]). Client programs use Sesame Access APIs to access the server locally or remote through HTTP and RMI. Functional modules like Query Module (SeRQL, RQL, RDQL), Export Module (data exporting into the RDF(S) format) and Admin Module (administrative functionality) are clients of SAIL API – Storage And Interface Layer. SAIL layer is the main component of Sesame, that provides an application programming interface that abstracts from the storage device used (in-memory storage, disk-based storage, RDBMS) and takes care of inferencing. Note, that queries in SeRQL, RQL and RDQL are translated into a sequence of SAIL API calls. Therefore, substantial part of query evaluation process is done in respective query modules [28].

As already stated above, Sesame can store RDF data into a relational database (MySQL, Oracle and SQL Server) or into an object-relational database (PostgreSQL). We briefly describe relational schema for these two storage approaches in the following (details are available in [28]).

**Sesame and MySQL.** The relational schema (see Figure 9) is fixed and includes tables to store RDFS ontology (class, subclassOf, property, subPropertyOf, domain, range, type, labels, etc.) and single table TRIPLES for RDF statements.

**Sesame and PostgreSQL.** PostgreSQL is an object-relational database that supports transitive subtable relations between its tables. This feature can be naturally used to model class and property hierarchy relationships. As a result, initially Sesame created a table for every ontology class.
and property. If a class is a subclass of other classes, the table created for it will also be a subtable of the tables for the superclasses. Likewise for properties being subproperties of other properties. Instances of classes and properties are inserted as values into the appropriate tables [28]. However, this storage schema revealed several disadvantages:

- Data insertion is slow and, in particular, incremental uploads requiring creation of a new table are very expensive in PostgreSQL.
- Once created, the subtable relations are fixed, so that changes in a class or a property hierarchy result in new database schema creation.
- Cycles in a class hierarchy can not be modelled properly using subtable relations.

As a result, Sesame abandoned the subtable approach and stores all RDF statements into a single table with three columns: Subject, Predicate, Object.

### 8.4. DLDB

DLDB [47] is a knowledge base system that extends a relational database management system with ontology-based inferencing. DLDB provides support for RDFS, DAML+OIL and OWL, uses FaCT DL reasoner for inferencing and MS Access for storing annotations and inferred information. The system is positioned as

Relational schema is automatically generated based on an ontology: for each class and property, a table is generated with the corresponding class or property name. Further, for each table, a view is created which duplicates table data and additionally stores inferred data. For example, for class Artifact, that has two subclasses Sculpture and Painting (see Figure 6), the system will create:

- table Artifact, which stores all instances of the class;
- the view, which stores all instances of the class and its subclasses.

Queries for DLDB are expressed in KIF-like format using the system API and translated into SQL queries over generated views.

---

1 KIF – Knowledge Interchange Format; see http://logic.stanford.edu/kif/kif.html
8.5. DAMLJessKB and OWLJessKB

DAMLJessKB [42] and OWLJessKB [4] are memory-based description logic reasoners for DAML and OWL respectively. The key component of both tools is production system Jess (Java Expert System Shell), which serves as a rule-based reasoner and a memory-based knowledge base.

The basic approach of DAMLJessKB (OWLJessKB) is illustrated in Figure 10 (copied from [42]). The DAMLJessKB process (Figure 10(a)) includes mapping RDF XML syntax into a list of triples, mapping triples into facts in a production system, derivation of rules from the semantics of the language. Given an XML source document (Figure 10(b)), an RDF parser generates a stream of triples (Figure 10(c)). These triples are asserted into the production system and rules derived from the semantics of the language are applied (Figure 10(d)) to populate the knowledge base with the additional facts which can be entailed from the input (Figure 10(e)) [42].

DAMLJessKB (OWLJessKB) differs from most Semantic Web reasoners, which are built on description logic and general theorem provers, in that, by using a production system, it assumes a closed world (i.e., if a fact does not exist it is assumed to be false) [42].

8.6. Jena

Jena [2, 52, 51] is a Java framework for building Semantic Web applications, that provides a programmatic environment for RDF, RDFS and OWL, including a rule-based inference engine. Jena stores RDF graphs in memory or in a database and supports two forms of querying: triple match and RDQL [7]. In the following, we briefly describe persistent RDF storage, querying and inferencing in Jena.

Persistent RDF storage has evolved from Jena1 to Jena2, the second generation of the Jena toolkit. Jena1 used the normalized triple store approach (similar to Sesame) illustrated in Figure 112 (copied from [51]). The statement table stores all triples (statements) and references resources and literals tables for subjects, predicates and objects. The normalized schema in Jena1 was used for MySQL, PostgreSQL and Oracle, however Berkley DB used denormalized schema storing triples in a single table. This approach is very efficient in space, however requires a three-way join to retrieve a triple.

Jena2 exploits the denormalized schema approach (see Figure 12 (copied from [51])). Separate literals and resources tables are only used to store values of long literals and URIs respectively, when their length exceeds some threshold (can be specified). This approach takes more space (literal and URI values may be stored repeatedly), however shows better retrieval performance.

Additionally, Jena2 supports property tables to augment the triple store. A property table clusters multiple property values for the same subject in one table row. It uses one column for the subject and one column for each predicate value. In general, a property table stores all statements for its set of predicates. For example, a Person property table might store all statements about names, addresses and phones. There would be no statements with those predicates in the triple store [52].

Finally, Jena2 supports property-class tables. A property-class table is a special kind of property table that serves two purposes. It records all instances of a specified class, i.e., resources that have that class. It also stores properties of that class, i.e., each property in the table must have the class as its domain. Thus, a property-class table has two or more columns: one for the subject resource, a second boolean column indicating if the subject has been explicitly asserted as a class member (as op-
posed to inferred as a member), and zero or more columns for property values [51].

There is also research on developing application-specific schema for RDF [52]. However, in Jena, application-specific schema is not based on class or property hierarchies (like in DLDB), but on properties of RDF data. Jena provides the following tools to enable application-specific schema approach:

1. Data mining tool to discover patterns in RDF graphs and RDF query logs. For example, the tool can discover property co-occurrence patterns to suggest candidate property tables.

2. RDF synthetic data generator with support of high degree control over generated data characteristics. For example, it is capable of modeling relationships among class instances.

More details on Jena application-specific schema are available in [33, 52].

Jena2 supports two forms of querying: triple match (also known as find operation) and RDQL [7]. A triple match retrieves all triples that match template (Subject, Predicate, Object), where Subject, Predicate and Object can be constants or don’t-care wildcards. RDQL queries are compiled into conjunction of triple match templates with variables to specify joins. Property tables may complicate query processing. For example, when Predicate is not specified (don’t-care or variable), all property tables and the triple store must be scanned. It is difficult to generate a single SQL query in this case.

Inference in Jena is implemented through inference graphs. The inference graph API can be used to access a range of different inference engines. Jena ships with a pair of generic rule engines - a forward chaining engine based on the standard RETE algorithm and backward chaining engine which uses SLD style backtracking with tabling. These engines can run in a hybrid mode (forward-backward chaining). Jena includes built-in rule sets which enable these rule engines to provide RDFS and a useful subset of OWL inferences. The architecture is extensible to allow external rule engines to be used [52].

The current rule engines will work over persistent graphs but all inferencing is done in memory, outside the database engine. Some RDF systems support inferencing directly in the storage engine by precomputing and storing some or all of the entailments (e.g., Sesame). The hybrid rule engine approach of Jena lends itself to this - the partial closure generated by the forward rules could be persisted in the database with the backward rules being used to rewrite queries dynamically [52].

8.7. KAON

The Karlsruhe Ontology (KAON) [3, 22] is an open-source ontology management infrastructure targeted for business applications. It includes a comprehensive tool suite allowing easy ontology creation and management and provides a framework for building ontology-based applications. An important focus of KAON is scalable and efficient reasoning with ontologies.

The conceptual architecture of KAON is illustrated in Figure 13 (copied from [22]). Client applications and services are not of this paper focus, we refer the reader to [22] for more details.

The primary role of the middleware layer is to provide an abstraction for ontology access, which is achieved by means of the KAON API, which provides a unified interface to all clients. An important component on this layer is KAON RDF Server, which is a data source specialized in storing RDF data and is responsible for data persistence, concurrent modification, transaction support. Another role of the middleware layer is the dynamic instantiation and delegation of requests to the underlying external services layer.

Roles of the data and remote service layer include:

- access to physical data stores such as databases or file systems.
- management of external services, such as reasoning engines.

KAON can store RDF data in memory, a database (SQL2-compatible DBMS, such as SQL Server, Oracle, PostgreSQL, etc.), file system, etc. The database persistent storage supports triple store and more complex relational schema – Metamodel structure for Engineering Server (more details are available in [45]).
<table>
<thead>
<tr>
<th>KB System</th>
<th>Storage Type</th>
<th>Database Schema</th>
<th>Update Support</th>
<th>Inference Support</th>
<th>Query Language</th>
<th>Scalability/Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semagix Freedom</td>
<td>RDBMS</td>
<td>Triple store</td>
<td>Yes</td>
<td>Yes</td>
<td>API (Ontology and metabase)</td>
<td>Over a million of instances; 10ms per query for 64 concurrent users</td>
</tr>
<tr>
<td>TAP</td>
<td>File system, RDBMS (MySQL,BerkleyDB)</td>
<td>Triple store</td>
<td>Yes</td>
<td></td>
<td>GetData (RDF graph)</td>
<td></td>
</tr>
<tr>
<td>Sesame</td>
<td>Memory, ORDDBMS (PostgreSQL), RDBMS (MySQL,Oracle,etc.)</td>
<td>Triple store, ?</td>
<td>Yes</td>
<td>Yes</td>
<td>SaRQL, RQL, RDQL</td>
<td>Tested on 3 (DB) and 1 (memory) million triples; loading time for 6.8 million triples (583MB) is too long (order of weeks); linear scalability on querying</td>
</tr>
<tr>
<td>DLDB</td>
<td>RDBMS (Access)</td>
<td>Application-specific (separate table for instances of each class and property)</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>Tested on 6.8 million triples (583MB); linear scalability on data loading, repository size and querying</td>
</tr>
<tr>
<td>DAMLJessKB OWLJessKB</td>
<td>Memory</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes; Jess reasoner</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jena</td>
<td>Memory, ORDDBMS (PostgreSQL), RDBMS (MySQL,Oracle,etc.)</td>
<td>Triple store, property-class tables, application-specific (data mining)</td>
<td>Yes</td>
<td>Yes; built-in or external reasoner</td>
<td>Triple match, RDQL</td>
<td>Tested on small data sets – ten thousands of triples; querying is, in general, slower then for Sesame [39]</td>
</tr>
<tr>
<td>KAON</td>
<td>Memory, file system, ORDDBMS (PostgreSQL), RDBMS (SQL Server, Oracle,etc.)</td>
<td>Triple store, metamodel structure</td>
<td>Yes</td>
<td>Yes; external reasoner only</td>
<td>RDF-QEL (Datalog-like)</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Summary Table of System Features

Querying is supported using RDF-QEL query language (designed in the Edutella project). Inferencing is possible through external reasoning services, and is not implemented in KAON directly.

### 8.8. Summary Table of System Features

To summarize our discussion of existing KB systems, we compare their features in Table 1.

Additionally, detailed evaluation of a few systems (Sesame, DLDB, DAMLJessKB and OWLJessKB) for large DAML+OIL and OWL data sets can be found in [39, 38]. [39] compares system performance in terms of data loading, querying (14 test queries), query completeness and soundness.

### 8.9. Other Systems

There exist many other systems that support RDF(S)/DAML+OIL/OWL storage, querying and reasoning: 4Suite, DAML DB, EOR, Haystack, Inkling, Parka Database, rdDB, RDFLib, RDFStore, RDFSuite, Redfoot, Redland, Edutella, etc. Summary information about them can be found in specialized surveys, such as [14], [16] and [44]. Our section complements those surveys with new systems (Semagix Freedom, TAP, OWLJessKB, DLDB), as well as with more recent specifications for Sesame, Jena and KAON.

### 9. Conclusions

In this survey, we reviewed papers related to Semantic Web search: semantics for search, ontology-based search, query languages and knowledge base systems that enable Semantic Web search. The paper complements existing surveys with new systems, more detailed and recent specifications on some of systems.

Our conclusions drawn from this survey include:

- Existing RDF(S) query languages are not complete, lacking expressivity.
- Lack of standards on query languages, rules and inferencing resulted in many different incompatible implementations, which makes them difficult to compare, learn, exploit, and so forth. It is not always clear which inferencing path a reasoner should choose as correct.
- Existing systems are not mature and have the following limitations: (1) performance and scalability, which are significantly influenced by inefficient storage schemas and inference algorithms, (2) low expressivity of implemented query languages, (3) completeness of query results, which is influenced by insufficient support of inference, (4) soundness of query results, which is influenced by incorrect inference, (5) lack of performance/scalability experiments.

### References


