RDF Storage and Retrieval Systems

Alice Hertel[1], Jeen Broekstra[2], and Heiner Stuckenschmidt[3]

[1] Fraunhofer Institute for Information and Data Processing, Fraunhoferstr. 1, 76131 Karlsruhe, Germany alice.hertel@iitb.fraunhofer.de
[2] Technische Universiteit Eindhoven, PO Box 513, 5600 MB Eindhoven, The Netherlands j.broekstra@tue.nl
[3] University of Mannheim, Schloss, 68131 Mannheim, Germany heiner@informatik.uni-mannheim.de

Summary. Ontologies are often used to improve data access. For this purpose, existing data has to be linked to an ontology and appropriate access mechanisms have to be provided. In this chapter, we review RDF storage and retrieval technologies as a common approach for accessing ontology-based data. We discuss different storage models, typical functionalities of RDF middleware such as data model support and reasoning capabilities and RDF query languages with a special focus on SPARQL as an emerging standard. We also discuss some trends such as support for expressive ontology and rule languages.

1 Introduction

It is widely acknowledged that information access can benefit from the use of ontologies. For this purpose, available data has to be linked to concepts and relations in the corresponding ontology and access mechanisms have to be provided that support the integrated model consisting of ontology and data. The most common approach for linking data to ontologies is via an RDF representation of available data that describes the data as instances of the corresponding ontology that is represented in terms of an RDF Schema (compare chapter 3). Due to the practical relevance of data access based on RDF and RDF Schema, a lot of effort has been spent on the development of corresponding storage and retrieval infrastructures.

In this chapter, we summarize the state of the art with respect to existing storage and retrieval technologies for RDF data. In particular, we first review the general architecture of RDF infrastructures that normally consist of a storage and a middleware layer. We discuss important aspects of these layers covering different storage formats for RDF data, common middleware functionalities such as RDF Schema reasoning and basic operations for data access and manipulation. Throughout the chapter, we discuss these aspects on a general level and only point to particular systems to provide examples of concrete

implementations. We further discuss RDF query languages as the most common interface for interacting with ontology-based RDF data and present the SPARQL language in more detail. In section 7 we also provide a very brief overview of existing approaches to extend RDF storage and retrieval systems to support more complex ontology languages than RDF Schema. We close with a discussion of current trends and speculate about future developments.

2 Architecture of RDF Stores

An RDF store allows storage of RDF data and schema information, and provides methods to access that information. Thus, the two primary components of an RDF store are a repository and a middleware that builds on top of that repository. The middleware can be further divided into components as the access methods can be categorized into methods for adding, deleting, querying and exporting data. To describe the different components in detail, we assume a layered architecture as proposed in [1] and regard the layers from the bottom up.

Different repositories are imaginable, e.g. main memory, files or databases, but the access methods should remain the same. Thus, it is reasonable to encapsulate the access to the repository in an own layer, which provides well-defined interfaces to the upper layers and can be exchanged if another repository is used. The inference support also resides in this layer as close to the repository as possible. Sesame [1] implements such a layer and calls it the Storage And Inference Layer (SAIL).

The above mentioned access methods are located on a higher level and address the interfaces of the SAIL (or directly address the repository if there is no SAIL implementation). According to the different requirements of each access method they can be realized in different modules: The *admin module* provides the functionality for adding new data to and deleting data from the RDF store. Especially when loading data from files this requires parsing and validating RDF, so an RDF parser and an RDF validator are usually part of the admin module. The *query module* handles queries to the RDF store. As these queries can be formulated in any kind of RDF query language, several query modules may be necessary, each implementing a parser and handler for one query language. Finally, the *export module* allows a dump of the RDF store into files for data exchange with other systems.

These modules can be accessed locally or remotely, e.g. using SOAP or RMI. This is why the highest layer in the middleware contains protocol handlers that can manage different access modes. Fig. 1 shows the generic architecture as proposed in [1]. This architecture is not only valid for Sesame - other RDF store implementations have a similar modular structure reflecting the different aspects of an RDF store.

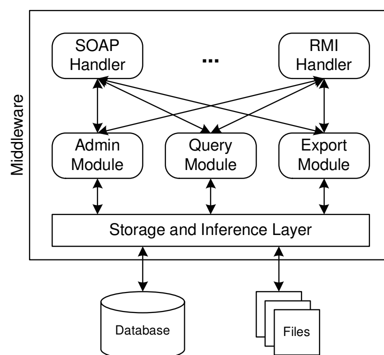

Fig. 1. Generic Architecture of an RDF Store (Sesame)

3 Storing RDF Data

RDF schemas and instances can be efficiently accessed and manipulated in main memory. For persistent storage the data can be serialized to files, but for large amounts of data the use of a database management system is more reasonable. Examining currently existing RDF stores we found that they are using relational and object-relational database management systems (RDBMS and ORDBMS).

Storing RDF data in a relational database requires an appropriate table design. There are different approaches that can be classified in generic schemas, i.e. schemas that do not depend on the ontology, and ontology specific schemas. In the following we describe the most important table designs showing their advantages and shortcomings.

3.1 Generic Schemas

The most simple generic schema is the triple store with only one table required in the database. The table contains three columns named `Subject`, `Predicate` and `Object`, thus reflecting the triple nature of RDF statements. This corresponds to the *vertical representation* for storing objects in a table in [2].

The greatest advantage of this schema is that no restructuring is required if the ontology changes. Adding new classes and properties to the ontology can be realized by a simple `INSERT` command in the table. On the other hand, performing a query means searching the whole database and queries involving joins become very expensive. Another aspect is that the class hierarchy cannot be modeled in this schema, what makes queries for all instances of a class rather complex.

The triple store can be used in its pure form [3], but most existing systems add several modifications to improve performance or maintainability. A

common approach, the so-called *normalized triple store*, is adding two further tables to store resource URIs and literals separately as shown in Fig. 2, which requires significantly less storage space [4]. Furthermore, a hybrid of the simple and the normalized triple store can be used, allowing to store the values themselves either in the triple table or in the resources table [5].

Triples:

Subject	Predicate	IsLiteral	Object
r1	*r2*	*False*	*r3*
r1	*r4*	*True*	*l1*
...	...	...	...

Resources:

ID	URI
r1	*...#1*
r2	*...#2*
...	...

Literals:

ID	Value
l1	*Value1*
...	...
...	...

Fig. 2. Normalized Triple Store

In a further refinement, the `Triples` table can be split horizontally into several tables, each modeling an RDF(S) property:

- `SubConcept` for the `rdfs:subClassOf` property, storing the class hierarchy
- `SubProperty` for the `rdfs:subPropertyOf` property, storing the property hierarchy
- `PropertyDomain` for the `rdfs:domain` property, storing the domains and cardinalities of properties
- `PropertyRange` for the `rdfs:range` property, storing the ranges of properties
- `ConceptInstances` for the `rdf:type` property, storing class instances
- `PropertyInstances` for the `rdf:type` property, storing property instances
- `AttributeInstances` for the `rdf:type` property, storing instances of properties with literal values

These tables only need two columns for `Subject` and `Object`. The table names implicitly contain the predicates. This schema separates the ontology schema from its instances, explicitly models class and property hierarchies and distinguishes between class-valued and literal-valued properties [1, 6].

3.2 Ontology Specific Schemas

Ontology specific schemas are changing when the ontology changes, i.e. when classes or properties are added or removed. The basic schema consists of one table with one column for the instance ID, one for the class name and one for each property in the ontology. Thus, one row in the table corresponds to one instance. This schema is corresponding to the *horizontal representation* in [2] and obviously has several drawbacks: large number of columns, high sparsity, inability to handle multi-valued properties and the need to add columns to the table when adding new properties to the ontology, just to name a few.

Horizontally splitting this schema results in the so called one-table-per-class schema: one table for each class in the ontology is created. A class table provides columns for all properties whose domain contains this class. This is tending to the classic entity-relationship-model in database design and benefits queries about all attributes and properties of an instance.

However, in this form the schema still lacks the ability to handle multi-valued properties, and properties that do not define an explicit domain must then be included in each table. Furthermore, adding new properties to the ontology again requires restructuring existing tables.

Another approach is vertically splitting the schema, what results in the one-table-per-property schema, also called the *decomposition storage model*. In this schema one table for each property is created with only two columns for subject and object. RDF(S) properties are also stored in such tables, e.g. the table for `rdf:type` contains the relationships between instances and their classes.

This approach is reflecting the particular aspect of RDF that properties are not defined inside a class. However, complex queries considering many properties have to perform many joins, and queries for all instances of a class are similarly expensive as in the generic triple schema.

In practice, a hybrid schema combining the table-per-class and table-per-property schemas is used to benefit from the advantages of both of them. This schema contains one table for each class, only storing there a unique ID for the specific instance. This replaces the modeling of the `rdf:type` property. For all other properties tables are created as described in the table-per-property-approach (Fig. 3) [7]. Thus, changes to the ontology do not require changing existing tables, as adding a new class or property results in creating a new table in the database.

ClassA:

ID
...#1
...

Property1:

Subject	Object
...#1	*...#3*
...	...

ClassB:

ID
...#3
...

Fig. 3. Hybrid Schema

A possible modification of this schema is separating the ontology from the instances. In this case, only instances are stored in the tables described above. Information about the ontology schema is stored separately in four additional tables `Class`, `Property`, `SubClass` and `SubProperty` [8]. These tables can be further refined storing only the property ID in the `Property` table and the domain and range of the property in own tables `Domain` and `Range` [1]. This approach is similar to the refined generic schema, where the ontology is stored the same way and only the storage of instances is different.

To reduce the number of tables, single-valued properties with a literal as range can be stored in the class tables. Adding new attributes would then require to change existing tables. Another variation is to store all class instances in one table called `Instances`. This is especially usefull for ontologies where there is a large number of classes with only few or no instances [8].

3.3 Further Issues

There are further issues that may require an extension of the triple-based schemas and thus are affecting the design of the database tables:

- Storing multiple ontologies in one database
- Storing statements from multiple documents in one database

Both points are concerning the aspect of provenance, which means keeping track of the source an RDF statement is coming from.

When storing multiple ontologies in one database it should be considered that classes, and consequently the corresponding tables, can have the same name. Therefore, either the tables have to be named with a prefix referring to the source ontology [7] or this reference is stored in an additional attribute for every statement. A similar situation arises for storing multiple documents in one database. Especially, when there are contradicting statements it is important to know the source of each statement. Again, an additional attribute denoting the source document helps solving the problem [7].

The concept of named graphs [9] is including both issues. The main idea is that each document or ontology is modeled as a graph with a distinct name, mostly a URI. This name is stored as an additional attribute, thus extending RDF statements from triples to so-called quads. For the database schemas described above this means adding a fourth column to the tables and potentially storing the names of all graphs in a further table.

3.4 Object-Oriented Features

Current ORDBMS provide the *subtable* facility which allows for a better modeling of the subclass and subproperty relationships. The table of a subclass is then created as a subtable of the superclass table. Consequently, querying for all instances of a class does not require searching for all triples with the `rdfs:subClassOf` property or looking up a `SubClass` table. However, this feature should be used carefully, as a new subtable can only be added at the bottom of the hierarchy. Otherwise, the complete table hierarchy needs to be rebuilt [1, 8].

Oracle[4] offers another object-relational feature: an own datatype to store RDF based on a graph data model. RDF triples can be persisted, indexed and queried, similar to other object-relational data types.

[4] See http://download-east.oracle.com/otndocs/tech/semantic_web/pdf/rdfrm.pdf

Although the RDF model has several object-oriented characteristics and most RDF stores are internally working with an object model, approaches to store RDF data and schema information using object database management systems (ODBMS) are rarely known. (Object-)Relational databases are still predominant, when large amounts of data have to be persisted on a server, and object databases did not and will most probably not replace them. However, new developments of ODBMS may show some advantages over RDBMS in certain applications, e.g. for embeddable persistence solutions in mobile devices. This is why storing ontologies in an ODBMS is worth a closer look.

4 RDF Middleware

What we call RDF middleware is the layer implementing the access to the physical RDF data store. Besides an inference mechanism, the access layer should provide functions for creating, querying and deleting data in the store. While adding data requires parsing and ideally a validation of the incoming RDF sentences, querying the RDF store needs the implementation of some kind of query language as well as an interpretation and a translation of this query language into calls to the physical RDF storage. Another important feature of this layer is the possibility to export data to a file for exchange with other systems.

4.1 Inference for RDF

Inference for RDF is specified by the RDF(S) entailment rules described in [12]. The practice-relevant rules can be roughly divided into the two following groups:

- Inferring the transitive closure for the properties `rdfs:subClassOf` and `rdfs:subPropertyOf`
- Inferring class memberships analysing the use of properties and their domains and/or ranges

One approach is to compute the transitive closure using a recursive algorithm and to store it in database views. This algorithm constructs a view for each class, starting with the class table and adding the views of all of its subclasses examining the statements with the `rdfs:subClassOf` relationship in the database. Analogously, a view for each property is constructed from the `rdfs:subPropertyOf` relationships. A similar algorithm can be used to infer class membership from the properties of an instance [7].

An alternative is to use a production rule system that generates new facts from existing ones by forward chaining or applies backward chaining on a query presented to the system. This brings up an important aspect of the inference, namely the time, when the inference is executed. There are two possibilities:

- Inference in advance (*eager evaluation*)
- Inference at query runtime (*lazy evaluation*)

The eager evaluation is computing the deductive closure in advance, so the time to evaluate a query is reduced [1]. However, it also may cause a dramatic increase of the amount of stored data, potentially generating entailments that are rarely matching queries. In contrast, the lazy evaluation only performs evaluation of entailments matched by a given query, so no unused entailments need to be generated and stored. This significantly increases the query processing time. A compromise is to use both methods: those entailment rules that generate fewer entailments are evaluated in advance, while those requiring more storage space and less evaluation time are evaluated at query time [4].

Although we describe the inference mechanism as part of the middleware, the algorithms can also be defined as stored procedures in the database, leaving the inference task to the database management system. This depends on the capabilities of the DBMS used for storing the data.

4.2 Querying Data

For formulating a query to the RDF store there are several approaches:

- Implementing a proprietary query API
- Implementing a query language

Proprietary query APIs are defining their own query format. E.g. DLDB [7] is using conjunctive queries composed of atoms whose structure is based on First Order Logic. Constructing an SQL query is done through a translation algorithm that substitutes predicates and variables by table and column names. Another possibility is to create an own query language, e.g. KAON Query [6] or SeRQL [1].

Most RDF stores are using one of the common RDF query languages like RQL, RDQL or SPARQL [1, 4, 8]. This means implementing a parser that analyses the syntax of this query language. A potential intermediate step is to translate the parsed query into relational calculus [4], a graph [8] or the object model [1] to capture the query semantics. After that, the SQL query sentence is formed and sent to the database.

The syntax of the created SQL query usually depends on the underlying DBMS. This is why the implementation of an additional intermediate layer is reasonable that abstracts from the actual storage mechanism offering storage and retrieval functions. An example for that is the Storage And Inference Layer (SAIL) in Sesame. The layer can be exchanged according to the used DBMS and can even be placed on top of another SAIL to offer further functionality like caching recent query results [1].

An important aspect for accessing data is **query optimization**. It can be left to the database system, considering the sophisticated evaluation and optimization mechanisms of modern RDBMS. So, the query must be translated

to SQL as completely as possible. This is the approach used by most RDF stores. Another approach is optimizing the query in the middleware itself, which is particularly interesting if the query engine should be independent of the underlying storage like in Sesame. Here, the query is translated into a set of SQL queries and joins or other operations are performed in the query engine. This not only requires an optimization strategy but also implies a transaction management, because one RQL query can result in multiple SQL queries and the state of the database must not change until all these queries are executed [1].

4.3 Adding, Deleting and Exporting Data

Adding data to the RDF store can be realized by creating new concepts, properties or instances in main memory using the API and then calling a function to store them into the knowledge base [5, 6]. Another possibility is reading RDF data from a file or an online source, which is implemented by all RDF stores as it is important for loading an ontology. Reading RDF data requires an RDF parser for reading in the statements and mapping them on the object model or directly on the database schema. Most systems use a parser that reads the RDF/XML notation, e.g. the ARP (Another RDF Parser), which is part of the Jena toolkit, or the Raptor RDF parser. Optionally, an RDF validator can be used to check the incoming data for correctness and for compliance with already loaded schemas [8]. In this case, the schemas should be loaded before the instances.

Delete operations in RDF stores have to be handled very carefully. While completely clearing the store is a quite simple function, deleting single statements can entail the deletion of other related statements. This not only requires recomputing the deductive closure for the RDF store, but also a mechanism for truth maintenance. Hence, deletions become quite costly [1].

To exchange data with other systems an export mechanism is required. Most RDF stores implement such an export function which allows to serialize the ontology and instance data from the RDF store into a file. The common formats for serializing RDF are N-Triples, N3 notation and RDF/XML notation.

5 RDF Query Languages

As mentioned in the preceding section, the use of query languages is the most common way of interacting with an RDF store. Many query languages already exist that could, in principle, be used to interact with RDF data. The most obvious example is SQL, the standard query language for relational databases. In this section, we will explore what properties a query language for semistructured data, and in particular for RDF, should have, and what the difference is with existing approaches such as SQL. We will then discuss several

proposals for query languages. In particular, we will describe the SPARQL query language in more detail.

5.1 General Properties of Query Languages

We can identify several general properties with which one can characterize query languages. Here, we name six such properties.

- *Expressiveness:* Expressiveness indicates how powerful queries can be formulated in a given language. Ideally, a query language should be expressive enough to allow the retrieval of any arbitrary combination of values from the queried model, that is, be *complete* with respect to its datamodel. Usually, expressiveness is restricted to maintain other properties such as safety and to allow an efficient (and optimizable) execution of queries.
- *Closure:* The closure property requires that the results of an operation are again elements of the data model. This means that if a query language operates on a graph data model, the query results would again have to be graphs.
- *Adequacy:* A query language is called adequate if it uses all concepts of the underlying data model. This property therefore complements the closure property: For the closure, a query result must not be outside the data model, for adequacy the entire data model needs to be exploited.
- *Orthogonality:* The orthogonality of a query language requires that all operations may be used independently of the usage context.
- *Safety:* A query language is considered safe, if every query that is syntactically correct returns a finite set of results (on a finite data set). Typical concepts that cause query languages to be unsafe are recursion and negation.

5.2 Path Expressions

One of the main distinguishing features of query languages for semi-structured data is their ability to reach to arbitrary depths in the data graph. To do this, these languages all use the notion of path expressions. A path expression is a simple query, the result of which, for a given data graph, is a set of nodes. For example, consider the following bit of XML:

```
<?xml version="1.0"?>
<body>
  This page is written by
  <author>Jeen Broekstra</author>.
  <location>
    His tel.nr. at work is <tel>3686</tel>,
    his number at home is <tel>555722</tel>, and his
    room number is <room>HG7.76</room>.
  </location>
</body>
```

The result of the path expression `body.location.tel` would be the set of nodes with the associated values '3686', '555722'.

Many useful regular expressions can be used in path expressions to facilitate more complex expressions than just specification of the complete path. For example, a regular expression `location|name` specifies either a `location` node or a `name` node. Another very useful pattern is the wildcard, which matches any node label. Using the symbol to express this, `body.tel` matches any path consisting of a `body` node followed by any node, followed by a `tel` node. Also, closure operations, like arbitrary repeats of a regular expression can be used. For example, `body*.tel` specifies the set of `tel` nodes that occur at arbitrary depth within the `body` node. At another level of abstraction, regular expressions can also be used to express matches on the actual string format of labels. For example the regular expression `body."[aA]uthor"` matches any `author` node within the `body`, possibly with the first letter capitalized.

Path expressions, although they are an essential feature of query languages for semistructured data, can only return a subset of nodes in the database. They can not construct new nodes, perform joins, or test values stored in the database. In other words: path expressions are necessary but not sufficient for a good query language on semistructured data. A query language that lacks path expressions cannot be considered adequate, nor sufficiently expressive for querying semistructured data.

5.3 Why not just SQL?

For strictly relational data (as opposed to semistructured data), SQL is by far the most widely supported query language, including support for large data storage, efficient indexing schemes, query optimizers, etc. It would therefore be attractive if we could use this robust and widely available technology for our purposes of querying semistructured data. Unfortunately, this can only be done at the cost of a very large gap between the data model in the repository (e.g. RDF) and the data-model on which the query language is based (the relational model).

To exemplify this, let us look at how the scenario would look for an XML implementation in a relational database: as a first step, we would have to encode the XML data model in the relational model. This would be possible by assigning each node in an XML tree a unique identifier, with each entry in the relational database linking such a node with all its descendants and attributes. The problems start when we want to use this as the basis for querying the XML structure: each XML query should be compiled into an SQL query on the underlying relational tables. Typically, a single XML query (such as: 'return all descendants of a given node') must be compiled into a complicated set of SQL queries. It is not even clear whether a finite set of SQL queries could be generated for every reasonable XML query.

Although perhaps attractive as a short term solution, we feel that in the long run this is not an appropriate solution. Rather, techniques for large data

storage, indexing schemes, query optimizers, etc. should be provided for the native data model (be it XML or RDF), instead of relying on these techniques for a completely different data model.

5.4 Querying RDF

RDF documents and RDF schemata can be considered at three different levels of abstraction:

1. at the *syntactic level* they are XML documents;
2. at the *structure level* they consist of a set of RDF triples;
3. at the *semantic level* they constitute one or more graphs with partially predefined semantics.

We can query these documents at each of these three levels. We will briefly consider the pros and cons of doing so for each level in the next few sections.

Querying at the Syntactic Level

As we have seen previously, RDF models can be written down in XML notation. It would therefore seem reasonable to assume that we can query RDF using an XML query language (e.g. XQuery[5]). However, this approach disregards the fact that RDF is not just an XML notation but has its own data model that is different from the XML tree structure: whereas XML is an ordered, node-labeled tree structure, RDF is an unordered, node- and edge-labeled graph structure. XML querying techniques have no functionality for dealing with differentiating between node and edge labels or with the absence of order or a tree root. Moreover, relationships in the RDF data that are not immediately apparent from the XML tree structure become very hard to query in this approach.

Querying at the Structure Level

When we abstract from the syntax any RDF document represents a set of triples, each triple representing a statement of the form subject-predicate-object. A number of query languages have been proposed and implemented that regard RDF documents as sets of such triples, and that allow to query such a triple set in various ways.

However, querying at this level means that we now interpret any RDF model *only* as a set of triples, including those elements which have been given special semantics in RDF Schema. For example, the fact that `rdfs:subClassOf` is a transitive relation is ignored at this level.

[5] See `http://www.w3.org/TR/xquery/`

Querying at the Semantic Level

When we consider RDF models at the semantic level we query the full knowledge of everything that the RDF model entails, and not just those facts that happen to be represented explicitly.

There are at least two options to achieve this goal:

1. Compute and store the deductive closure of a graph as a basis for querying.
2. Let a query processor infer new statements as needed per query.

While the choice of an RDF query language is, in principle, independent of the choice made in this respect, the fact remains that most RDF query languages have been designed to query a simple triple store and have no specific functionality or semantics to discriminate between data and schema information.

5.5 SPARQL

The SPARQL Query Language [25] is a W3C Candidate Recommendation for querying RDF, and as such is fast becoming the standard query language for this purpose. In September 2006, almost all major RDF query tools have begun implementing support for the SPARQL query language. Even though other query languages (e.g. SeRQL [1], RQL [26], RDQL [27]) have existed longer and have a more mature implementation base and a more expressive feature set, they typically are supported by only one or two tools, hindering interoperability. Several surveys and comparitive analyses of these different query languages have been published, a fairly comprehensive one can be found in [29]. In this chapter, we will concentrate on the SPARQL query language, giving a brief introduction into its basic usage, highlighting some interesting features. For a formal analysis of the semantics and complexity of the SPARQL language, we recommend reading [28].

Basic Queries

The SPARQL query language is based on matching graph patterns. The simplest graph pattern is the triple pattern, which is like an RDF triple, but with the possibility of a variable instead of an RDF term in the subject, predicate or object positions. Combining triple patterns gives a basic graph pattern, where an exact match to a graph is needed to fulfill a pattern.

As a simple example, consider the following query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?c
WHERE
{
    ?c rdf:type rdfs:Class .
}
```

The above query retrieves all triple patterns where the property is `rdf:type` and the object is `rdfs:Class`. In other words, this query, when executed, will retrieve all classes.

Note that like the namespace mechanism we have previously seen for writing down RDF in XML, SPARQL allows us to define prefixes for namespaces and use these in the query pattern, to make queries shorter and easier to read. In the rest of this chapter, we will omit the declaration of the 'rdf' and 'rdfs' prefixes, for brevity.

To get all instances of a particular class, for example the FOAF vocabulary class 'Person', we write:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?i
WHERE
{
  ?i rdf:type foaf:Person .
}
```

SPARQL makes no explicit commitment to support RDFS semantics. Therefore, the result of this query depends on whether or not the system answering the query supports RDFS semantics. If it does, then the result of this query will include all instances of the subclasses of `Person` as well. If it does not support RDFS semantics, then it will only retrieve those instances that are explicitly of type 'Person'.

Using `select-from-where`

As in SQL, SPARQL queries have a SELECT-FROM-WHERE structure:

`SELECT` specifies the *projection*: the number and order of retrieved data

`FROM` is used to specify the source being queried. This clause is optional; when not specified we can simply assume we are querying the knowledge base of a particular system.

`WHERE` imposes constraints on possible solutions in the form of graph pattern templates and boolean constraints.

For example, to retrieve all e-mail addresses of persons, we can write

```
SELECT ?x ?y
WHERE
{
  ?x foaf:mbox ?y .
}
```

Here `?x` and `?y` are variables, and `?x foaf:mbox ?y` represents a resource-property-value triple pattern.

We can create more elaborate graph patterns to get more complex information from our queries. For example, to retrieve all persons with name "Bob" and their phone numbers, we can write

```
SELECT ?x ?y
WHERE
{
    ?x foaf:name "Bob";
       foaf:mbox ?y .
}
```

Here `?x foaf:name "Bob"` collects all resources which have a name ”Bob”, as discussed, and binds the result to the variable `?x`. The second pattern collects all triples with predicate `mbox`. There is an *implicit join* here, in that we restrict the second pattern only to those triples, the subject of which is in the variable `?x`. Note that in this case we use a bit of syntax-shortcut as well: we use a semi-column to indicate that the following triple pattern shares its subject with the previous one, so the above query is equivalent to writing down:

```
SELECT ?x ?y
WHERE
{
    ?x foaf:name "Bob" .
    ?x foaf:mbox ?y .
}
```

We demonstrate an *explicit join* by a query that retrieves the name of all persons known by the person with name ”Bob”.

```
SELECT ?n
WHERE
{
   ?x rdf:type foaf:Person ;
      foaf:name ?n .
   ?c foaf:name "Bob" ;
      foaf:knows ?y .
   FILTER (?x = ?y).
}
```

In SPARQL, we use a FILTER condition to indicate a boolean constraint. In this case, the constraint is the explicit join of the variables `?x` and `?y` by using an equality (`=`) operator.

Optional Patterns

The graph patterns we have seen so far are mandatory patterns: either the knowledge base matches the complete pattern, in which case an answer is returned, or it does not, in which case the query does not produce a result. However, in many cases we may wish to be more flexible. Consider, for example, the following bit of RDF:

```
<foaf:Person rdf:about="#bob">
 <foaf:name>Bob</foaf:name>
</foaf:Person>

<foaf:Person rdf:about="#alice">
 <foaf:name>Alice</foaf:name>
 <foaf:mbox>alice@example.org</foaf:mbox>
</foaf:Person>
```

As you can see, this fragment contains information on two people. For one person it only lists the name, for the other it also lists the e-mail address. Now, we want to query for all people and their e-mail addresses:

```
SELECT ?name ?email
WHERE
{ ?x rdf:type foaf:Person ;
      foaf:name ?name ;
      foaf:mbox ?email .
}
```

The result of this query would be:

?name	?email
Alice	alice@example.org

So, despite the fact that Bob is listed as a person, the query does not return him: the query pattern does not match because he has no e-mail address.

As a solution we can adapt the query to use an optional pattern:

```
SELECT ?name ?email
WHERE
{ ?x rdf:type foaf:Person ;
      foaf:name ?name .
   OPTIONAL { ?x  foaf:mbox ?email }
}
```

The meaning is roughly "give us all the names of persons, and *if known* also their e-mail address" and the result looks like this:

?name	?email
Bob	
Alice	alice@example.org

This covers the basics of the SPARQL query language. For a full overview of the SPARQL language and an explanation of more advanced features, such as named graphs, we recommend reading the SPARQL specification at `http://www.w3.org/TR/rdf-sparql-query/`.

6 Scalability of RDF Stores

In terms of data storage and retrieval, scalability and performance is a very important issue. The performance of an RDF store depends on various factors: the underlying database system, the database representation of the RDF schema and instances, the efficiency of the query engine, and the performance of the inference engine. A detailed overview of the scalability and performance of different RDF stores would be out of scope of this chapter, but we can mention some interesting points.

Theoharis et al. [10] benchmarked different database representations and provide detailed results for the approaches described in section 3. In this

evaluation the onotology specific schema in its hybrid form performs better in terms of query execution times of taxonomic queries than the generic schemas. Although only one sort of queries has been evaluated this shows the weakness of the generic schemas. However, there is always a trade-off between the query execution times and the overhead for ontology evolution and table management: ontology specific schemas suffer from potentially large numbers of tables, and from the need to change the database schema when adding or deleting a class or property in the ontology.

An elaborate method and toolset to evaluate Semantic Web repositories as a whole is the Lehigh University Benchmark (LUBM) [11]. Although it is focussing on OWL applications, the LUBM can be applied to most of the RDF stores mentioned above, but there are only few evaluations available. LUBM provides means to generate a test dataset, several test queries, support for different degrees of reasoning as well as multiple performance metrics for load time, repository size, query response time, and query completeness and soundness.

The W3C maintains a web site recording the size of the largest deployed installations of triple stores[6]. End of february 2008 the site reports a number of systems that have been tested with about one billion statements. The largest data set is reported by the YARS2 System that is claimed to be able to store 7 billion triples generated using the LUBM benchmark.

7 Beyond RDF Schema

While the development of storage and retrieval systems for semantic data so far has been focussed on supporting RDF and RDF Schema there is also an interest in extending available infrastructures to more expressive languages. In particular, supporting more expressive ontology languages such as OWL-Lite and OWL-DL as well as expressive rule languages is a subject of active work. Other activities include the extension of representation and query languages with advanced features such as time [22], preference and uncertainty (e.g. [20, 21, 23]). In the following, we focus on the first kind of activities.

The most straightforward extension of existing RDF infrastructures is a support for ontologies encoded in OWL. As OWL can be serialized in RDF, the corresponding models can be stored in any RDF repository without changing the systems. The structural complexity of the OWL encoding in RDF, especially the high number of blank nodes, however, makes the access to these models rather cumbersome. In order to overcome these problems, many RDF stores use dedicated APIs as part of the middleware layer to support the storage, retrieval and manipulation of OWL ontologies. While some systems such as Jena use their own ontology API, other systems like KAON adopted the proposal for a standardized OWL API described in [24].

[6] http://esw.w3.org/topic/LargeTripleStores

Naturally, extensions to more expressive languages do not only aim at providing support at the syntactic level, but also with respect to the semantics of the corresponding languages. As mentioned above, most RDF stores support RDF Schema reasoning on the basis of a specialized set of deduction rules. A common way of extending this fixed schema is to provide support for user defined rule sets. These rule sets can be used for defining parts of the semantics of OWL [17]. Examples of systems supporting OWL-Lite reasoning on the basis of custom rule sets are Sesame, Jena and OWLIM [15]. Besides this, customized rule sets can also be used for capturing domain specific knowledge [16] and for defining efficient subsets of the RDF Schema Semantics for particular applications [18].

An alternative way of supporting OWL semantics is to provide an interface to dedicated Description Logic reasoners (e.g. Racer, FaCT or Pellet) either via specialized data structures or on the basis of the standardized DIG API (http://dig.sourceforge.net/). Systems differ in the amount of derivable knowledge that is actually integrated into the RDF model for query answering. The BOR reasoner (http://www.ontotext.com/bor/) for example computes the subsumption hierarchy of an OWL ontology and stores the derived subClass relations in the RDF model for further processing. Furthermore, there are some RDF compatible systems that implement expressive rule languages such as KAON2 which implements disjunctive datalog [14] or OntoBroker that implements F-Logic [13].

8 Conclusion

After reviewing a number of existing RDF storage and retrieval systems, we can draw some conclusions about the state of the art and general trends in the fields. On the general level, we can say that there is strong convergence of technologies which is documented by the mergence of SPARQL as a standard query language but also in terms of features that are common to different systems. For instance, we can observe that most RDF stores are not really specialized database systems for RDF data but rather an intelligent middleware that wraps existing database technology. Besides providing special support for the graph data model that is characteristic for RDF data, the main functionality provided by this middleware is support for ontological reasoning. An observation that can be made in connection with these two main functions is the fact that almost all systems rely on relational databases that provide very limited support with respect to data model and reasoning. There are very little approaches that try to delegate some of these aspects to the storage model as well by using deductive or object oriented database technologies.

With respect to further development of RDF technologies, we can identify two trends. The first one that was already mentioned in section 7 is the extension of existing systems to more expressive representation languages. In this context, rule languages (compare chapter 5) are the most promising can-

didates because it has been shown that rule-based reasoning has the potential to scale to very large data sets whereas ontological reasoning based on description logics shows serious limitations when large numbers of instances are involved. The other major direction of development concerns the scalability of RDF infrastructures to internet scale. In this context, approaches for distributed RDF processing are becoming more and more important (see also chapter 30). Both aspects, expressive representation languages and distribution are essential with respect to realizing the vision of the semantic web and are therefore important steps towards real semantic web applications.

References

1. Broekstra J (2005) Storage, Querying and Inferencing for Semantic Web Languages. PhD Thesis, Vrije Universiteit Amsterdam
2. Agrawal R, Somani A, Xu Y (2001) Storage and Querying of E-Commerce Data. In: Proceedings of the 27th Conference on Very Large Data Bases, VLDB 2001, Roma, Italy.
3. Oldakowski R, Bizer C, Westphal D (2005) RAP: RDF API for PHP. In: Proceedings of Workshop on Scripting for the Semantic Web, SFSW 2005, at 2nd European Semantic Web Conference, ESWC 2005, Heraklion, Greece.
4. Harris S, Gibbins N (2003) 3store: Efficient bulk RDF storage. In: Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems, PSSS 2003, Sanibel Island, Florida, USA.
5. Jena2 Database Interface - Database Layout. http://jena.sourceforge.net/DB/layout.html
6. Gabel T, Sure Y, Voelker J (2004) KAON - An Overview. Insititute AIFB, University of Karlsruhe. http://kaon.semanticweb.org/main_kaonOverview.pdf
7. Pan Z, Heflin J (2004) DLDB: Extending Relational Databases to Support Semantic Web Queries. Technical Report LU-CSE-04-006, Dept. of Computer Science and Engineering, Lehigh University
8. Alexaki S, Christophides V, Karvounarakis G, Plexousakis D, Tolle K (2001) The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In: Proceedings of the 2nd International Workshop on the Semantic Web, Hongkong.
9. Caroll J, Bizer C, Hayes P, Stickler P (2004) Semantic Web Publishing using Named Graphs. In: Proceedings of Workshop on Trust, Security, and Reputation on the Semantic Web, at the 3rd International Semantic Web Conference, ISWC 2004, Hiroshima, Japan.
10. Theoharis Y, Christophides V, Karvounarakis G (2005) Benchmarking Database Representations of RDF/S Stores. In: Proceedings of the 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland.
11. Guo Y, Pan Z, Heflin J (2005) LUBM: A Benchmark for OWL Knowledge Base Systems. In: Journal of Web Semantics 3(2), 2005, pp158-182.
12. RDF Semantics - W3C Recommendation. http://www.w3.org/TR/rdf-mt
13. Angele J, Lausen H (2004) Ontologies in F-Logic. In: Staab S, Studer R (eds) Handbook on Ontologies. Springer, Berlin Heidelberg New York

14. Hustadt U, Motik B, Sattler U (2007) Reasoning in Description Logics by a Reduction to Disjunctive Datalog. In: Journal of Automated Reasoning
15. Kiryakov A, Ognyanov D, Manov D (2005) OWLIM a Pragmatic Semantic Repository for OWL. In: Proceedings of the International Workshop on Scalable Semantic Web Knowledge Base Systems, SSWS 2005, WISE 2005, New York City, New York, USA
16. ter Horst H (2005) Combining RDF and Part of OWL with Rules: Semantics, Decidability, Complexity. In: Proceedings of the 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland.
17. ter Horst H (2005) Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. In: Journal of Web Semantics 3:79-15
18. Munoz J, Perez C, Gutierrez C (2007) Minimal Deductive Systems for RDF. In: Proceedings of the 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria
19. Sintek M, Decker S (2002) TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. In: Proceedings of the First International Semantic Web Conference, ISWC 2002, Sardinia, Italy
20. Bernstein A, Kiefer C (2005) iRDQL - Imprecise Queries Using Similarity Joins for Retrieval in Ontologies. In: Proceedings of the 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland.
21. Siberski W, Pan J, Thaden U (2006) Querying the Semantic Web with Preferences. In: Proceedings of the 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA
22. Gutierrez C, Hurtado C, Vaisman A (2007) Introducing Time into RDF. In: IEEE Transactions on Knowledge and Data Engineering, Special Issue on Knowledge and Data Engineering in the Semantic Web Era 19:207-218
23. Hurtado C, Poulovassilis A, Wood P (2006) A Relaxed Approach to RDF Querying. In: Proceedings of the 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA
24. Bechhofer S, Lord P, Volz R (2003) Cooking the Semantic Web with the OWL API. In: Proceedings of the 2nd International Semantic Web Conference, ISWC 2003, Sanibel Island, Florida, USA
25. Prud'hommeaux E, Seaborne A (2006) SPARQL Query Language for RDF. W3C Candidate Recommendation. http://www.w3.org/TR/rdf-sparql-query
26. Karvounarakis G, Christophides V, Plexousakis D, Alexaki S (2000) Querying Community Web Portals. Techreport, Institute of Computer Science, FORTH, Heraklion, Greece. http://www.ics.forth.gr/proj/isst/RDF/RQL/rql.pdf
27. Seaborne A (2004) RDQL - A Query Language for RDF. W3C Member Submission. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109
28. Perez J, Arenas M, Gutierrez C (2006) The Semantics and Complexity of SPARQL. In: Proceedings of the 5th International Semantic Web Conference, ISWC 2006, Athens, Georgia, USA
29. Haase P, Broekstra J, Eberhart A, Volz R (2004) A Comparison of RDF Query Languages. In: Proceedings of the 3rd International Semantic Web Conference, ISWC 2004, Hiroshima, Japan

Index

databases
  object-relational, 6
  relational, 3

OWL, 17

path expressions, 10

RDF
  query Languages, 9
  query languages, 12
  storage models, 3
RDF schema
  reasoning, 7
RDF Stores
  Sacalability of, 16
RDF stores
  architecture of, 2
  middleware, 7

SPARQL, 13
SQL, 11