Storing RDF as a Graph

Valerie Bönström, Annika Hinze, Heinz Schweppe
Institute of Computer Science
Freie Universität Berlin, Germany
{boen, hinze, schweppe}@inf.fu-berlin.de

Abstract

RDF is the first W3C standard for enriching information resources of the Web with detailed meta data. The semantics of RDF data is defined using a RDF schema. The most expressive language for querying RDF is RQL, which enables querying of semantics. In order to support RQL, a RDF storage system has to map the RDF graph model onto its storage structure. Several storage systems for RDF data have been developed, which store the RDF data as triples in a relational database. To evaluate an RQL query on those triple structures, the graph model has to be rebuilt from the triples.

In this paper, we presented a new approach to store RDF data as a graph in a object-oriented database. Our approach avoids the costly rebuilding of the graph and efficiently queries the storage structure directly. The advantages of our approach have been shown by performance test on our prototype implementation OO-Store*.*

1. Introduction

The *Resource Description Framework* (*RDF*) is the first W3C standard for enriching information resources of the Web with detailed descriptions. Information resources are, for example, web pages or books. Descriptions can be characteristics of resources, such as author or content of a web site. We call such descriptions meta data. The enrichment of the Web with meta data enables the development of the so-called *Semantic Web* [2]. The usability of the Sematic Web depends on the compatibility and processing possibilities of these meta data.

RDF supports creating and processing meta data by defining a default structure. This structure can be used for any data, independent of their character. Thus, the application areas of RDF are numerous, e.g., web-based services, peer-to-peer networks, and semantic caching models. They all have in common that huge amounts of data have to be processed when querying RDF data. Consequently, for an extensive use of RDF data, the applications need efficient storage systems. Several systems have been developed for the storage and querying of RDF data, for example, Sesame [4] and RDFSuite [1].

RDF data can be represented using XML, a triple structure or a graph. Only the graph representation enables the semantic interpretation of the RDF schema. All existing RDF systems store RDF data as triples in an object-relational database. To support the semantic interpretation, current implementations map the formal graph model onto the storage structure. In order to query the semantics, the graph has to be constructed from the triples. This mapping constitutes an unnecessary impact on the system performance.

This paper presents the new approach of storing RDF data as a graph in an object oriented database. Our approach has several advantages over the existing solutions:

- It simplifies the storage design because the graph can be directly stored without further reorganization.
- It allows to interpret the graph already in the storage without mapping.
- It efficiently uses the relationship between the RDF query language RQL and OQL for query processing.

We have verified these advantages in the performance test results achieved on our prototypical implementation.

The remainder of this paper is organized as follows: Sections 2 and 3 introduce the basic concepts of RDF and RDF schema and discuss the querying options. Our approach for storing RDF data is illustrated in Section 4, the subsequent Section 5 describes our prototypical implementation. In Section 6, we discuss the results of the performance analysis of our prototype. Finally, we conclude the paper by discussing possible future research directions in Section 7.

2. Background

In this section, we introduce the context of our study. We describe the RDF data model and the RDF schema.

Proceedings of the First Latin American Web Congress (LA-WEB 2003)
0-7695-2058-8/03 $17.00 © 2003 IEEE

2.1. RDF

The Resource Description Framework (RDF) is a language for representing meta-data. The *RDF data model* [10] defines the structure of the RDF language. The data model consists of three data types:

- Resources: All data objects described by a RDF statement are called resources. For example, resources are web sites or books.
- Properties: A specific aspect, characteristic or relation of a resource is described by a property. For example, properties are the creation date of a web site or the author of a book.
- Statements: A statement combines a resource with its describing property and the value of the property. RDF statements are the structural building blocks of the language.

A RDF statement is typically expressed as "resource - property - value" - triple, commonly written as P(R,V): A resource R has a property P with value V. These triples can also be seen as object-attribute-value triple.

Statements can also be expressed as graphs with nodes for resources and values where directed edges represent the properties. Figure 1 shows the graph of the resource R with an edge for the property P directed to the property value V.

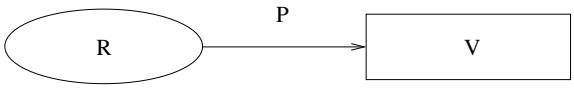


Figure 1. Graph representation of a "resource - property - value" - triple

Resources are represented in the graph as circles. Properties are represented by directed arcs. (Property-)values are represented by a box. These values are called graph endnodes. Values can also become resources if they are described by further properties, i.e., if a value forms a resource in another triple. They are then represented by a circle.

The Example 1 shows both the triple and the graph representation for the meta-data of a particular web page.

Example 1 (Different representations of a statement) *Let us consider a webpage specified by the URL www.valerie.de. This webpage has the author Valerie. We model this page as a resource with the property "creator". We show the triple and graph representation of this data.*

data: *Valerie is the creator of the resource www.valerie.de.*

triple: *hasCreator(www.valerie.de, Valerie)*

graph: *see Figure 2*

Figure 2. Graph representation of RDF triple in Example 1

2.2. RDF Schema

The RDF data model does not make any assumptions about the application area in which the data is used. There are no reserved terms to model the data. Additionally, the RDF data model has no mechanism to define names for properties or resources. For that purpose, the *RDF schema* is needed to define resource types and property names. Different RDF schemas can be defined and used for different application areas.

The W3C proposed a RDF schema called *RDF-Schema* [3]. The RDFSchema defines a basic type system for RDF data. The main RDFSchema constructs are *Class* and *Property* as resource types and *subClassOf* and *subPropertyOf* as property names. This terminology allows to declare resources as an instance of one or more classes by using the *type*-property. The *subClassOf* - property allows the specification of hierarchies of classes. The *subPropertyOf* -property defines a hierarchy of properties. The basic type system defined by RDF-Schema can be extended by new terms into a new type system.

RDF schema statements are valid RDF statements because their structure follows the structure of the RDF data model. The only difference to a pure "resource - property - value" - triple is, that an agreement about the specific meaning for reserved terms and statements has been made. Thus, the RDF schema provides a vocabulary for defining the semantics of RDF statements.

Example 2 shows the usage of the properties *type, domain, range* and *subClassOf* and resources *Resource, Property* and *Class* defined by RDFSchema.

Example 2 (Application of RDF Schema to Example 1) *The original data of Example 1 are now enriched by using the RDF Schema.*

data: *Valerie is the creator of the resource www.valerie.de.*

triples: *subClassOf(Website, Document)*
domain(hasCreator, Document)
range(hasCreator, Literal)

type(www.valerie.de, Web Site)
type(Valerie, Literal)
hasCreator(www.valerie.de, Valerie)

graph: *see Figure 3*

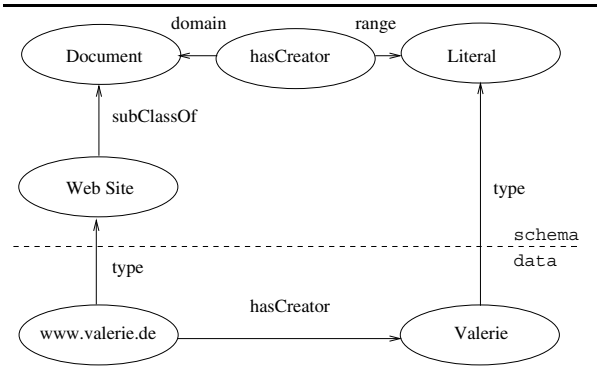


Figure 3. Graph representation of RDF Schema in Example 2

We can see that a complex graph with a specific semantics can be built.

3. Querying RDF/S

We have seen that RDF data can be represented as a set of triples (structure level) or as a graph (semantic level). A third representation form uses XML(syntax level). It has been shown that the XML representation has several drawbacks for query processing (see [4]). Therefore, we concentrate on querying of RDF data at the structure and semantic level. As we shall see, the abstraction level determines the query language.

3.1. Querying the structure level

At the structure level, queries on RDF data are based on the data model structure, i.e., on the triples. For an overview of query languages for the structure level see the survey of Tuan Anh TA [14]. The best known RDF query language on this level is *RDQL* [6], which is derived from *SQUISH* [11].

Querying RDF data on the structure level does not support the evaluation of data semantics. We illustrate this drawback in the following Example 3.

Example 3 (Querying RDF data with RDQL) *We refer to the data set defined by the triples in Example 2. The following queries are formulated in RDQL. Let us consider the following question: Return all resources of type 'Document'.*

1. *SELECT ?x WHERE (?x, <type>, <Document>)*
   This query evaluates the data at the structure level. Each single triple is evaluated. That means that each triple will be interpreted individually, but transitive correlations of several triples are not detected. In our example, 'www.valerie.de' is of type 'Web Site', and 'Web Site' is a subclass of 'Document'. Therefore, the RDFSchema semantics of 'type' and 'subClassOf' define that 'www.valerie.de' is also of type 'Document'. But the document 'www.valerie.de' is not retrieved by the query. It would only be detected if the explicit statement 'type (www.valerie.de, Document)' existed.
2. *SELECT ?x WHERE (?x, <type>, ?c1) , (?c2, <subClassOf>, <Document>) AND ?c1 = ?c2*
   This query would solve the first problem because it evaluates the triple and, additionally, considers correlations of two triples. But for further correlations (second level) of 'subClassOf'-triples this query would not work.

RDQL and other languages that query at the structure level do not provide transitive closure, which would be required for finding all correlations. In general, languages that query at the structure level do not support the interpretation of the semantics. To follow the correlations, i.e., class hierarchies, the graph representation of the data has to be used.

3.2. Querying the semantic level

At the semantic level, queries on RDF data are based on the graph representation of the data. Thus, queries do not only retrieve results about explicit statements but also consider correlations between statements. Correlations represent the data's semantics as defined by a RDF schema. RQL [13, 9] is the first standardized query language at the semantic level for RDF. RQL is a typed language with a syntax based on OQL. RQL uses the graph model, therefore, it is possible to query data, schema, or both. In Example 4, we show queries on data and schema, respectively.

Example 4 (Querying RDF data with RQL) *Again, we refer to the data set defined by the triples in Example 2. The query language is RQL. Again, we consider the following question: Return all resources of type 'Document'.*

1. *SELECT y FROM y type Document*
   A data-related RQL query is processed without interpreting the graph. The query gives all resources of type 'Document'. The pattern of the triple 'type(?, Document)' will be searched for in the graph. As before, the

result of this query is empty because the semantic has not been interpreted.

2. *SELECT $y FROM { :$y } . type Document*
   Schema related queries in RQL use a path search in the graph. The query retrieves all direct and indirect resources of type Document. Now, we receive the correct result 'www.valerie.de'.

Only queries interpreting the schema make use of the semantic level. Only those queries retrieve the correct result including transitive closure. So far, only RQL enables the interpretation of the graph and the evaluation of the complete and correct result in terms of the semantic.

This analysis of querying at the semantic level shows that RQL is currently the most powerful query language. Therefore, an efficient RDF storage system has to support RQL and the formal graph model has to be mapped onto the storage structure. Our approach for solving this problem is presented in the next section.

4. An object-oriented concept to store RDF

We briefly introduced the representation of RDF as a graph in Section 2. In this section we introduce a complete new approach to map the graph model on the storage structure. All existing concepts are based on triple storages. That means, the RDF data is stored as a set of triples, mostly in tables of a relational database. In systems that support RQL queries, the graph has to be rebuilt from the triples before the semantic can be interpreted. This is unnecessary if the graph is directly mapped onto the storage structure.

A direct mapping can be realized using an object oriented storage concept. All edges and nodes/vertices of the graph are realized as complex objects. The graph is encoded by references between these complex objects. This enables to directly map the graph onto the storage structure.

Our object-oriented concept needs different object types. The graph representation of the data in Figure 1 shows that nodes are always resources or values. In a RDF schema properties can also be used as resources. That means, both the edges and nodes, representing the RDF statements, are instances of an object type. A statement is created by the resource object referencing the value object using a property object.

Example 5 (The object-oriented storage concept for RDF) *The graph of Example 2 is mapped onto the object-oriented storage structure. Figure 4 shows that the structure of the original RDF graph is preserved. Resources, properties and values are modelled as objects. A statement is graphically represented as a directed arc from the resource object over the property object ending at the value object. For example the triple hasCreator(www.valerie.de, Valerie) is represented in the graph by having a reference from object 'www.valerie.de' across the object 'hasCreator' to 'Valerie'.*

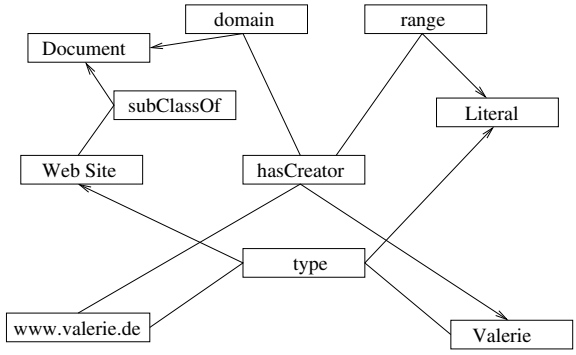


Figure 4. Object oriented concept of RDF graph in Example 2

To store the RDF graph within a object oriented concept has the following advantages:

- The storage concept is simplified due to the direct mapping of the graph onto connected objects. All nodes and edges can be translated directly into objects. All relations between nodes and edges are described by references.
- The usage of the concept is easy: all operations, such as uploading, deleting and searching data, that are needed in a RDF storage system, are executable on the stored graph without further translations.

A prototypical implementation of our approach is presented in the next section.

5. Implementation

In this section, we describe our prototypical implementation *OO-Store* of a RDF storage system based on an object-oriented database system. First, we discuss possible storage systems for object-oriented graphs. Then, we introduce the details of our specific implementation. Finally, we point out the advantages of the chosen storage design.

5.1. A storage system for the object oriented concept

The object oriented concept enables the mapping of the graph on the storage structure. For storing the graph persistently, an object oriented storage system is required. Typically either object-oriented database systems (oodb), such

as FastObjects, or persistent storage systems for objects, such as PDOM of GMD for XML objects [7], are used. Both systems allow the user to store and manage complex objects with attributes referencing each other. Thus, for the mere storage of RDF graphs, both system types may be used. But the RDF data contained in the graph has to be retrieved efficiently. An oodb enables access to objects and their relations using the query language OQL. RQL queries can be easily translated into OQL queries because the RQL syntax is based on OQL (see Section 3). Persistent storage systems do not provide any specific means for object retrieval, such as a query language. Because of the close relationship between RQL and QQL, oodb are advantageous over persistent storage systems in terms of retrieval of RDF data. Therefore, our implementation of an RDF storage system is based on an object-oriented database system, namely FastObjects [8].

5.2. RDF storage design for oodb

In an object-oriented concept to store RDF data, the resource object references the value object using a property object. Each resource, property and value is represented by a single object. If a component appears several times in the RDF triple set, multiple references to the single object are used. Our implementation for storing the RDF graph in an oodb uses three types of objects: *Resources*, *Literals*, and *Values* as shown in Figure 5. The components of an RDF triple (resource, property, and value) are represented in the database as follows: resources are represented by objects of type *Resources*. Properties are represented by objects of type *Resources*. Values are either objects of type *Resources* or *Literals*. They are of type *Resources* if they reference other objects, i.e., they additionally represent resources of other triples. They are of type *Literals* if they represent endnodes of graphs (see Section 2). If a value is solely a resource of a triple in the applied RDF schema, it has the type *Literal*. The object types Resources and Literals are subclasses of the superclass *Values* and inherit the attribute *outedges*. This attribute is used to create RDF triples from the single components: each *Resource* or *Literal* object stores a list of references to properties and values. The *outedges* attribute maps property keys ($key_1 ... key_n$) onto value lists ($list_1 ... list_n$).

Each property key points to its respective list of values. The properties and values of a specific resource can be retrieved by following the mappings in the *outedges* attribute of the resource. Thus, the RDF graph is stored in the oodb by an object/reference structure and not as single triples. The database schema stores the RDF data as a graph with an object oriented storage structure. The storage structure itself represents the formal graph model in a physical form of objects and references.

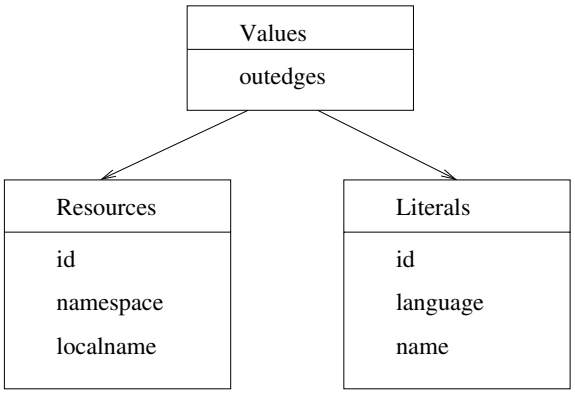


Figure 5. The object types of our oodb implementation

The Example 6 shows the transformation of a formal graph model into the object-oriented storage structure.

Example 6 (Storage structure for the oo storage concept) *We show the application of the database schema on our exemplary data structure (as introduced in Figure 5). The following tables in Figure 6 list the created objects with their attributes. In the table of Resource objects we find, for example, the object with id 9. The object represent the resource 'www.valerie.de'. The outedges attribute contains two triples 'type(www.valerie.de, Web site)' and 'hasCreator(www.valerie.de, Valerie)'.*

Objects of type Literals

id	language	value	outedges
9	en	Valerie	type–>Literal

Objects of type Resources

id	namespace	localname	outedges
1	ExpSchema	Literal	
2	ExpSchema	type	
3	ExpSchema	domain	
4	ExpSchema	range	
5	ExpSchema	subClassOf	
6	ExpData	Web Site	subClassOf–>Document
7	ExpSchema	Document	
8	ExpData	hasCreator	domain–>Document, range–>Literal
9	ExpData	www.valerie.de	type–>Web Site, hasCreator–>Valerie

Figure 6. Objects for data in Example 2

5.3. Design discussion

The direct usage of the graph, instead of triples, implies several advantages for the object-oriented DB approach compared to existing relational solutions, such as Sesame[1]. It simplifies the storage concept compared to existing storages. For example, inferred semantic data is not explicitly stored because all semantics can be derived from the graph. For more details on a discussion about differences in storage concepts see [5]. The second advantage is the direct interpretation of the data semantics instead of building the graph from triples. RQL can query the semantics of the data directly, i.e., the hierarchies of and relationships between the data. These semantic queries follow the edges inside the graph. OQL realizes this search on the stored graph by following the references of the objects.

The main advantage of storing RDF data in an object oriented database is the close relationship of OQL and RQL, which allows simple translation and processing of queries. The queries do not have to be divided into triple filters as necessary in existing relational RDF storages. There, each query is divided into a set of basic SQL statements querying triple sets, which have to be combined for the final results. In our oodb solution, each query retrieves the desired data directly. These different ways of processing a RQL query have different processing performances, as we will illustrate using two examples of RQL queries. Example 7 shows that the degree of performance difference depends on the applied SQL query optimization. As optimizations, we considered in this example join order and indexing. Example 8 shows that querying triples with SQL can demand additional query or storage effort.

Both examples show that querying the triple with SQL can retrieve identical results with the same performance as querying the graph with OQL. They additionally confirm the advantage of our object-oriented approach, which guarantees to process the RQL query with the same performance that SQL can only achieve with optimization and additional effort.

Example 7 (Processing a RQL data query) *This example refers to the data of Example 2. We present how to to process the given RQL data query with either SQL or OQL. As can be seen, SQL has the same performance as OQL only if the optimal way to process the triple filters is used.*

RQL: *Of which type are the creators of www.valerie.de?*

SQL: *The following triple-filters have to be processed:*
TF1 (www.valerie.de hasCreator X)
TF2 (X type ?)
(? is the queried result set.)

We assume that the triple table contains N triples, with N1 triple relevant for TF1 and N2 triples relevant for TF2. N? is the number of triples of the result set and N > N1, N2, N?.

Optimal join order and indexing: *The optimal join order of the triple-filters is first to process TF1 and then to use the set X for processing TF2. By using an index, we directly find all N1 triples relevant for TF1 and, with the knowledge of set X, we also find all N? triples of the result set for TF2 directly. In total, we have to consider N1 + N? objects.*

No join order and no indexing: *Not using the optimal order means to process triple-filter TF2 without knowing the set X. This demands an additional join of filters TF1 and TF2 to evaluate set X. Without using an index, we have to consider N triples for triple-filter TF1 and TF2 and N2 triples for their join, which gives 2N + N2 in total.*

OQL: *The RQL query is processed on the graph by following the outgoing hasCreator-edge of object www.valerie.de to all objects (which are N1 objects) and from there all outgoing type-edges to all objects of the result set (which are N? objects). Thereby we only consider objects (in total N1 + N?) that are relevant for the RQL query and are therefore part of the result set.*

Example 8 (Processing a RQL schema query) *This example refers to the data of Example 2. We present how to process the given RQL schema query either SQL or OQL. As can be seen, querying triples demands additional query or storage effort to achieve the same result as querying the graph.*

RQL: *Of which type is www.valerie.de?*

SQL: *The following triple-filters have to be processed:*
TF1 (www.valerie.de type X1)
TF2 (X1 subClassOf X2)
TF3 (X2 subClassOf X3)
TF4 (.....)
(X1 + X2 + X3 .... is the queried result set.)

To calculate the transitive closure induced by this RQL query, we have to apply additional subClassOf - filters as long as relevant triples can be found. The last filter will have no result and is unnecessary effort. Some relational RDF systems like Sesame [4] do not calculate the transitive closure but store all implicit triples. Thereby only filter TF1 gives the queried result set.

OQL: *The transitive closure can be calculated on the graph by following all outgoing subClassOf-edges as long as any exists. Thereby we only consider objects that are relevant for the RQL query and therefore are part of the result set.*

1 Sesame is a framework that supports different RDF storages. The RDF data access is realized by using an API layer for translating the RQL queries into storage specific queries. Using this layer instead of a direct translation has an additional impact on the system performance. We consider the original Sesame implementation using a relational database.

Nr	Query	Description
QD1	select X, Z from Z http://www.w3.org/2000/01/rdf-schema#subClassOf X. http://www.w3.org/1999/02/22-rdf-syntax-ns#type Y where Y = http://www.w3.org/2000/01/rdf-schema#Class	Returns all possible instances *X* and their sub-classes *Z*, if *X* is of *#type #class*.
QD2	http://www.w3.org/2000/01/rdf-schema#Resource	Returns all complete instances of class *#Resource*.
QD3	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	Returns all instances of property *#type*.

Table 1. Example queries regarding the data

6. Experiences

In this section, we present our experiences with the implementation introduced in the previous section. We first describe the technical environment and our test settings. Then, we present the performance data obtained in our tests. Finally, the test results are discussed.

6.1. Test environment

We compared our prototypical implementation OO-Store with the one from Sesame. The tests have been carried out on a Windows 2000 machine with 500 MB RAM and a Intel Pentium CPU 1.60 GHz. Both implementations have been tested in the context of the Sesame Framework (version 0.3 with Tomcat server 4.0), which has been written in Java. Within this framework, for our project we exchanged the query processing component. We refer to the original implementation as *Sesame* with a MySql-database (version 3.23.49) and to our implementation as *OO-Store*. We used test data taken from the Open Vine Catalog [12].

In this paper, we present the test results regarding ten selected RQL queries as listed in Table 1 to 3. The RQL queries have been directly tested in Sesame and using a OQL translation in our implementation OO-Store. The performance has been measured as response time between submission of a query until all results are returned, including connection time to the database.

We evaluated two aspects: first, the performance of queries that belong to different query categories: queries on data, on the schema, and hybrid queries. Second, the influence of a growing data base on the query performance.

6.2. Performance of different query categories

At first we show the test results regarding different categories of RQL-queries: queries on the data, on the schema and on both of them together (hybrid queries). We expect that different categories will show different performance behavior because of the differences of interpretation effort. For example, processing a schema and data related query takes more time than processing a data related query because the interpretation of the semantics is more complex.

Our hypothesis is that in general our implementation should perform better because of the advantages described in Section 5. We also expect that the impact of the advantages differs depending on the character of the query (see above).

The following three Figures 7(a) to 7(c) show the performance results observed in each category of queries(data, schema, hybrid). In each figure, we compare the performance of both implementations based on the same data base of 5000 triples. Each figure shows the response time in milliseconds(msec) on the y-axis for three different queries (on the x-axis).

During the tests, a cache effect could be observed for both implementations, which will not be discussed in detail here. Because of caching, the response times decrease a lot after the first execution. Therefore, all response times shown here are mean values of measurements taken after the second execution. We observed the following test results:

1. In general, the times for quering in OO-Store are lower than Sesame's, independent of the size of the data base and the character of the query.

2. The different categories of the queries have an impact on the performance for both implementations. Within each category (schema, data or hybrid), the query response times show a similar time behavior for each of the implementations. For example, the response times for data queries on our OO-Store are all in the order of 100 msec, while the response time of the same queries on Sesame are in the order of 1000 msec. We see that on average schema and hybrid queries (see

Nr	Query	Description
QS1	select $super, $sub from Class $super, Class $sub where $sub in subClassOf($super)	Returns all classes and their sub-classes.
QS2	select $super, $sub from Class $super, Class $sub where $sub in subClassOf($super)	Returns all classes and their direct sub-classes.
QS3	select @P, @Q from Property @P, Property @Q where @P in subPropertyOf(@Q)	Returns all properties and their direct sub-properties.

Table 2. Example queries regarding the schema

Nr	Query	Description
QH1	select * from X @P . @Q Y	Returns for all values of all properties of all possible instances again all properties and their respective values.
QH2	select * from X : $X @Q Y : $Y	Returns for all possible properties all *range* - and *domain* - classes and their respective instances.
QH3	select * from @P X : $X.@Q Y : $Y	Returns for *range* - classes and their instances of all possible properties again all properties and *range* - classes and their respective instances.

Table 3. Example queries regarding the data and schema(hybrid)

Figures 7(b) and 7(c)) need more time than data related queries (Figure 7(a)).

3. Comparing the three categories (i.e., the data in the three Figures), the performance curves for OO-Store and Sesame do not show the same behavior. While for OO-Store the performance of schema queries and data queries is almost similar, in Sesame we see strong performance differences. The reasons for this behavior lies in the different strategies for query processing in the two implementations. Sesame splits the queries into triple filters to query the tables of triples. So more complex queries (i.e schema and data related) demand the usage of more triple filters. By using an oodb, every query interprets the graph no matter whether it is a complex query or not. More complex queries only have to follow the paths of the graph more deeply. That means that the category of the query does not influence the times for oodb as much as for Sesame.

6.3. Influence of data base size

Here, we discuss the test results of a single RQL-query on a growing database. We distinguish two cases:

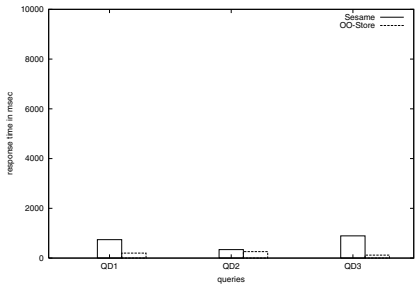
1. Increasing amount of data (in the database) with similarly increasing amount of query relevant data (i.e., the result set is also increased)
2. Increasing amount of data (in the database) with constant amount of relevant data (i.e., the result set is not changed)

Our hypothesis is that when the amount of relevant data grows with the database, the response times become higher because more data has to be evaluated. We assume, based on the previous tests, that our implementation performs better than Sesame.

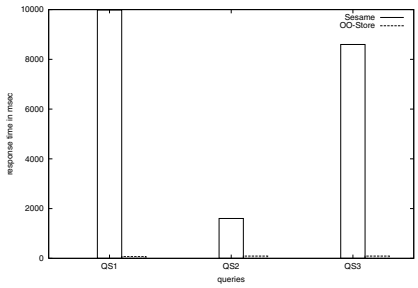
The following two figures present the results observed for a growing database with a growing result set (Figure 8(b)) and for a growing database with the same result set (Figure 8(a)) for both implementations.

Both figures show the response time in msec on the y-axis, and the amount of triples stored in the database in thousands on the x-axis. We observed the following test results:

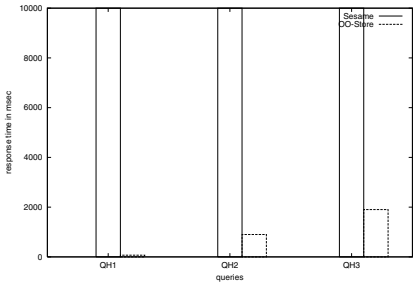
1. The response times for both implementations depend on the size of the data base and the size of the result set. A growing data base implies increasing response times if the amount of relevant data grows similarly. In



(a) Response time for data queries



(b) Response time for schema queries

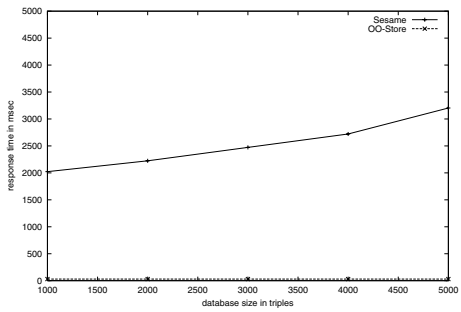


(c) Response time for hybrid queries

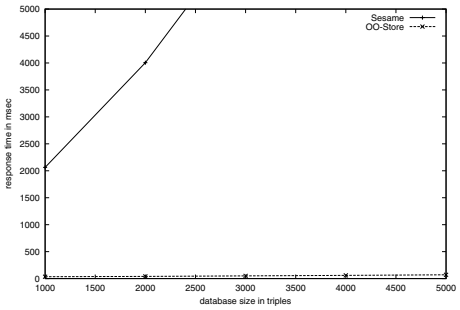
Figure 7. Performance for the different categories of RDF queries on 5000 triples

that case, for both implementations the time for query evaluation grows because more relevant data has to be processed.

2. A growing data base with constant amount of relevant data causes a different behavior for both implementations. The response time for Sesame increases because independent of the relevant data more triples in the tables exist. Sesame's triple filter have to extract the relevant data from a larger amount of data. The response time for our implementation OO-Store remains constant because queries are evaluated by starting at a certain point in the graph and by following the relevant paths. The paths to process are of the same length if the amount of relevant data remains constant.



(a) Response times on a growing database with a constant resultset



(b) Response times on a growing database with a growing resultset

Figure 8. Performance of a single query on a growing database

However, for all evaluated cases, our OO-Store implementation shows better performance than Sesame. Our experiments confirm the assumption about the advantages of querying the stored graph.

7. Conclusion and outlook

In this paper we presented a new approach to store RDF data as a graph.

First we discussed the RDF data model that defines triples as the base structure of the RDF language. Based on this data model, RDF schema gives a certain semantics for the data. Then, we have shown that these semantics can be only interpreted by using the graph representation of RDF data. Therefore, a powerful query language has to be based on the graph model. Currently, only the RQL query language fulfills this requirement.

In order to support RQL, a RDF system has to map the graph model onto its storage structure. Existing systems store the triples in tables of a relational database system. Consequently, for the evaluation of a schema-related query, i.e., a query requiring a semantic interpretation of RDF data, the graph has to be built from the triples. In our approach, this additional step is not necessary. We proposed to directly store the graph representation of the RDF data in an object-oriented database. This approach simplifies the storage concept, enables us to directly and efficiently query the stored graph, and take advantage of the close relation of RDF and OQL.

As proof of concept we presented our reference implementation of OO-Store, an RDF storage system based on the object oriented database system Fast Objects. We have compared our system with Sesame, a well-known RDF system. We have shown that for all evaluated cases, our OO-Store implementation has better performance than Sesame. Our experiments confirm the assumption about the advantages of querying the stored graph.

Currently, we are implementing a full RQL parser, which automatically translates RQL queries into OQL. We are also investigating the incorporation of RDF into user profiling systems and the usage of RDF for content-based routing.

Acknowledgements. We wish to thank the database group at the FU Berlin and Jeen Broekstra of the Sesame group for their valuable comments on our approach.

References

[1] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The RDFSuite: Managing Voluminous RDF Description Bases. In *Proceedings of the 2nd International Workshop on the Semantic Web (SemWeb'01), in conjunction with Tenth International World Wide Web Conference (WWW10)*, Hongkong, May 2001.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, March 2001.

[3] D. Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C working draft, April 2002. available at `www.w3.org/TR/rdf-schema/`.

[4] J. Broekstra, A. Kampman, and F. van Harmelen. *Sesame: An Architecture for Storing and Querying RDF Data and Schema Information*. MIT Press, 2001. available at citeseer.nj.nec.com/broekstra01sesame.html.

[5] V. Bures. Effiziente Speicherung von RDF. Master's thesis, Institute for Computer Science, Freie Universitt Berlin, Germany, October 2002. available at www.valerie-bures.de.

[6] Hewlett-Packard Company. RDQL - RDF Data Query Language . available at `http://www.hpl.hp.com/semweb/rdql.htm`, last visited April 2003.

[7] P. Fankhauser, G. Huck, and I. Macherius. Components for data intensive XML applications. *ERCIM News*, (41), April 2000.

[8] Homepage of FastObjects Inc. available at `www.fastobjects.de`, last visited March 2003.

[9] G. Karvounarakis, V. Christophides, and D. Plexousakis. Querying Semistructured (Meta)Data and Schemas on the Web: The case of RDF and RDFS. Technical Report 269, FORTH-ICS, February 2002.

[10] O. Lassila and R.R. Swick. Resource Description Framework(RDF) Model and Syntax Specification. W3C working draft, February 1999. available at `www.w3.org/TR/REC-rdf-syntax/`.

[11] Libby Miller. Inkling: RDF query using SquishQL. RFC draft, 2002. available at `ilrt.org/discovery/2001/02/squish/`.

[12] Homepage of the Open Vine Project. available at `www.openwine.org/data.jsp`, last visited July 2002.

[13] G.K. Sofia. RQL: A Declarative Query Language for RDF. In *Proceedings of the WWW2002*, Honolulu, Hawaii, USA, May 2002.

[14] Tuan Anh TA. RDF Query: current status and future , 2001. available at `www.infres.enst.fr/~ta/web/rdf/rdf-query.html`, last visited March 2003.