DLDB: Extending Relational Databases to Support Semantic Web Queries

Zhengxiang Pan **Jeff Heflin**

Department of Compute Science, Lehigh University
19 Memorial Drive West,
Bethlehem, PA 18015, USA
{zhp2, Heflin}@cse.lehigh.edu

Technical Report: LU-CSE-04-006

Abstract: We present DLDB, a knowledge base system that extends a relational database management system with additional capabilities for DAML+OIL inference. We discuss a number of database schemas that can be used to store RDF data and discuss the tradeoffs of each. Then we describe how we extend our design to support DAML+OIL entailments. The most significant aspect of our approach is the use of a description logic reasoner to precompute the subsumption hierarchy. We describe a lightweight implementation that makes use of a common RDBMS (MS Access) and the FaCT description logic reasoner. Surprisingly, this simple approach provides good results for extensional queries over a large set of DAML+OIL data that commits to a representative ontology of moderate complexity. As such, we expect such systems to be adequate for personal or small-business usage.

Keywords: DAML+OIL, Knowledge Base, Relational Database, Description Logic Reasoner, Storing RDF

1 Introduction

DAML+OIL [Connoly et al., 01] enables the creation of ontologies and provides extensive semantics for Web data. This language is heavily influenced by description logics. Research on DL reasoners is primarily focused on intensional queries, that is, queries about the structure of an ontology. However, it is almost certain that the majority of Semantic Web queries will be extensional ones. Databases are excellent tools for storing and querying data, but lack the ability to perform the inference sanctioned by DAML+OIL entailments. This paper describes one method to extend relational databases to support DAML+OIL semantics.

We design DLDB, a lightweight knowledge base system that has the capability of DAML+OIL inference and supports semantic web queries. Making use of the FaCT DL reasoner [Horrocks, 00], DLDB has been successfully implemented on a common RDBMS: Microsoft Access. It can process, store and query DAML+OIL formatted semantic content. The main purpose of this system is to investigate how DL reasoning and relational database systems can be combined to support extensional queries about DAML+OIL documents. By extensional, we mean queries that concern ground data, as opposed to queries about the structure of the ontologies. This system is optimized

for ontologies of moderate sizes (at the magnitude of hundreds of classes and properties).

In this paper we describe some interesting aspects of DLDB systems. Section 2 discuses the alternatives and final design of DLDB, and summarizes the features of DAML+OIL and RDF [W3C, 99] supported by the system. In section 3 we elaborate the implementation issues and query API. Section 4 provides an overview and discussion of related work. We conclude in section 5.

2 Design

Since DAML+OIL builds on RDF, we will first look at how to represent RDF information in a database. Then, we describe how to add support for RDF and DAML+OIL inference to our design.

2.1 RDF(S) Storage in Relational Databases

The RDF data model can be viewed as either a graph or a set of triples. In the triple model, each triple consists of a property, subject and object. RDF Schema (RDFS) defines a number of classes and properties that have specific semantics. A class is a set of resources, and corresponds to the notions of type or category in other representations. The *rdf:type* property is used to relate a resource to the class of which it is a member, and all classes are simply resources of the type *rdfs:Class*. Properties are also resources, and are members of the *rdf:Property* class. Additional semantics for properties can be given using the *rdfs:domain* and *rdfs:range* properties, which constrain the domain and range of a property.

When storing semantic web information in a database, the first problem is to decide on the appropriate table design. Although there are differences between RDF's graph-based model and the semi-structured model of XML, our work can benefit from the research on storing XML data in relational databases [Florescu and Kossman, 99]. However, one must be careful when applying the results of this research, because the RDF model theory places additional requirements on queries, especially those that depend on the semantics of *rdfs:subClassOf*. Here we discuss a number of alternatives and evaluate them in terms of applicability to RDF.

Horizontal DB

One potential approach is to use horizontal schema [Agrawal et al., 01]. Only one "universal" table is needed in the database. Figure 1 demonstrates the structure of this table. An individual's ID, source class's name and all the properties' values are fields in that table. Hence, every individual (instance) falls into one entry in the table. While the data model is simple, there are some drawbacks within this approach:

1. Large number of columns. Considering the scale of the knowledge base and the number of properties, the large number of columns might exceed the limitations of the underlying relational database system.

2. Limits on property values. For one individual, the table structure limits it to have only one value for each property. However, many properties are naturally multi-valued.

3. Sparsity. Obviously, every property occupies a field in the table no matter how many records have values to fill in this particular field. Hence, there could be many null fields in the database.

4. Maintenance. Every time the system incorporates a new ontology or changes an existing ontology, the table needs to be restructured. Such changes can be very expensive if the table is very large.

5. Performance. It is known that this approach results in a significantly larger database size and the loading time is much longer than in other approaches [Florescu and Kossman, 99].

A variation for this approach is to set up such a table for each ontology. Clearly this will reduce number of columns in any table and make changes to the table less likely. However, the limits on property values still remain; the tables will have significantly more columns than in typical databases, and will still be very sparse.

Instance's ID	Type	Property_1	Property_2	...	Property_n
….#1	*Class_A*	*Value_a*	*Value_b*	...	*Value_c*
….#2	*Class_B*	*Value_d*	*Value_e*	...	*Value_f*

Figure 1: Horizontal DB representation

Vertical table

Another alternative is the vertical table approach, which is employed by Jena and TAP [Beckett and Grant, 01]. In [Alexaki et al., 01], it is also named the "Generic Representation". Like the horizontal approach, there is a single table, but it is "vertical" instead of "horizontal", as shown in Figure 2. Three fields, "Predicate", "Subject" and "Object" are in that table. There are several similar approaches that also create tables based upon RDF statements and resources, such as [Berners-Lee, 98] [Melnik, 01] [Gertz and Sattler, 01]. This approach has the simplest database, and allows files to be easily parsed and loaded into the database. Each RDF triple can be immediately stored as it is parsed, allowing for a streaming implementation. Another advantage is, unlike the horizontal one, the table never has to be restructured. However, this design means that any query has to search the whole database and queries that involve joins will be especially expensive. In particular, queries about the members of a class will be particularly difficult, because there is no explicit treatment of the class hierarchy. As a consequence, a series of queries must be issued simply to find all of the subclasses of a class.

Predicate	Subject	Object
Type	*….#1*	*Class_A*
Property_1	*….#1*	*Value_a*
Property_2	*….#1*	*Value_b*
...	...	...

Figure 2: Vertical table representation

Horizontal class

This approach is similar to the horizontal database approach but has a much smaller granularity. There is a separate table for each class in the ontology. Each table is named for its class and like the horizontal database approach, has a number of columns for properties (see Figure 3). However, unlike the horizontal database approach, every table contains fields only for those properties whose *rdfs:domain* cover it; hence the tables are less sparse. This essentially corresponds to the entity-relational approach frequently used when designing databases. The main advantage of this approach is its ability to efficiently handle queries about the properties of an individual or a set of individuals. Since every record covers an individual and all its related property values, only a few retrieval steps are executed to fulfill many queries. Like the horizontal DB approach, the individuals may not have values for some properties, although it should be less sparse. However, some properties may not have explicit domains, implying that they may be applicable for any type of object, and thus must be included in every table.

Table name: Class_A

ID	Property_1	Property_2	...	Property_n
...#1	*Value_a*	*Value_b*	...	*Value_c*

Figure 3: Horizontal class representation

Table per property

Yet another alternative is to assign a table to each property:

PROPERTY_name(Subject,Object)

In the database community, this approach is called the "decomposition storage model" [Agrawal et al., 01]. It is also being used in the PARKA knowledge representation system [Stoffel et al., 97]. Particularly, the instances of each class is recorded in the table for the *rdf:type* property(see Figure 4). In this alternative, the number of tables in the database increases to some extent. However, the size of each table decreases. The advantage is a faster response time for simple queries, because they can search a few small tables as opposed to large ones. However, for complex queries that consider many properties, there may need to be many join operations. Like the vertical table approach, queries involving the implicit instances of a class can be particularly expensive, unless, like Parka, explicit reasoning is built in to the database itself to deal with this issue.

Table name: Type

Subject	Object
...#1	*Class_A*
...#1	*Class_B*

Table name: Property_1

Subject	Object
...#1	*Value_a*
...#2	*Value_d*

Figure 4: Table per property representation

Hybrid approach
We adopted an approach that combines the property table approach with the horizontal class aproach (see Figure 5). It is similar to the "Specific Representation" in [Alexaki et al., 01]. According to this model, creating tables corresponds to the definition of classes or properties in ontology. The classes or properties' ID should serve as the table names.

As in the horizontal class approach, each class has a table that records information about its instances. However, here we only record the instances of the class as indicated by the *rdf:type* relation. Since each row in the class tables corresponds to an instance of the class, we only need an 'ID' field here to record the ID of the individuals who are instances of this particular class.

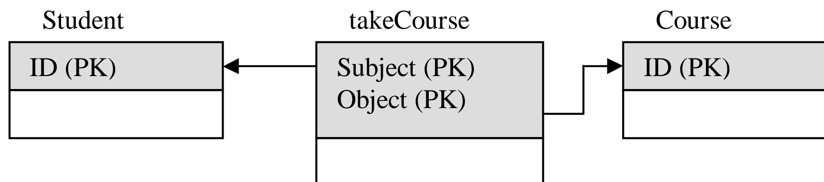

Figure 5: Example data model for a small ontology using Hybrid design

The rest of the data about an instance is recorded using the table per property (or decompositional) approach. Each instance of a property must have a subject and an object, which together identify the instance itself. Thus the 'subject' and 'object' fields are set in each table for property. Normally, the 'subject' and 'object' fields are foreign keys from the 'ID' field of the class tables that are the domain or range of the property, respectively. However, if the property's range is a declared data type, then the object may be a datatype.

For ontologies with a moderate number of classes (a few hundred at most), this approach should perform well for simple queries. The experiments of [Agrawal et al., 01] showed that the table per property approach performed as well as or better than the horizontal and vertical approaches. However, underlying database systems may have a maximum number of tables, and ontologies with too many classes (e.g. the openDirectory ontology) either can't be accommodated or have a significant overhead [Alexaki et al., 01]. Yet, such schemas are rare, as discovered by the survey in [Magkanaraki et al., 02]. Therefore, we chose this approach, because it should work with the most common forms of schemas.

2.2 RDF(S) Entailment in Relational Databases

Once a table design is chosen, we need to implement RDF entailment. RDF entailment is relatively simple, mostly consisting of taxonomic inference using *rdfs:subClassOf* and *rdfs:type*, and similar inference for *rdfs:subPropertyOf*.

In our system class hierarchy information is stored through views. A view is a form of query in relational database; it creates the logical combination of tables and fields. From the user's perspective, it looks like a "virtual" table. In our design, the view of a class is defined recursively. It is the union of its table and all of its direct subclasses' views. Hence, a class's view contains the instances that are explicitly

typed, as well as those that can be inferred. The algorithm used to create class views is shown in Figure 6.

In the terminology of deductive databases, the views are intensional database (IDB) relations which are defined by logical rules. The tables are extensional database (EDB) relations which store the explicit information from the documents.

For example, consider the following statements in a RDF model:

```
<rdfs:Class rdf:ID="Student"/>
<rdfs:Class rdf:ID="UndergraduateStudent">
    <rdfs:subClassOf rdf:resource="#Student" />
<rdfs:Class/>
<Student rdf:ID="Susan"/>
<UndergraduateStudent rdf:ID="Jack"/>
```

When we load this model into the database, the Class view creation algorithm will define the view of Student as:

```
SELECT * FROM Student
UNION SELECT * FROM UndergraduateStudent_view;
```

If we query for all the instances of Student, this view ensures that Jack as well as Susan will be included in the result set.

function CreateViews(R)
    inputs: R, a RDF model
    static: $T_1, T_2, \ldots T_n$, a set of database tables
        $V_1, V_2, \ldots, V_n$, a set of database views

    for all triples (type x Class) $\in R$ do
        let T_x be the table containing explicit instances of class x
        let V_x be the view corresponding to class x
        $V_x \leftarrow T_x$
    for all triples (subClassOf y x) $\in R$ do
        let V_x be the view corresponding to class x
        if $\neg(\exists x'$, ((subClassOf y x')$\in$ R) $\wedge$ ((subClassOf x' x)$\in R$)) then
            let V_y be the view corresponding to class y
            $V_x \leftarrow V_x \cup V_y$

Figure 6: RDF Class view creation algorithm

The *rdfs:subPropertyOf* relationship between properties is implemented in a similar way. The only remaining RDF entailments are those that entail class membership based on the use of a property and its domain or range. This could be accommodated in the view approach by adding another view to the union, however we do not currently implement this.

2.3 Supporting DAML+OIL Entailment

DAML+OIL has many features from description logics, the most significant are the constraints for class description. Using these, a DL reasoner can compute class subsumption, i.e., the implicit subClassOf relations. Our database design can benefit from subsumption by using a DL reasoner [Horrocks et al., 99] to precompute subsumption, and then using the inferred class taxonomy to create our class views. This process can be described as:

```
function createDAMLViews(R)
   input: R, a DAML+OIL model
   variable: R', a new model

   R' := R
   for all R ⊨DAML+OIL (subClassof x y) do
      R' := R' ∪ {(subClassOf x y)}
   createViews(R')
```

Figure 7. DAML+OIL Class view creation algorithm

Here $\models_{DAML+OIL}$ is the DAML+OIL logical consequence (entailment) relation. Although space limitations prevent us from defining DAML+OIL entailment here, a short example can illustrate its benefits. Consider that the class Student is defined as all the people who take a Course, the class GradStudent is defined as all the people who take a GradCourse, and the class GradCourse is a subclass of Course. Then, by DAML+OIL entailment we can conclude that GradStudent is a subclass of Student.

Using the above mechanism, our system stores the results of subsumption and only consults the DL reasoner once for each new ontology in the knowledge base. Whenever queries are issued concerning the instances of the ontology, the inferred hierarchy information can be automatically utilized. The intuition here is that ontologies don't change frequently although they can be imported or referred to many times. Thus, precomputation improves the computational efficiency which can save time as well as system resources. Note that at this time, we have not considered those elements (features) that are not related to subsumption. They are: *daml:inverseOf*, *daml:equivalentTo*, *daml:hasValue*, *daml:sameIndividualAs*, *daml:UnambigousPropery* and *daml:UniqueProperty*, etc. In the next section, we describe how this design has been implemented.

3 Implementation

In this section, we present the implementations for above design in three aspects: the database schemas on relational database, the inference mechanism and the query functionality.

3.1 Implementation of Database Schemas

In our implementation, we use Microsoft Access as our relational database management system. The reason we chose Access lies in its popularity on PCs, thus most people can set up and use our system easily.

In addition to our table design, some details should be taken into account when implementing the database schemas for the system. First, different ontologies may use the same IDs for different classes or properties. In order to store multiple ontologies in a single database, we have to distinguish them as they serve as table names. This involves the name convention in the database. Intuitively, a class or property's ID plus its ontology's ID will be a good choice for a unique identifier. However, an ontology's ID is a URI, which often exceeds the table name's length limitation. So we give a local unique sequence number to each loaded ontology. Then each table's name is a class or propert ID plus its ontology's sequence number. This is supported by an extra table

ONTOLOGIES(Url, Sequence_Number)

that is used to register and manage all the ontologies in the knowledge base. The sequence number will be assigned by the system when an ontology is first loaded into the database.

Sometimes it is important to know which document a particular statement came from, or how many documents contain a particular statement. We support this capability by including a 'source' field in each class and property table. Together with other fields, it serves as the multiple-field primary key of the table. In other words, the combination of all the information of one record identifies each instance stored in the knowledge base. An example of class and property tables might be:

STUDENT(ID, Source)

TAKECOURSE(Subject, Object, Source)

For many applications, it is important that the class taxonomy can be displayed. There is no reason to recompute the subsumption every time this information is needed. Therefore, an additional table is used for each ontology to store hierarchy information:

HIERARCHY (ClassName, Super)

This table is similar to a property table for "subClassOf" property, as demonstrated in the property table approach in section 2. It has two fields, class name and its direct super class's name. Although we build class hierarchy on views, the hierarchy is implied by the design of the view. This additional table explicitly specifies the class hierarchy and makes it easier to retrieve the class hierarchy for applications such as a query panel.

In order to shrink the size of database and hence reduce the average query time, we assign each URI a unique ID number in our system. We use a table:

URI-INDEX(URI, ID)

to record the URI-ID pairs. Thus, for a particular resource, its URI is only stored once, its corresponding ID number will be supplied to any other tables. By

discriminating the *DataTyp*e properties and *ObjectType* properties, the literals are kept in their original form without being substituted by ID numbers. Since the typical URL can be long as 10-30 characters and the size of number is only 4 bytes in our database, this results in a significant savings in disk space utilized. Furthermore, due to the reduced table size, query performance is also increased. Unsurprisingly, the tradeoff of doing the URI-ID translation is to increase the load time. The majority of the extra load time occurs when looking up the URI-index table to find the ID for a URI. In order to reduce this overhead, we create indices for both columns in the URI-index table. These indices increase the database's size slightly, but significantly shorten the ID-lookup query time especially when the URI-index table is very long. In addition, we use a hash table to cache every URI-ID pair found during the current loading process. Since URIs are likely to repeat in one document, this cache save a lot of time by avoiding lookup queries when possible.

3.2 FaCT and Inferences

In our implementation, we borrow some code from OilEd [Bechhofer et al., 01] and use FaCT [Horrocks, 00] as our reasoner. Our system uses the FaCT system to perform DL reasoning and precompute subsumption.

> "*FaCT(Fast Classification Terminologies) is a Description Logic (DL) classifier that can also be used for modal logic satisfiability testing. The FaCT system includes two reasoners, one for the logic SHF and the other for the logic SHIQ, both of which use optimized implementation of sound and complete tableaux algorithms. FaCT's most interesting features are its expressive logic (in particular the SHIQ reasoner), its optimized tableaux implementation, and its CORBA based client-server architecture*" [Horrocks, 00].

Although FaCT is implemented in Lisp, its CORBA interface [Bechhofer et al., 99b] enables Java-based programs to interact with it. Its SHIQ reasoner is used to reason on ontologies. In our design, the reasoning only occurs when loading an ontology. First, a DAML parser borrowed from OilEd parses the original ontology source file to an ontology object, and then by executing the *SubmitToFaCTCommand* from OilEd, this object is translated into an equivalent SHIQ knowledge base and serialized to a temporary XML-formatted file [Bechhofer et al., 99a]. Note that the DTD for this XML file is specific to FaCT. The reasoner running on the FaCT server reads that XML file to construct concepts, checks for the consistency of classes, determines the implicit subsumption relationships and reports what has been discovered by rewriting that temporary XML file. After the reasoner terminates, the program executes *CommitFaCTChangesCommand* from OilEd to rearrange the class hierarchy in the ontology object based on the temporary XML file.

As a result, the ontology object contains consistent classes with all of the subsumption relationships explicitly expressed. DLDB then creates tables and views for corresponding classes and properties using a variation of the algorithm from Section 2.2. At this time, each view just contains the corresponding table for its class. When all tables and views are ready, the system adjusts each view according to the hierarchy information in the ontology object. This adjustment starts from an arbitrary class. If a class has direct subclasses, its original view is dropped and a new view which is the union of its table and all of its direct subclasses' view is created. If a class doesn't

have any subclasses, its view is kept. By repeating this for all classes and properties, the ontology is entered into the knowledge base properly and instances can be eventually stored into those tables.

3.3 Query API

The query API receives queries from users, executes query operations and returns query results to them. This API currently supports conjunctive queries and is implemented as a set of Java classes. Query is a Java class that contains one or more Atom classes, each of which is a conjunct. The structure of an atom object is based on atoms in First Order Logic; it consists of a predicate name and a series of terms. Before executing a query, a query object must be created first. During execution, predicates and variables in the query are substituted by table names and field names through translation. Finally, it forms a standard SQL query sentence and sends it to the database via JDBC. Then the database's DBMS processes the SQL query and returns appropriate results.

Let's take a sample query to illustrate how the process works. The user's original query is "find any graduate student who takes course0 at department0 at foo University." At this stage, the atoms in the query object issued to system are in KIF-like format:

```
(Type GraduateStudent ?X)
(TakeCourse ?X http://www.foo.edu/department0/course0)
```

where questions marks indicate variables.

First the predicates and variables are translated into table names, field (column) names and conditions through the algorithm in Figure 8.

Then, that query is translated into SQL sentences like:

```
SELECT
GraduateStudent_2_view.ID, takeCourse_2_view.object
FROM
GraduateStudent_2_view, takeCourse_2_view
WHERE
GraduateStudent_2_view.id = takeCourse_2_view.subject
AND
takeCourse_2_view.object=
http://www.foo.edu/department0/course0'
```

Since we built the class and property hierarchy when loading the ontology, there is no need to call the DL reasoner at query time. The results are typically the instances (directly or through subsumption) of the specified class with related properties' values.

```
Function translateQuery(query)
   Input: query, a kif-like format query
   Variable: table, name of a database table
   Variable: column, name of a column of a table in database
   Hashtable: Vars
   Variable: SELECT list, a select clause in SQL syntax
             FROM list, a from clause in SQL syntax
             WHERE list, a where clause in SQL syntax
   function getTableName(class or property name) returns the corresponding table
name in  the database

   for every atom in the query, from the first one to the last one do
      arg1 -> the first argument of the atom
      arg2 -> the second argument of the atom
      pred -> predicate of the atom
      If predicate is "Type" then
         table = getTableName(arg1) ;
          add table to FROM list;
         add table's ID column to SELECT list;
          put "table.ID" in Vars, arg2 as key;
      else
          if FROM list doesn't contain pred then
             table = getTableName(pred);
             add table to FROM list;
          if find a record in Vars whose key is arg1 then
             add table's object column to SELECT list;
             column = get the record in Vars whose key is arg1;
             add "column '=' table's subject" to WHERE list
          else
             put "table.subject" in Vars, arg1 as key;
          if arg2 is a variable and find a record in Vars whose key is arg2 then
             Add table's subject column to SELECT list;
             column = get the record in Vars whose key is arg2;
             add "column '=' table's object" to WHERE list;
          if arg2 is a variable and find no record in Vars whose key is arg2 then
             put "table.object" in Vars, arg2 as key;
          if arg2 is a literal then
             add "table's object '=' arg2" to WHERE list;

return FROM list, SELECT list and WHERE list
```

Figure 8: The query translation algorithm

4 Related Work

Many researchers have investigated the use of relational database systems to process semi-structured documents [Agrawal et al., 01] [Florescu and Kossman, 99]. Their results are extensively utilized by people when design RDF repositories. As we discussed in section 2, we should be aware that RDF model requires to accommodating semantics besides instance data.

There are a number of repositories for RDF, but few have explicit support for DAML+OIL. DAML DB [Dean and Neves, 01] stores statements and other fixed-length information in memory-mapped files instead of physical databases. It also maintains linked-lists of statements involving the same subject, predicate, and object. However, it does not directly support any DAML+OIL entailments.

Sesame [Broekstra et al., 02] is an RDF Schema-based repository and querying facility. Using the Repository Abstraction Layer, Sesame can be implemented on top of a variety of storage devices, including relational databases, triples stores and object-oriented databases etc. Sesame currently supports PostgreSQL (7.0.2 or later), MySQL (3.23.47 or later) and Oracle9i. On PostgreSQL, Sesame uses a table design similar to ours, except that, since PostgreSQL is an object-relational DBMS, it employs subtable relations between tables instead of views to construct subsumptions. Sesame has also implemented the query engine for RQL (RDF Query Language).

The ICS-FORTH RDFSuite [Alexaki et al., 02] consists of the Validating RDF Parser (VRP), the RDF Schema-Specific Data Base (RSSDB) and the RQL Interpreter. Its database schema is very similar to Sesame and also based on an ORDBMS model.

Some other projects ([Guha, 01] [Beckett, 03]) also implement RDF repository on relational database. Yet all the above systems lack inference mechanisms, and thus do not take full advantage of DAML+OIL's semantics.

KAON Server [Volz et al., 03] is a Semantic Web Management System. It has a RDF Server to persist RDF models in a relational DBMS. KAON includes many registered components to support various application scenarios, including FaCT as a DL reasoner, a rule-based system and an ontology editor. The KAON API acts as a kernel and provides managements and communications for these components.

BOR [Jordanov and Simov, 02] is another DL reasoner. SeBOR is the integration of BOR and Sesame. SeBOR also uses a knowledge compilation technique, but since there are few details on the actual implementation, we could not compare our approach to theirs.

5 Conclusion

We described the design and implementation of DLDB. Preliminary experiments show that the use of views in a relational database and the FaCT reasoner make the results much more complete for some queries, while the costs (such as the increases on query time, loading time and database size) are considerably low or even negligible. Table 1 shows how the performance of DLDB scales with the number of instances in the system. For instance, a DLDB system that contains information about 5 typical universities (17,150 instances) occupies about 100M bytes disk space. A highly selective query regarding 1 class and 1 property can be completed in 200 milliseconds or so. For the full experiment description and more details, please refer to our paper on benchmark tests [Guo et al., 03]. All these prove that the idea of using description logic to extend the relational database is feasible.

We choose Microsoft Access as our underlying database because it is a low-cost system that is already available to many people. However, MS Access is not particularly scaleable, and such a system would not be suitable for a large-scale knowledge portal. Instead, we see this system as fitting the needs of the personal or small busi-

ness user who wishes to take advantage of semantic web technology. The integration of a common desktop database system with basic description logic reasoning gives such users the best of both worlds.

However, it is important to note that our design is not dependent on Access. In fact, any RDBMS can be used. We believe that given a suitable underlying database, this approach will scale well, and this is one of our chief directions for future work. Other future directions include adding support for more RDF and DAML+OIL entailments, and experimenting with the performance of various design alternatives.

Number of instances	Load time (hr:min:sec)	Size on disk (KB)	Typical query time (ms)
17,150	0:6:51	13,042	32 - 418
107,421	0:22:39	73,925	200 - 4,962
218,690	1:27:24	147,949	396 - 11,953
462,316	3:06:32	311,099	881 - 32,948
1,146,186	7:59:46	766,738	2295 - 115,782

Table 1: Performance of DLDB

Acknowledgement

Some of the material in this paper is based upon work supported by the Air Force Research Laboratory, Contract Number F30602-00-C-0188. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.

Reference:

[Agrawal et al., 01] R. Agrawal, A. Somani, and Y. Xu. *Storage and Querying of E-Commerce Data*. In Proc. of VLDB 2001

[Alexaki et al., 01] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis & K.Tolle, *On Storing Voluminous RDF Description: The case of Web Portal Catalogs*, In Proc. of the 4th International Workshop on the Web and Databases (WebDB2001) in conjunction with ACM SIGMOD'01 Conference, 2001.

[Alexaki et al., 02] S. Alexaki, N. Athanasis, V. Christophides, G. Karvounarakis, A. Maganaraki, D. Plexousakis and K. Toll. *The ICS-FORTH RDFSuite: High-level Scalable Tools for the Semantic Web*. Poster Session of the Eleventh International World Wide Web Conference (WWW'02), Honolulu, Hawaii, USA, May 8, 2002.

[Bechhofer et al., 99a] S. Bechhofer, I. Horrocks, P. F. Patel-Schneider, and S. Tessaris. *A Proposal for a Description Logic Interface*. In Proceedings of the International Workshop on Description Logics (DL'99), pages 33-36, 1999.

[Bechhofer et al., 99b] S. Bechhofer, I. Horrocks and S. Tessaris. *CORBA interface for a DL Classifier*. March 1999.

[Bechhofer et al., 01] S. Bechhofer, I. Horrocks, C. Goble and R. Stevens. *OilEd: a Reasonable Ontology Editor for the Semantic Web*. In Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence. Springer-Verlag LNAI Vol. 2174, pp 396--408. 2001.

[Beckett and Grant, 01] D. Beckett and J. Grant, *Mapping Semantic Web Data with RDBMSes*. 2001, http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/

[Beckett, 03] D. Beckett. *Redland RDF Application Framework*. 2003. http://www.redland.opensource.ac.uk/

[Berners-Lee, 98] T. Berners-Lee. *Relational Databases on the Semantic Web*. Sep.1998, http://www.w3.org/designissues/rdb-rdf.html

[Broekstra et al., 02] J. Broekstra, A. Kampman and F. van Harmelen. *Sesame: A generic Architecture for Storing and Querying RDF and RDF Schema.* In Proceedings of the First Internation Semantic Web Conference. 2002.

[Connoly et al., 01] D. Connolly, F. van Harmelen, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider and L. A. Stein. *DAML+OIL (March 2001) Reference Description*, 2001 http://www.w3.org/TR/daml+oil-reference

[Dean and Neves, 01] M. Dean and P. Neves. *DAML DB*. Sep. 2001. http://www.daml.org/2001/09/damldb/

[Florescu and Kossman, 99] D. Florescu and D. Kossman. *A performance evaluation of alternative mapping schemes for storing XML data in a relational database*. Technical report, INRIA, France, May 1999.

[Gertz and Sattler, 01] M. Gertz and K. Sattler. *A Model and Architecture for Conceptualized Data Annotations*. Technical Report CSE-2001-11, Department of Computer Science, University of California, Davis, December 2001.

[Guha, 01] R.V.Guha. *RDFDB : An RDF Database*. 2001. http://rdfdb.sourceforge.net/

[Guo et al., 03] Y. Guo, J. Heflin, Z. Pan. *Benchmarking DAML+OIL Repositories*. In Proceedings of the Second International Semantic Web Conference (ISWC 2003). 2003.

[Horrocks, 00] I. Horrocks. *Benchmark Analysis with FaCT*. In Proc. TABLEAUX 2000, pages 62-66, 2000.

[Horrocks et al., 99] I. Horrocks, U. Sattler, and S. Tobies. *Practical Reasoning for Expressive Description Logics*. In Proc. of LPAR'99, pages 161-180, 1999. http://www.cs.man.ac.uk/~horrocks/Publications/download/misc/corba-fact-idl.ps.gz

[Jordanov and Simov, 02] S. Jordanov, K. Simov. *BOR System Documentation*. 2002. http://www.sirma.bg/OntoText/bor/BOR.htm

[Magkanaraki et al., 02] A. Magkanaraki, S. Alexaki, V. Christophides, and D. lexousakis. *Benchmarking RDF Schemas for the Semantic Web*. In Proceedings of the First International Semantic Web Conference (ISWC 2002). 2002.

[Melnik, 01] S. Melnik. *Storing RDF in a relation database*. Dec. 2001, http://www-db.stanford.edu/~melnik/rdf/db.html

[Stoffel et al., 97] K. Stoffel, M. Taylor, J. Hendler. *Efficient Management of Very Large Ontologies.* In Proceedings of American Association for Artificial Intelligence Conference (AAAI-97), AAAI/MIT Press 1997.

[Volz et al., 03] R. Volz, D. Oberle, B. Motik, S. Staab. *KAON SERVER -A Semantic Web Management System*. In Proceedings of the 12th World Wide Web, Alternate Tracks - Practice and Experience, Hungary, Budapest. 2003.

[W3C, 99] World Wide Web Consortium. Resource *Description Framework (RDF)*, 1999 http://www.w3.org/RDF/