Genea: Schema-Aware Mapping of Ontologies into Relational Databases

Tim Kraska* Uwe Röhm

University of Sydney
School of Information Technologies
Sydney, NSW 2006, Australia
mail@tim-kraska.de roehm@it.usyd.edu.au

Abstract

Ontologies have become an important mechanism for describing and exchanging data in the semantic web, as well as in application areas such as bioinformatics or health care. This paper addresses the problem of how to efficiently store and query ontology instance data using an RDBMS. Our approach automatically creates a schema-aware relational representation of the ontology using generic mapping rules. Part of the ontology reasoning, such as on subsumption relationships, is done during schema-generation and also at load-time of the instance data, so that query processing becomes faster. We implemented our approach in an OWL-mapping tool, called Genea, that can automatically create a compact relational schema and load the instance data from a given ontology written in OWL. We report on results from a quantitative and qualitative evaluation of our approach as compared to both schema-oblivious and schema-aware RDF stores from the semantic web community. Genea was found to be a generic and effective tool for ontology mapping. Because the generated schemas capture more semantics, they allow for faster querying than previous approaches.

1 Introduction

Building the semantic web requires use of ontologies, which describe the semantics of data and the knowledge in a certain application field. Examples are the bioinformatics or the health sector. One of the most popular languages for specifying and instantiating ontologies is the Web Ontology Language (OWL) that builds on top of RDF [23]. For example, the BioPAX data exchange standard for metabolic pathways [2] is a domain specific ontology that is written in OWL.

Many research focuses on intensional queries that allow to reason about the structure of an ontology and to infer new facts. In this work, we assume the ontology structure to be known and relatively fixed, such as is the case with the BioPAX standard, and concentrate on extensional queries, that is queries retrieving the instance data itself. We target application scenarios such as an integrated web database for metabolic pathways that imports data from several sources using the BioPAX standard, and that can provide fast access to the database via keyword searches, class hierarchy browsing, and pathway visualisation [9].

There are several possibilities of storing and processing ontology data. First of all, ontologies could be stored natively as files without any database. But this requires giving up, or reimplementing, all the services information systems are used to, such as buffer management, transactions, or optimisation. For ontologies written in OWL and hence using XML syntax, another approach is to use XML databases. However, this would restrict us to the syntax level and does not capture the semantics of the ontology.

Another possibility is storing in relational databases. If we want to store ontology instance data in relational tables, we get all the advantages including indexes and constraints for free. However, the question is what an appropriate relational schema for a given ontology should look like? In a *schema-oblivious representation*, we could store all RDF triples in a single 'triple table', or we could shred the OWL file into XML pieces and store them - which again leaves us on the syntax level [4, 6].

We propose a *schema-aware representation*: Our goal is to process and store data described by an ontology in a relational database in such a way that the schema reflects the semantics given by the ontology. We propose a specific way of doing this that results in a compact, almost 'human readable' schema, and additionally, we show a way how to do this automatically. Our main contributions are

- a set of mapping rules for translating ontolo-

* Current affiliation: University of Münster, Germany

International Conference on Management of Data
COMAD 2006, Delhi, India, December 14–16, 2006

gies written in OWL into a compact relational database schema with index support,

- a two-phase process of schema creation and data loading that uses logical inference during both phases to automatically materialise substantial parts of the ontology's semantics in the relational database,
- a prototypical implementation of a corresponding OWL mapping tool, called *Genea*, and
- an experimental evaluation of the effectiveness and performance of our approach as compared to existing schema-oblivious (Sesame) and schema-aware (DLDB) approaches [3, 13].

We investigate the feasibility and robustness of our schema generation approach using different synthetic and real-world ontologies. The main results are that our approach leads to relational schemas that are smaller (in terms of number of generated tables and views) and contain more semantics (in terms of key constraints and indexes) than comparable OWL and RDF stores. Furthermore, the generated schema also allows for significantly faster query processing and better scalability with the ontology sizes than the existing approaches.

The remainder of this paper is organised as follows: In the next section, we give a brief overview of the background of ontologies and approaches to store them in relational databases, as well as related work. In Section 3, we present our proposed set of mapping rules and the loading process. The results of an experimental evaluation of our mapping rules are presented in Section 4. Section 5 concludes.

2 Background

An ontology is a semantic data model that allows us to describe entities (classes and properties) of the real-world and how they relate. The formal semantics of ontology languages such as OWL allow reasoning about logical consequences of the described entities and their relationships, i.e. facts not literally present in the ontology, but entailed by the meaning. This makes ontologies well-suited to capture the semantic of emerging fields such as bioinformatics where concepts are still evolving or data has to be integrated from several sources.

The Web Ontology Language (OWL) is the W3C standard of an ontology language for publishing and sharing data using ontologies on the web (aka 'Semantic Web') [23]. OWL is an extension of the Resource Description Framework (RDF) [14] and is notated in XML. It has three sublanguages: OWL Lite, OWL DL, and OWL Full. OWL Full has the most expressiveness by allowing predicate logic expressions, but ontologies written in OWL Full can be undecidable; OWL DL is a sublanguage of OWL Full that is more restrictive as it is based on description logics and as such is always decidable; OWL Lite is the most restrictive one with the least expressivity[1]. In the following, we will use OWL DL and will refer to it shortly as OWL.

There are several possible mapping methods for ontology instances written in OWL to a relational schema. For example, as OWL is an extension of RDF and is expressed in XML, every XML or RDF storing system could be used such as, e.g., Sesame [3] or RDF-Store [15]. Following the taxonomy of [22], there are two main approaches for representing OWL and RDF data in databases:

Schema-oblivious mapping. A schema-oblivious representation stores both OWL class and instance data in a single table. This can be done in form of one large universal table in which each row represents one instance of a class [6]. More popular and flexible are however *triple stores*, also called *vertical table*: A single table where each record corresponds to an RDF triple of the form (<*subject-predicate-object*>) [1, 15]. This is the most flexible database schema for storing OWL ontologies, which in addition allows files to be easily parsed and loaded into the database. However, many queries have to recursively search the database. Queries that involve joins or queries about the members of a class will be particularly difficult, because there is no explicit treatment of the class hierarchy.

Schema-aware mapping. Alternatively, we can map the OWL schema information into corresponding tables and then load the instance data into the appropriate tables [13, 7, 22]. The common approach is to generate separate tables per each class and each property. This however leads to a huge number of tables. While simple queries can yield fast response times, complex queries that consider many properties may need many join operations and thus can be very expensive in terms of query time.

A schema-aware mapping promises the best query performance, especially for more complex queries [22]. In [13], a variant of the schema-aware mapping is presented that additionally to the class and property tables captures subsumption hierarchies via dynamic SQL views. Basically, for every super-class, a view is generated that combines all sub-classes instances with the instances of the super-class itself.

Note that the handling of subsumption relationships is orthogonal to the mapping approach. With **load-time inference**, we reason about the class-level ontology meaning during the loading phase of instance data into the database. This allows to materialise all

[1]For example, cardinality values allow only 0 or 1.

memberships of a given individual with explicit tuples in the corresponding tables. The trade-offs are increased loading times and increased storage requirements. On the other hand, **query-time inference** leaves the inference of subsumption relationships to queries. This however complicates query processing which has to retrieve all instances of, e.g., a superclass dynamically. We can combine both *load-time* and *query-time* inference with both schema-oblivious and schema-aware mappings. For example, [13] uses a schema-aware mapping with query-time inference.

The aim of our Genea system is to support read-dominent environments with solely extensional, instance-level queries, such as an on-line web database that is loaded and refreshed periodically from an ontology file. Our main goal is hence to improve the performance of extensional queries as much as possible; the database creation step can be done offline and hence become more complex. Consequently, we combine a schema-aware mapping approach with load-time inference and class-membership materialisation. To further improve query response times, our approach also creates the schema as compact as possible by utilising any OWL cardinality constraints for reducing the number of additional tables.

2.1 Related Work

As OWL is an extension of RDF and is expressed in XML, in principle all XML or RDF/S storage mechanisms can be used to store OWL. There are a number of native XML databases available, such as Tamino [17] or Natix [5]. Furthermore, all major relational database vendors nowadays support storing of XML data within their engine [12, 16, 11]. All these approaches concentrate on mapping and storing of plain XML data; the higher-level semantics of OWL are not considered, which makes reconstructing and querying the meaning of the ontologies very difficult.

Example for RDF stores are Sesame [3] or RDFStore [15]. Sesame is a prototype for storing and querying RDF and RDFS (RDF Schema) [3]. The system is developed in a modular way, offering an exchangeable storing mechanism so that Sesame supports different repositories or DBMS. Sesame can deal with OWL using a special OWL repository called OWLIM which uses a simple triple storage. All those pure RDF and XML storage approaches have the disadvantage that they do not use the underlying semantics and that they do not evaluate the OWL syntax. This complicates queries and increases the amount of data to be stored.

The approach most closely related to this work is DLDB [13] which also presented a schema-aware approach to store ontologies in relational databases. DLDB generates a separate table for each class and property, and also generates SQL views that represent class hierarchies. There are two main differences between DLDB and Genea: Firstly, Genea materialises subsumption relationships at load-time by generating additional tuples in the corresponding class tables, while DLDB relies on views that have to be evaluated at query-time. Secondly, Genea captures the semantics of OWL datatype and object properties by storing all single-valued properties directly in the domain's class table(s) and by introducing explicit primary and foreign keys. DLDB in contrast generates one table per each class and property.

Recently, [22] presented a performance study of different schema-aware and schema-oblivious RDF/S stores. This study concentrated on taxonomic RDF/S queries, while the focus of our evaluation is on database generation and extensional queries on properties and paths.

3 Schema-aware Mapping of OWL into Relations

Our aim is to automatically generate a relational schema that captures as much of the class-level semantics as possible, and that efficiently supports extensional, instance-level queries. In general, we follow a 'standard' schema-aware mapping approach that generates separate tables for each class and property. However, we go beyond this straight-forward mapping in the way we materialise subsumption relationships and how we map OWL property restrictions.

The rationale behind our approach is twofold: we want to materialise as much instance data as possible at load-time instead of dynamically inferring it at query-time; we also aim to reduce the number of tables, which will result in less joins for property-lookup queries. Consequently, we rely on a reasoning tool to pre-compute subsumption relationships during the schema generation and additionally during the loading of instance data into the schema. Such pre-processing will slow down the loading process but has a very positive impact on the query performance (as we will see in Section 4). It can be seen as a kind of materialized inference.

We propose a set of rules for transforming an OWL ontology into a pure relational database schema. Please note that we explicitly do not use any object-relational features for the schema generation; one fundamental design decision of the overall project was to only rely on standard SQL capabilities [9]. This work concentrates on OWL DL. Due to the limited space, we will concentrate our discussion on the most important and central OWL constructs used in the evaluation. For a complete list of all OWL language constructs and their mappings please refer to [10].

3.1 Basic Mapping Rules

In the following discussion, we will present the rules in form of templates for an OWL ontology (prefix `OWL`)

and how they are mapped into the corresponding relational model (prefix RM). The relational primary keys are underlined. We will further use examples from the BioPAX level 1, version 1.4 ontology (shortly: the BioPAX ontology) [2].

Rule 1: Simple Named Classes and Individuals

We map simple named OWL classes to separate relations, and individuals to rows in (one or more) relations. The new relation[2] gets a primary key attribute thing_id that also refers to its super-class. The root of this class relation hierarchy is the Thing relation that centrally maps the URIs of instances to thing_ids.

```
OWL: <owl:Class rdf:about="namespace#classname"/>

 RM: Thing ( thing_id INT, URI string UNIQUE )
     classname ( thing_id INT REFERENCES Thing )
```

As every individual is identified by an URI, the key of the mapped relations could be the URI. However, this naive approach would be quite inefficient with regard to both storage and indexing as the URIs are text values that can be quite long. Hence, we encode all instance URIs into integer values which is also the standard approach taken by most other RFS/OWL stores [3, 13].

Consequently, the Thing relation becomes the central lookup place whenever queries filter for URIs. For further space reduction, it could be split into two parts – a namespace relation that encodes all namespaces into IDs, and a thing relation with just the instance name part of the URI and a foreign key to the namespace table. There is however the usual trade-off between space requirements and query performance, as such a split would introduce an additional join for many filtering queries. Hence we decided to keep namespace and instance name together in the URI attribute of the Thing relation.

All generated relations are **automatically indexed**. We generate primary key constraints for each relation as indicated by the underlined attributes in our rules. Primary keys are automatically indexed by all relational database systems. In the case of the central Thing relation, we also explicitly index the URI attribute (indicated by the UNIQUE keyword in Rule 1) because it will be the central filter attribute for many queries.

Rule 2: Inheritance

OWL allows defining inheritance in ontologies via the construct rdfs:subClassOf[3]. We apply a standard mapping for class hierarchies into relations by creating two separate relations corresponding to the superclass and the subclass and by letting the key of the subclass relation become a foreign key to the parent class' relation:

```
OWL: <owl:Class rdf:about="class2">
       <rdfs:subClassOf rdf:resource="class1"/>
     </owl:Class>

 RM: class2 ( thing_id INT REFERENCES class1 )
```

This approach requires individuals, that are members of the subclass, to be inserted at load-time into several relations: into the subclass plus all parent relations. It however makes it unnecessary to create and query additional views in order to represent the parent classes. This design is in favour of taxonomic queries on the class hierarchy, while queries that also examine attributes from superclasses may require additional joins.

Rule 3: Simple Datatype Properties

Datatype properties define relationships between individuals of classes and data values. Each property has a *name*, a *domain* and a *range*[4]. It specifies that individuals must be members of the domain class if they are related to data values of the range. By default, a datatype property in OWL is multi-valued[5]. We map those multi-valued datatype properties into separate relations which have two attributes: a value attribute storing the property values, and a reference attribute as a foreign key to the relation representing the class to which individuals that have this datatype property must belong.

```
OWL: <owl:DatatypeProperty rdf:ID="propertyname">
       <rdfs:domain rdf:resource="class"/>
       <rdfs:range  rdf:resource="RDF_range"/>
     </owl:DatatypeProperty>

 RM: propertyname (
          class_id INT REFERENCES class,
          value map(RDF_range) )
```

The straight-forward approach is to define a composite primary key over both attributes. However, the value attributes are often text values without length restriction, and only few databases can handle primary keys over those values. Genea offers two options: It can map such text properties to a VARCHAR which has a implementation-dependent maximum length, or it can introduce an additional attribute called seq_id as an automatically generated integer value, and include this into the composite primary key. In any case, we explicitly index also the value attribute beside the composite primary key index.

[2]Whenever possible, we strive to use the class name as the name for the mapped relation (cf. Rule 1). If however the name of a class itself is not unique in the schema, we prefix it with the corresponding namespace.

[3]Note that every class in OWL implicitly inherits from the class Thing.

[4]A property can have several domains and ranges, in which case we will generate separate tables of corresponding structures.

[5]Single-valued properties can be expressed using cardinality constraints, which are explained later.

The range of a datatype property corresponds to the database data domain; it can be a plain (untyped) or typed literal. A typed literal is an instance of a XML Schema datatype (XSD) class and the OWL specification includes a list of suggested datatype classes from the XSD definition. For these datatype classes, it is possible to provide translations into the datatypes of a concrete database. The OWL specification does not define the class of plain literals, so we always map those values as text. As the datatypes vary from database to database, each database should have its own translator for the datatype classes. The invocation of the `map()` function in above's rule represents this datatype translation step.

Rule 4: Simple Object Properties

Object properties define relations between individuals of two classes. Similar to datatype properties, object properties have a *domain* and a *range*, only that the range can now be any `owl:Class` in OWL DL[6]. By default, object properties define binary, many-to-many relationships between the domain and the range.

We apply the standard mapping to transform a binary many-to-many relationship into a relational schema by creating a new relation for the relationship. The new relation contains two foreign keys to the relation of the domain class and to the relation of the range class, respectively. We refer to the first attribute which has the foreign key to the domain table, as *reference attribute*, and to the other one as *value attribute*.

```
OWL: <owl:ObjectProperty rdf:ID="objpropertyname">
       <rdfs:domain rdf:resource="class"/>
       <rdfs:range  rdf:resource="class2"/>
     </owl:ObjectProperty>

 RM: objpropertyname (
         class_id  INT REFERENCES class,
         class2_id INT REFERENCES class2 )
```

This rule implies that whenever several properties are defined for the same range and domain, each property is mapped to a separate relation. Again, we explicitly index both attributes as well as create a composite primary key index.

3.2 Mapping Example

The following is a cut-out from the BioPAX ontology that demonstrates all OWL constructs we have discussed so far:

Using our mapping rules, the following nine relations are generated to map the given BioPAX fragment: To make the schema in Figure 2 easier to understand, we use light-grey boxes for relations that represent classes, dark-grey boxes for relations representing properties, thick arrows to illustrate foreign keys caused by inheritance, and normal arrows for all other types of foreign keys.

[6]As literals are not OWL classes, datatype and object properties are disjoint in OWL DL – in contrast to OWL Full.

```
<owl:Class rdf:about="#entity"/>
<owl:Class rdf:about="#utilityClass"/>
<owl:Class rdf:about="#interaction">
  <rdfs:subClassOf rdf:resource="#entity"/>
</owl:Class>
<owl:Class rdf:about="#pathway"/>
  <rdfs:subClassOf rdf:resource="#entity"/>
</owl:Class>
<owl:Class rdf:about="#physicalEntity">
  <rdfs:subClassOf rdf:resource="#entity"/>
</owl:Class>
<owl:Class rdf:about="#dataSource">
  <rdfs:subClassOf rdf:resource="#utilityClass"/>
</owl:Class>
<owl:DatatypeProperty rdf:ID="SYNONYMS">
  <rdfs:domain rdf:resource="#entity"/>
  <rdfs:range rdf:resource=
      "http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
<owl:ObjectProperty rdf:ID="DATA_SOURCE">
  <rdfs:range rdf:resource="#dataSource"/>
  <rdfs:domain rdf:resource="#entity"/>
</owl:ObjectProperty>
```

Figure 1: BioPAX cut-out.

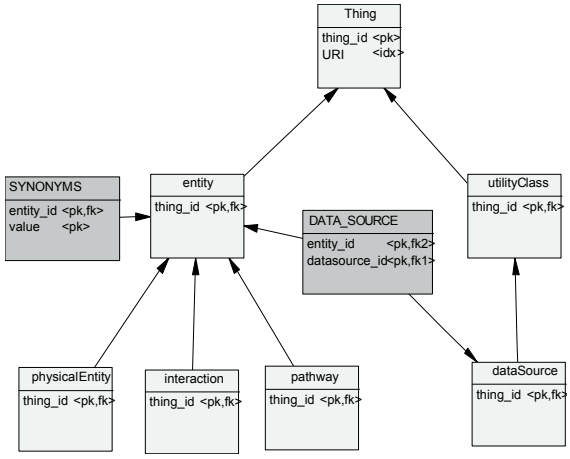

Figure 2: Generated Relational Schema.

3.3 Mapping Rules for Anonymous Classes

Anonymous classes differ from named classes in the way that they are not described by a class name (syntactically represented as a URI reference) but by constraints placed on the class extension. The class extension refers to the set of individuals that are associated with a given OWL class. The individuals in the class extension are called the instances of the class (cf. [23]).

Anonymous classes are always used together with a named class. The way they work together is decisive for the mapping. Possibilities include the use in `rdfs:domain` in object and datatype properties, in `rdfs:range` of object properties or in `rdfs:subClassOf` as seen in the preceding subsection. As a matter of course it is possible to apply

anonymous classes nearly everywhere, where a class description is possible (cf. [23]). In the following, we show the basic concept of the mapping. We introduce anonymous classes by means of `rdfs:subClassOf` and `rdfs:range` for object properties.

The simplest kind of anonymous classes are property restrictions. Commonly, value und cardinality constraints are distinguished.

Rule 5: Cardinality Constraints

OWL allows cardinality constraints on the number of values a property (both datatype and object properties) can take for a particular class. Upper bounds can be set using `owl:maxCardinality`, `owl:minCardinality` sets a lower bound respectively. A shortcut for a min and max cardinality with the same value is `owl:cardinality`. It defines that the number of values has to be exactly some number.

Cardinality constraints allow us to greatly reduce the number of relations in the generated schema: We include all properties with a max cardinality of 1 as an attribute directly in the relation for the class that inherits the anonymous class with the restriction. These attributes are generated corresponding to Rules 3 and 4, with datatype properties mapped to an attribute of appropriate type, and object properties becoming a foreign key attribute.

Furthermore, we map a minimum cardinality of 0 to a `NULL` constraint, and a minimum cardinality of 1 to a `NOT NULL` constraint. The following is an example of a restriction on a datatype property with both min and max cardinality set to 1. The corresponding property is hence included as a `NOT NULL` attribute of an appropriately mapped RDFS type directly in the class' relation.

```
OWL: <owl:Class rdf:about="classname">
     <rdfs:subClassOf>
      <owl:Restriction>
       <owl:onProperty>
        <owl:DatatypeProperty rdf:about="property"/>
       </owl:onProperty>
       <owl:cardinality
  rdf:datatype="http://www.w3.org/2001/XMLSchema#int"> 1
       </owl:cardinality >
      </owl:Restriction>
     </rdfs:subClassOf>
     </owl:Class>

 RM: ALTER classname
       ADD property map(RDF_range) NOT NULL;
```

Max cardinalities of more than 1 are still mapped according to Rules 3 and 4 to separate tables for the domain and object properties, respectively. Genea then additionally defines integrity constraints that ensure the corresponding min and max cardinalities. Depending on the capabilities of the target database systems, this can be done with either table-level `CHECK` constraints or triggers.

Rule 6: Value Constraints

The second kind of constraints are value constraints. Like property ranges, value constraints limit the range of the property, but they apply only to the particular class description. The value constraint `owl:allValuesFrom` corresponds to the range definition for properties. It is used to describe an anonymous class of all individuals for which all values of the property satisfy the specified range.

For our mapping, we suggest checking the range for each property on the class level, on which the property is applied the first time, and ignore later restrictions. As discussed in the previous section, we only map the property once, namely on the level where it is defined first. However, it is not possible to restrict the range without using triggers or database constraints. Only if the property is not inherited, we can use the information of the restriction class directly and apply the restricted range instead of the original one.

```
OWL: <owl:Class rdf:about="classname">
      <rdfs:subClassOf rdf:resource="class1"/>
      <rdfs:subClassOf>
       <owl:Restriction>
        <owl:onProperty>
         <owl:ObjectProperty rdf:about="property"/>
        </owl:onProperty>
        <owl:allValuesFrom rdf:resource="class2"/>
       </owl:Restriction>
      </rdfs:subClassOf>
     </owl:Class>
 RM: classname (
      class1_id INT REFERENCES class1,
      class2_id INT REFERENCES class2 )
```

The OWL value constraint `owl:someValuesFrom` describes that at least one value of the property is an instance individual of the class description or data value in the range. To recollect the difference between the two types of initialization, `owl:someValuesFrom` is analogue to the existential quantifier of predicate logic. As a matter of fact, it is not possible to map this construct without special database features such as constraints or triggers.

[1]The last value constraint is `owl:hasValue`. This constraint is a built-in OWL property that links the anonymous class to either an individual or a data value depending on the kind of property it restricts. Indirectly, this corresponds to constants and can be mapped to custom domains in the database. For datatype properties this is very simple, because all values can be read directly from the ontology and used in the domain. For object properties it is more complicated: because we reference individuals through the URI ID, we first have to generate the URI ID representing the class URI, and then use it in the domain. However, if the property is multi-valued, only one value for the property has to fulfil this constraint and so only database constraints or triggers can ensure the correctness.

The next number of anonymous classes includes `owl:unionOf`, `owl:intersectionOf`, and `owl:complementOf`. These kinds of class descriptions are usually used in description logic and correspond to AND, OR and NOT. For space reasons, we have to concentrate our discussion of the first two, and leave `owl:complementOf` aside.

Rule 7: Union

The `owl:unionOf` class description defines an anonymous class, whose class extension contains the union of all individuals which are members of one of the class descriptions in the list. This is analogue to logical disjunction. For example, in the BioPAX ontology the property PARTICIPANTS can either be an entity, a physicalEntityParticipant or an entity and a physicalEntityParticipant at the same time. We explicitly distinguish three uses of `owl:unionOf`:

1. The first case is `owl:unionOf` together with `rdfs:domain`. This is the simplest case that needs no special consideration as it only defines where the property is applied.
2. The next case is `owl:unionOf` with `rdfs:subClassOf`. This can be handled by explicitly expressing the union by a view; a subclass then gets a foreign key to the view[7].
3. The most complicated case is `owl:unionOf` together with `rdfs:range`. It is explained in more detail in the following.

```
OWL: <owl:ObjectProperty rdf:about="property">
      <rdfs:domain rdf:resource="class"/>
      <rdfs:range>
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="classA"/>
            <owl:Class rdf:about="classB"/>
            ...
          </owl:unionOf>
        </owl:Class>
      </rdfs:range>
    </owl:ObjectProperty>

 RM: property_classA (
           class_id  INT REFERENCES class,
           classA_id INT REFERENCES classA )
     property_classB (
           class_id  INT REFERENCES class,
           classB_id INT REFERENCES classB )
     ...
```

If the union is defined on a multi-valued object property, we generate separate tables for each range and set the foreign keys accordingly (see above). If the union is defined on a single-valued property, we include an attribute for each range class, respectively datatype directly in the classes mapped relations. Unfortunately, this way it is possible to define two participants, so additional triggers have to check this fact.

Alternatively, one can also define a view as the union of both classed' relations (to be more specific, it combines only the URI IDs of both classes) and set a foreign key to this view. At a first glance, the second option is nicer because all the discussed methods of handling cardinality can still be applied. However, the first method bares additional advantages in the context of `rdfs:subPropertyOf` and it additionally avoids joins.

Rule 8: Intersection

Intersections can be used like unions but have to be seen as logical conjunction instead of logical disjunction. They are often defined indirectly by multiple use of a construct such as `rdfs:range`.

```
OWL:<owl:ObjectProperty rdf:about="property">
     <rdfs:domain rdf:resource="class"/>
     <rdfs:range>
      <owl:Class>
       <owl:intersectionOf rdf:parseType="Collection">
         <owl:Class rdf:about="classA"/>
         <owl:Class rdf:about="classB"/>
         ...
       </owl:intersectionOf>
      </owl:Class>
     </rdfs:range>
    </owl:ObjectProperty>

 RM: property (
       class_id INT REFERENCES class,
       value    INT REFERENCES classA,classB,...)
```

For the typical case of an object property with an intersection in combination with `rdfs:range`, we propose the following mapping: The property's relation mapping includes a value attribute that references several tables. Each participating class description in the `rdfs:range` part is mapped as an additional foreign key constraint on this value attribute to the corresponding class' relation. Thus the individuals have to be members of all participating classes. Intersections in domains require a reasoning tool to identify the properties of each class.

For a discussion of mapping rules of intersection for datatype properties or in combination with other RDFS constructs please see [10].

Rule 9: Functional Properties

A functional property can define a global cardinality constraint on either datatype or object properties. It is introduced by `owl:FunctionalProperty` and sets the max cardinality for the property to 1.

```
OWL: <owl:ObjectProperty rdf:ID="property">
      <rdf:type rdf:resource="owl:FunctionalProperty"/>
      <rdfs:domain rdf:resource="class1" />
```

[7]Not supported by the current version of Genea.

```
     <rdfs:range  rdf:resourse="class2" />
    </owl:ObjectProperty>

RM: class1 ( thing_id  INT REFERENCES Thing,
             class2_id INT REFERENCES class2 )
```

We map functional properties in the same way as properties with a max cardinality of 1 (cf. Rule 5): we include the datatype attribute or the foreign key in the domain's relation. The above template demonstrates this with a functional object property.

Rule 10: Inverse Functional Properties

An inverse functional properties defines a global cardinality constraint on an object property. The object (i.e. the *range*) of such a property uniquely determines the subject (some individual as the *domain* of the property). More formally, for an inverse functional property P the following must hold:

$$\forall x, y, z : (P(y, x) \wedge P(z, x)) \Rightarrow y = z$$

This corresponds to a $1 : n$ relationship between the domain and the range of the inverse functional property.

```
OWL: <owl:InverseFunctionalProperty rdf:ID="P">
       <rdfs:domain rdf:resource="classY"/>
       <rdfs:range  rdf:resource="classX"/>
     </owl:InverseFunctionalProperty>

 RM: classX ( thing_id  INT REFERENCES Thing,
              classY_id INT REFERENCES classY )
```

In other words, the inverse of an inverse functional property is a functional property (cf. Rule 9). Consequently, we simply invert the roles of *domain* and *range* in Rule 9: we map an inverse functional property as a foreign key attribute included in the range table that references the domain table.

These ten rules are the most important ones to map OWL ontologies into a relational schema. In particular, these are the rules that we will need to map the benchmark ontologies in Section 4. However, OWL provides more constructs which need rules to be mapped, such as, e.g., Complement or Enumeration. However, due to limited space, we refer the interested reader to [10] for more details on how to map the remaining OWL constructs.

3.4 2-Phase Loading Process

In this subsection, we give a brief overview of the actual mapping process as implemented in our Genea prototype. It consists of two phases: In Phase 1, the ontology is mapped into a relational schema. Phase 2 loads instance data into the freshly generated schema.

Phase 1: Schema Generation

First, the ontology is analysed using an OWL inference engine (in our implementation, we use the Jena framework [20]): all named classes and their properties are determined, including properties that are indirectly defined, e.g. by using equivalences or unions; all cardinality constraints are identified, and unions, intersections, etc. on ranges are determined. Next, the relational schema is generated: first the tables, then indexes and foreign keys are added. In our implementation, it is configurable which parts of the schema shall be generated or whether several depending ontologies shall be loaded. In a last step, code in form of Java importer classes are generated. This generated code will be used in Phase 2 for loading instance data into the mapped schema.

Phase 2: Database Loading

As one result from Phase 1, Genea created a separate importer for each ontology that is also based on the Jena ontology inference framework. Those importers contain all necessary information to load data described by the corresponding ontology into the database. The importers use inference on the level of RDF Schema. In more detail, they use Jena to determine all individuals of one class with all their properties etc., including subClass relationships. Those individuals are then inserted into the corresponding relations. The whole loading process is transactional: before the insertion of a new individual, a safepoint is generated to which the importer can rollback if an individual insertion failed; only in case of a more severe failure, the whole loading process is aborted.

4 Evaluation

In the following, we present the results of an quantitative and qualitative experimental evaluation of Genea.

4.1 Experimental Setup

We have implemented our approach in a generic OWL-mapper tool called **Genea** using Java 1.5 and Jena 2.2, a framework that provides us with a rule-based inference engine for parsing RDF, RDFS and OWL [20]. The architecture of Genea is based on Jastor [18], an open source mapping tool for mapping OWL ontologies to Java classes. We are comparing the performance of Genea with two existing OWL mapping approaches:

DLDB. A schema-aware approach for mapping ontologies into a relational database management system with additional capabilities for DAML+OIL inference [13]. DLDB captures subsumption relationships between classes and properties into dynamic SQL views. We modified DLDB to run on a PostgreSQL database instead

	Ski	LUBM	Food	Wine	BioPAX 1.4
# of classes	7	43	63	74	27
# of properties	18	28	4	12	49
# of single-valued datatype class properties	8	0	0	1	20
# of multi-valued datatype class properties	4	4	0	1	14
# of single-valued object class properties	5	0	0	6	9
# of multi-valued object class properties	4	28	5	7	17

Table 1: Comparison of Ontology Complexities.

of Microsoft Access, for which it was originally developed, to allow for a fair comparison.

Sesame. A triple-store for RDFS with a PostgreSQL back-end [3]. It consists of one central triples table and 26 meta-data tables capturing RDFS semantics such as class memberships and subclass relationships. We used Sesame's RDFS repository that infers and materialises at load-time all subsumption triples based on the OWL/RDFS schema information.

We compare those three systems on five different, commonly used ontologies written in OWL:

- LUBM—the Lehigh University BenchMark ontology for OWL stores [8],
- Jastor's Ski ontology [19],
- the W3C sample wine ontology [25],
- the W3C sample food ontology [24], and
- the BioPAX v1.4 ontology [2].

While the first four are synthetic ontologies for benchmarking and demonstration purposes, the last ontology, BioPAX, is a standard exchange format for representing metabolic pathway information in bioinformatics. The BioPAX ontology is of most interest for us because Genea is intended for creating an efficient store for metabolic pathway data.

The Ski ontology is an OWL Full ontology containing RDF constructs, hence it tests the capability of the storage to handle OWL Full. The Wine and Food ontologies depend heavily on each other and can serve as test for the capability of solving the dependency between the ontologies. Additionally, those two ontologies make heavy use of equivalent classes. Table 1 provides some general statistics for each ontology.

We state the number of classes and properties contained in each ontology. To indicate how the properties are applied to the classes, we count the single- and multi-valued datatype and object properties. These figures only include the properties which are defined directly, not inherited ones.

Environment: All experiments are performed on a PC with a 3 GHz Intel Pentium IV processor, 2 GByte RAM, 120 GByte hard disk, and with Windows 2003 Server SP2 as operating system. All experiments are run on a PostgreSQL 8.1 database with a JDBC 3 driver. Genea itself is implemented in Java and runs on Suns JDK 1.5.0.

4.2 Comparison of Schema Complexity

First, we are interested in a general comparison of the complexity and the quality of the generated relational schemas. We tried to generate relational schemas for each of the five test ontologies using Genea, DLDB, and Sesame. Unfortunately, with DLDB it was only possible to generate the schema for the Lehigh University LUBM benchmark ontology. At least for the ski ontology, this was to be expected as DLDB cannot handle OWL Full. However, Genea and Sesame were capable of generating the relational schema for all ontologies.

	Genea	DLDB	Sesame
# of tables	76	80	27
# of foreign keys	109	0	0
# of views	0	75	0
# of indexes	141	81	37

Table 2: Schema Statistics for LUBM Ontology.

So for this comparison, we are concentrating on the LUBM ontology. We collected statistics about the generated schemas such as the number of tables, views, foreign keys, and indexes (cf. Table 2). These figures can serve as indicators for the quality of the generated schema. For example, less tables and views reduce the complexity of the schema, while foreign keys indicate how many constraints are mapped from the ontology to the schema, and the number of indexes indicates how optimized the schema is. Still all measures have to be considered carefully. While fewer tables are in general more scalable for larger ontologies with thousands of classes (DLDB and Genea are limited by the maximal schema size of the underlying database), they can also limit query performance. An additional indicator for the quality of the schema is hence the query complexity for extensional queries. This aspect of quality is discussed later.

Table 2 shows the statistics and the results of our experiments with the LUBM ontology. It shows that in terms of generated tables, both Genea and DLDB are very comparable. The reason is that LUBM's University ontology does not include any cardinality constraints[8] — one of the main optimisation possibilities of Genea. The remaining differences between Genea and DLDB are in the way how both approaches incorporate the reasoning into the schema: DLDB uses

[8]E.g., in LUBM a person could have more than one age...

views, Genea achieves the same with an additional loading step. In addition, Genea captures a large number of constraints by generating corresponding foreign key relationships. It also automatically provides an optimised physical design by generating appropriate indexes.

4.3 Performance of Database Loading Phase

The next phase is to generate the schema and to load the ontology instances into the generated database. We measured the database loading performance with each of the three OWL stores. For the following experiment, we again concentrate on the LUBM benchmark as the only ontology being supported by all three storages. We used the data generator tool UBA version 1.7 [21, 8] to randomly and repeatable generate individual data for LUBM's University ontology. This produces a synthetic data set with a pre-defined number of 'Universities', each consisting of approximately 10 MB OWL text files. We identify the data sets using a slightly different notation as in [8]:

LUBM n: A data set that contains n universities beginning at 'University0' and that is generated using a seed value (for the random number generator) of 0; each test database has a size of circa $n \times 10$ MBytes.

In the following, we will refer to the database scaling value n as 'LUBM benchmark factor'. We have built seven test sets (LUBM1, LUBM5, LUBM10, LUBM15, LUBM20, LUBM25 and LUBM50) that correspond to test data sizes of 10 to 500 MBytes. We also optimised the LUBM benchmark measuring tool, UBT version 1.1 [21], to use bulk loading and reduced its logging. Figure 3 shows the results of the loading in seconds for the different datasets.

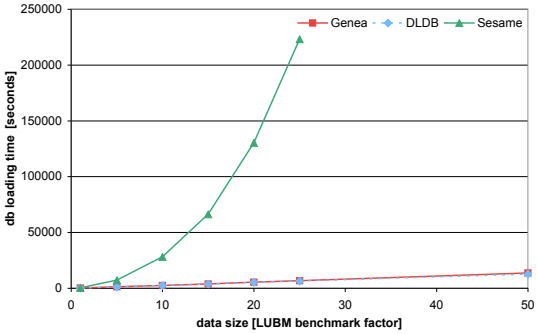

Figure 3: Database Loading Times in Seconds.

Both DLDB and Genea essentially show the same linearly increasing loading times. Only for the last three data sets, DLDBs performance is slightly faster than Genea because it does not materialise the subsumption relationships. Sesame is significantly slower. It needs more than 61 hours to load the LUBM25 data set, and as is shows an exponential loading behaviour, we stopped further loading tests after that. These values are consistent with [8] which reported loading times of 46 hours for LUBM20 on a slower server. The reason for Sesame's slow loading times is that in these tests, it is using an RDF-Schema repository that precomputes triples inferred from subsumption relationships at load-time. While Genea is also precomputing the subsumption relationships at load-time, it does so in main memory using the generated data loader and without need to access the already loaded data. As shown in Figure 3, this pays off with a much better loading performance as compare to Sesame.

4.4 Query Performance

We are finally interested in the query response times with the three OWL stores. In the following discussion, we concentrate on a subset of three queries of the LUBM benchmark [8] as shown in Table 3. Genea

LUBM Query#	result size	selectivity	description
$Q3$	6	high	property lookup
$Q6$	thousands	low	class query
$Q7$	59	high	path query

Table 3: Benchmark Queries

provides a SPARQL query processor that translates SPARQL queries into corresponding SQL queries[9], while the other two systems evaluated RQL versions of the queries; Table 4 gives an overview of the different query complexities with each OWL storage.

	Genea			DLDB			Sesame		
	joins	unions	selects	joins	unions	selects	joins	unions	selects
$Q3$	3	0	0	3	9	4	0	0	27
$Q6$	1	1	1	1	2	2	1	0	4
$Q7$	5	1	1	6	3	5	5	0	548

Table 4: Query Complexities

It shows that Genea and DLDB need comparable complex SQL queries, with a slight advantage for Genea, in particular on queries $Q4$ and $Q7$. The difference would have been clearer with an ontology which makes more use of cardinality constraints (cf. Mapping Rule 5), but unfortunately the LUBM ontology does not use those extensively.

There is a clear difference between the schema-aware approaches of DLDB and Genea, and Sesame. Sesame is a triple storage approach and therefore it needs multiple SQL queries to retrieve the result. This

[9]However note that we measured the response time of just the translated SQL queries as at the time of experiments the SPARQL query processor was not yet finished.

shows in much more complex SQL queries, again in particular for queries $Q4$ and $Q7$.

4.4.1 Property-lookup Queries

The first test is using a property-lookup query ($Q3$ — 'all the publications of a given professor') with high selectivity on large data sizes. The results are shown in Figure 4.

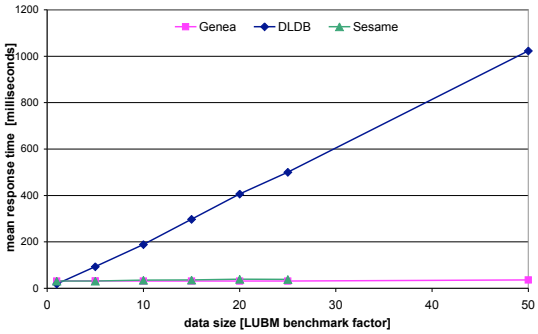

Figure 4: Response Times of Query $Q3$.

Genea and Sesame[10] show both a linear and even almost constant performance for those high-selective queries. They both benefit from the automatically generated indexes such as an index on the URI attribute on Genea's central 'Thing' table. DLDB shows a much worse linear scalability because it has no single indexes on the individual attributes of property tables; in the case of query $Q3$, it also accesses two views which encapsulate the 'publication' class-hierarchy, resulting in 10 unions over all base tables with linear complexity in the data size. Genea in contrast materialised those class subsumption relationships at load-time and needs to access only one publications table.

4.4.2 Class-Hierarchy Queries

Next, we are interested in the performance of the three OWL stores with class-hierarchy queries as represented by LUBM query $Q6$. This query lists all members of the STUDENT class which is in the mid-level of the person class hierarchy, and produce thousands of results. The runtime results are shown in Figure 5.

All three approaches show again a linear query response time. However, Genea shows a much better scalability than DLDB and Sesame, whose response times are now very similar. In this case, the query costs are dominated by the determination of class membership and the joins with low selectivity (the query does not have any filter predicate), and hence indexes are not the main performance factor. Genea mainly benefits from its materialised subsumption hierarchy, while DLDB again needs to union several tables before joining with its 'url_index' table.

[10]Due to the exponential loading times, we evaluated Sesame only up to LUBM25. The trends should become clear though.

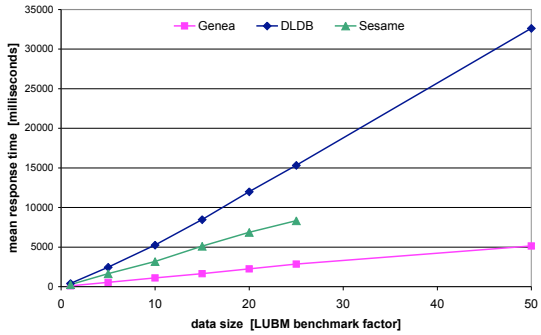

Figure 5: Response Times of Query $Q6$.

4.4.3 Complex Path Queries

Finally, we compare the three approaches with a complex 'path' query. Query $Q7$ determines all students who take a course of a certain professor. This query includes a filter predicate on an URI and several joins (cf. Table 4) and with DLDB also several union operations; Sesame as triple store produces a large number of individual small queries.

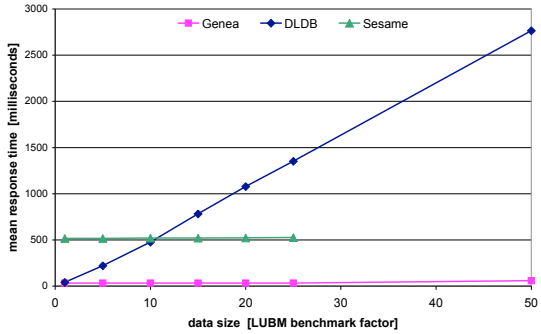

Figure 6: Response Times of Complex Queries $Q7$.

The results, as shown in Figure 6, reflect these different query complexities. Genea shows reasonably good response times below 60 milliseconds even for the largest test database. The filter predicates enable it to make effective use of the index on the URI attribute of the Thing table. DLDB is again much slower, showing a much worse scalability and response times of several seconds for query $Q7$. The main reason is that DLDB needs more complex queries in order to expand the views that encapsulate the accessed class hierarchies (student and professor). The triple store solution Sesame requires a lot of small queries to produce the result. But it actually has a better scalability than DLDB for query $Q7$, where it consequently outperforms DLDB for larger data sets.

5 Summary and Conclusions

We have developed a set of mapping rules to transform ontologies specified in the Web Ontology Language (OWL) into efficient relational database schemas. Our approach materialises a part of the logical inference of

an ontology directly into the relational schema. Most prominently, it evaluates the semantic constraints specified in an ontology to minimise the number of generated tables, and to automatically generate appropriate foreign keys and indexes.

Using the *Genea* OWL-mapper, our prototypical implementation of the suggested rules, we demonstrated that our approach is applicable for a wide range of ontologies — both synthetically generated and real-world ontologies. In particular, we were able to map the real-world example of the BioPAX ontology taking full advantage of the broad range of the cardinality constraints used in that specification.

We compared the performance of our approach with two other OWL and RDF stores, DLDB and Sesame, using the LUBM University benchmark. It showed that for extensional queries, a schema-aware approach to mapping OWL into a relational database can provide better performance than pure RDF triple stores. This finding is consistent with [22], although our results show that indexes and materialisation are essential. In most cases, Genea showed the best query performance and scalability; in particular, it outperformed DLDB because of its automatically generated indexes, and because it materialises subsumption relationships during load-time.

This materialisation is also done by the RDFS triple store which however showed the by far slowest database loading times with exponential slow-down. Its querying performance however was mixed: For simple queries with high selectivity, the triple store performed well and sometimes as fast as Genea. However, for complex queries or queries with low selectivity, it provided far slower querying times than Genea.

References

[1] S. Alexaki, V. Christophides, G. Karvounarakis, and D. Plexousakis. On storing voluminous RDF descriptions. In *Proc. of the 4th Int. Workshop on the Web and Databases (WebDB)*, pages 43–48, 2001.

[2] BioPAX core group. *BioPAX Level 1, Version 1.4*. URL: http://www.biopax.org/.

[3] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *Proceedings of the 1st International Semantic Web Conference (ISWC)*, volume 2342 of *LNCS*, pages 54–68, 2002.

[4] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the 1999 ACM SIGMOD Int. Conf. on Management of Data, June 1-3, Philadelphia*, pages 431–442, 1999.

[5] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4):292–314, December 2002.

[6] D. Florescu and D. Kossmann. Storing and querying xml data using a rdbms. *IEEE Data Engineering*, 22(3):27–34, September 1999.

[7] A. Gali, C. X. Chen, K. T. Claypool, and R. Uceda-Sosa. From ontology to relational databases. In *Proceedings of ER2004 Workshop on Conceptual Modeling for Web Information (CoMWIM)*, 2004.

[8] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2):158–182, 2005.

[9] J. Ho, T. Manwaring, S. Hong, U. Röhm, K. Xu, D. Fung, and T. Kraska. PathBank: Web-based querying and visualisation of an integrated biological pathway database. In *Proc. of the Int. Symposium on Bioinformatics Visualization (BioViz'06)*, 2006.

[10] T. Kraska. Genea Store: Storing OWL in relational databases using generic mapping rules. Master's thesis, School of IT, University of Sydney, 2005.

[11] V. Krishnamurthy. Oracle XML DB repository. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, CA, USA, June 9-12*, page 635. ACM Press, 2003.

[12] M. Nicola and B. van der Linden. Native XML support in DB2 universal database. In *Proceedings of the 31th International Conference on Very Large Data Bases (VLDB)*, pages 1164–1174. ACM Press, 2005.

[13] Z. Pan and J. Heflin. DLDB: Extending relational databases to support semantic web queries. In *Proc. of the 1st Int. Workshop on Practical and Scalable Semantic Systems (PSSS1)*, pages 109–113, 2003.

[14] W. Recommendation. *Resource Description Framework (RDF) Primer.*, Feb. 2004.

[15] A. Reggiori. RDFStore: Perl/C RDF storage and API. URL: http://rdfstore.sourceforge.net, 2004.

[16] M. Rys. XML and relational database management systems: Inside Microsoft SQL Server 2005. In *Proceedings of the 2005 ACM SIGMOD Conference on Management of Data*, pages 958–962, 2005.

[17] H. Schöning. Tamino - a DBMS designed for XML. In *Proceedings of the 17th Int. Conference on Data Engineering (ICDE)*, pages 149–154, 2001.

[18] B. Szekely and J. Betz. Jastor: Typesafe, ontology driven RDF access from java. URL: http://jastor.-sourceforge.net/. Last accessed: 12.3.2006.

[19] B. Szekely and J. Betz. Ski ontology. URL: http://-jastor.sourceforge.net/ski.owl. Last access: 03.2006.

[20] The Jena Team. Jena – a semantic web framework for java. URL: http://jena.sourceforge.net/.

[21] The Semantic Web and Agent Technologies Lab (SWAT). The Lehigh university benchmark (LUBM). URL: http://swat.cse.lehigh.edu/projects/lubm/.

[22] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of RDF/S stores. In *Proceedings of the 4th Int. SemanticWeb Conference (ISWC)*, 2005.

[23] W3C. The web ontology language (OWL). URL: http://www.w3.org/2004/OWL/.

[24] World Wide Web Consortium (W3C). Food ontology. URL: http://www.w3.org/TR/2003/CR-owl-guide- 20030818/food. Last accessed: 12.3.2006.

[25] World Wide Web Consortium (W3C). Wine ontology. URL: http://www.w3.org/TR/owl-guide/wine.rdf.