

Unifying Data and Domain Knowledge Using Virtual Views

Lipyew Lim
IBM T.J. Watson Research Ctr.
19 Skyline Dr.
Hawthorne, NY 10532
liplim@us.ibm.com

Haixun Wang
IBM T.J. Watson Research Ctr.
19 Skyline Dr.
Hawthorne, NY 10532
haixun@us.ibm.com

Min Wang
IBM T.J. Watson Research Ctr.
19 Skyline Dr.
Hawthorne, NY 10532
min@us.ibm.com

ABSTRACT

The database community is on a constant quest for better integration of data management and knowledge management. Recently, with the increasing use of ontology in various applications, the quest has become more concrete and urgent. However, manipulating knowledge along with relational data in DBMSs is not a trivial undertaking. In this paper, we introduce a novel, unified framework for managing data and domain knowledge. We provide the user with a virtual view that unifies the data, the domain knowledge and the knowledge inferable from the data using the domain knowledge. Because the virtual view is in the relational format, users can query the data and the knowledge in a seamlessly integrated manner. To facilitate knowledge representation and inferencing within the database engine, our approach leverages XML support in hybrid relational-XML DBMSs (e.g., Microsoft SQL Server & IBM DB2 9 PureXML). We provide a query rewriting mechanism to bridge the difference between logical and physical data modeling, so that queries on the virtual view can be automatically transformed to components that execute on the hybrid relational-XML engine in a way that is transparent to the user.

1. INTRODUCTION

Since the introduction of the relational data model, and its success in managing transactional data, various extensions have been proposed in the past decades so that data in different domains or applications of different nature can be brought into the relational world to be managed in the same rigorous and elegant manner. For example, the need to model object-oriented data relationships eventually gave birth to the Object-Relational DBMSs, which has since become the industry standard for database vendors. In the 1990s, on-line analytical processing (OLAP) distinguished itself from traditional transaction processing by providing support for better decision making. This brought the mechanism of data cubes, which enabled the data to be viewed from many different business perspectives. Recently, data mining has become increasingly important in day-to-day business, and as a result, various data mining oriented language and system extensions have been introduced by major database vendors.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '07, September 23-28, 2007, Vienna, Austria.

Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

Notwithstanding the progress that has been made, there is a desire to operate on the data as well as the knowledge associated with the data. The quest has been driving the database community to create better data models, languages, and systems. Recently, it has been intensified in several new application areas, including the semantic Web, for which the paramount interest lies in data semantics understanding and knowledge inferencing rather than simple transactional or analytical data processing.

Unfortunately, current DBMSs, albeit improved by many extensions over the past years, are not ready to manipulate data in connection with knowledge. More and more applications are developing ad-hoc systems that deal directly with ontologies. Still, since data is managed by DBMSs, it is desirable that the domain knowledge is managed in the same framework, so that users can query the data, the domain knowledge, and the knowledge inferred from the data in the same way as querying just relational data. We call such an effort *semantic data management*.

In order to support semantic data management in DBMSs, new extensions are required to bridge the gap between data representation and knowledge representation/inferencing. Towards this goal, we propose a framework that extends a DBMS to operate on data and their semantics in a seamlessly integrated manner. To insulate the users from the details of knowledge representation and inferencing, we present the users with a unified view, through which knowledge appears to be no different from data – it is manipulated by relational operators, and is fully incorporated and supported within the DBMSs. Before diving into the details of our method, we use an example to illustrate the task we are undertaking.

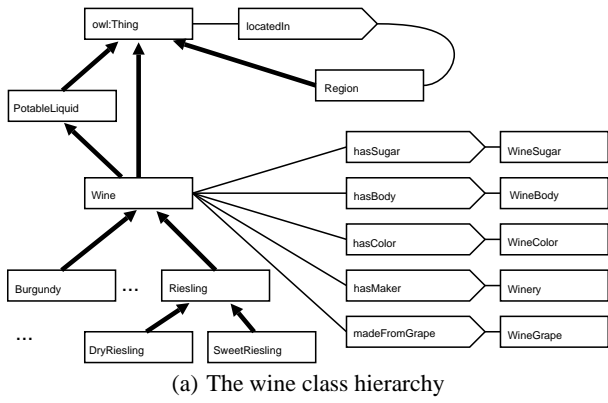
A Motivating Example. Consider a relational table for wines, as shown in Table 1. Every row in the wine table is associated with a specific instance of wine. Each wine has the following attributes: *type*, *origin*, *maker*, and *price*. A relational DBMS allows us to query wines through these attributes. The expressive power of such queries is known to be relational complete, which in a certain sense, is quite limited.

Human intelligence, on the other hand, operates in a quite different way. Humans have the ability to combine data with the domain knowledge, and this process sometimes takes place subconsciously. Let us consider the following two examples.

Id	Type	Origin	Maker	Price
1	Burgundy	CotesDOr	ClosDeVougeot	30
2	Riesling	NewZealand	Corbans	20
3	Zinfandel	EdnaValley	Elyse	15

Table 1: The Wine base table

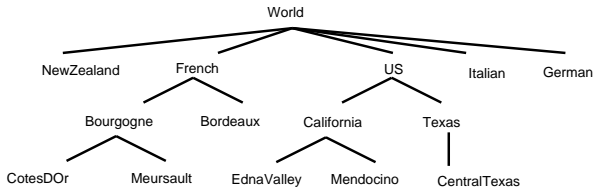
- When asked which wine originates from the United States (US), one would answer Zinfandel because its origin Ed-



(a) The wine class hierarchy

$(type=CotesDOr) \Rightarrow (hasFlavor=moderate)$
 $(type=CotesDOr) \Leftrightarrow (type=RedBurgundy) \wedge (origin=CotesDOrRegion)$
 $(type=RedBurgundy) \Leftrightarrow (type=Burgundy) \wedge (type=RedWine)$
 $(type=RedBurgundy) \Rightarrow (madeFromGrape=PinotNoirGrape)$
 $(type=RedBurgundy) \Rightarrow (madeFromGrape.cardinality=1)$
 $(type=RedWine) \Rightarrow (type=Wine) \wedge (hasColor=red)$
 $(type=Zinfandel) \Rightarrow (hasColor=red)$
 $(type=Zinfandel) \Rightarrow (hasSugar=dry)$

(b) A subset of the implications.



(c) The locatedIn property.

Figure 1: The wine ontology consists of a class hierarchy, implications or rules, and properties.

naValley is located in California. The fact that EdnaValley is in California, and California is in the US, is not explicitly represented in the data shown in Table 1, but belongs to the domain knowledge of geographical regions.

- When asked which wine is a red wine, one would answer Zinfandel and Burgundy. It is a known fact that Zinfandel is red, and although Burgundy can be either red or white, the Burgundy wines originating from CotesDOr are always red.

Clearly, the domain knowledge required to answer such queries is not present in the relational table.

Ontology. The first step toward solving the problem is to make domain knowledge machine accessible. In Fig. 1, we show the well-known wine ontology [25], which is used in the OWL guide [18]. For more information on the data model and syntax of the OWL ontology language, please consult [19, 18].

The wine ontology consists of i) a class hierarchy of objects, ii) properties associated with each object class, and iii) rules governing the objects, their properties, and the values these properties may take. Fig. 1(a) shows part of the class hierarchy in the wine ontology. The wine class is associated with five properties (`hasSugar`, `hasBody`, `hasColor`, `hasMaker`, `madeFromGrape`) and in-

herits one property (`locatedIn`) from its superclass `owl:Thing`. Each property is associated with a range class: values of the property are restricted to instances of the range class. For example, the `hasMaker` property takes values that are instances of the `Winery` class. Fig. 1(b) shows a subset of the rules in the wine ontology. The first rule prescribes that all instances of wine in the `CotesDOr` subclass have moderate favor. Fig. 1(c) shows the `locatedIn` property for all region object instances. Note that the `locatedIn` property is a property of the `owl:Thing` class and takes values that are instances of the `Region` class. The wine ontology also specifies the `locatedIn` property to be transitive; hence, all the `locatedIn` relations on region instances form a tree (or a directed acyclic graph).

Although the ontology as shown in Fig. 1 contains enough information to answer the two queries we mentioned before, they are unfortunately not in the relational form. Hence, relational DBMSs cannot make use of such information while evaluating the above queries. Nevertheless, an increasing number of applications require interaction with domain knowledge during data processing. It is much desirable if domain knowledge can be managed in DBMSs. The benefits are two-fold. First, in many cases, the data already resides in the DBMS, and the DBMS provides a wide range of transactional and analytical support that is indispensable in data processing. Second, a declarative query language such as SQL can insulate the users from the details of data representation and manipulation, while offering much opportunity in query optimization. This is a critical requirement in handling domain knowledge, which has flexible forms.

Our Whimsical Approach. Before we present our method for supporting semantic queries in RDBMSs, we ask, what is the most desirable way to express a semantic query in SQL? If possible, we would like to express the queries in the following way.

Example 1 (Semantic Query on Location) To find wines that originate from the US, we may naively issue the following SQL query:

```

SELECT W.Id
FROM Wine AS W
WHERE W.Origin = 'US';

```

Example 2 (Semantic Query on Wine Color) To find red wines, we may naively issue the following SQL query:

```

SELECT W.Id
FROM Wine AS W
WHERE W.hasColor = 'red';

```

Of course, neither of the above queries will return the intended results. For Example 1, none of the wines in the relational table has “US” as the value in the `Origin` column, thus no tuples will be returned. In order to provide semantically correct answers, the DBMS must know not only that `Origin` denotes a location but also location’s semantics, which is illustrated in Fig. 1(c). For Example 2, we engage an imaginary `HasColor` attribute for the wine table in the query. However, `HasColor` is not in the schema of the wine table. This is even more challenging than the previous query. In order to support the query in Example 2, first, both the user and the DBMS must know what `HasColor` stands for when it appears in a query, and how to derive the value for `HasColor` for any given wine.

The above examples in SQL are nothing more than our whimsical desires to marry domain knowledge and SQL, which seem to be as highly incompatible as it could be. In essence, this reflects a situation that has been long bothering the database community: on the one hand, we want to extend our arena as far as possible, but on the other hand, we are not ready to give up the comfort we have enjoyed in the spartan simplicity of SQL and the relational model.

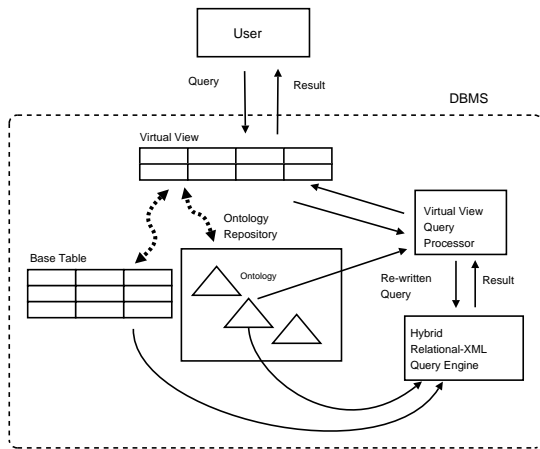


Figure 2: A schematic diagram of our framework for querying base table data with meaning from an ontology.

In this paper, we present a framework to support the queries we have shown above in the relational framework. Our endeavor focuses on making the transition from query-by-value to query-by-meaning as smooth as possible.

Other Challenges. Before we start working on a new modeling approach in order to accommodate the above queries in RDBMSs, we must first address two issues: how to store and access the domain knowledge or the ontology, and how to infer new knowledge.

Ontology data is of very different form from relational data. Because of this difference, XML-based protocols including RDF [11], RDFS [1], DAML+OIL [2], and OWL [19], have emerged as standards for encoding ontology. In other words, ontology is currently represented as semi-structured data. The relational data model remains ill-suited for storing and processing the semi-structured data efficiently. The flexibility of the XML data model, on the other hand, appears to be a good match for the required schema flexibility. However, the flexibility of XML in modeling semistructured data usually comes with a big cost in terms of storage and query processing overhead, which to a large extent has impeded the deployment of pure XML databases to handle such data.

Knowledge inferencing is an even more daunting challenge. It can be highly complicated as it engages a lot of details of the domain ontology. For instance, an ontological relationship can be transitive, and transitive relationships are involved in many useful queries (such as Example 1, which in essence, queries locations based on the `subregionOf` relationships). However, transitivity is difficult to express and costly to execute: In RDBMSs, we often have to resort to recursive SQL queries and this approach has been studied in [4]. Currently, to provide efficient support for ontology-based semantic queries in a DBMS, a well-known approach pre-processes the ontology and materializes the transitive closures for all transitive relationships in the ontology. For instance, materializing the `subregionOf` relationship will result in a table that contains every pair of locations (x, y) as long as x is a subregion of y . The main problem with this approach is its huge time and storage overhead. Furthermore, once the transitive closures have been materialized, it makes update of ontology data almost impossible.

In view of these challenges, we argue that neither pure relational nor pure XML databases can accomplish the task alone. In our framework, we support ontology-based semantic queries in a hybrid relational-XML DBMS, i.e., a RDBMS with support for XML data and XML queries.

Overview of our approach. We introduce a framework that aims at supporting a rich class of semantic-related queries within DBMSs in an easy-to-express and potentially efficient-to-process manner. A schematic diagram of our framework is shown in Fig. 2.

As shown in the figure, we create a relational virtual view on top of the data and the domain knowledge. A virtual view is created by specifying how the data in relational tables relate to the domain knowledge encoded as ontologies in the ontology repository. Through this virtual view, data and knowledge can be queried together, new knowledge can be derived, and our whimsical ideas in Example 1 and Example 2 can be realized. The virtual view is an interface through which users can query data, domain knowledge, and derived knowledge in a seamlessly unified manner.

In order to support the virtual view, we augmented a DBMS with an ontology repository for managing ontological information. Before ontologies can be used in the DBMS, users must first register ontology files with the ontology repository. These ontology files are then pre-processed into a representation more suitable for query processing: class hierarchies and transitive properties are extracted into trees, and implications are extracted into an implication graph. These trees and graphs are encoded and stored as XML data. Clearly, RDBMSs cannot meet this challenge, which is the reason we base our framework on DBMSs with native XML support.

Once the virtual view is created, SQL queries can be written against it just like against any other relational table. Our framework processes the queries on the virtual view by re-writing them into queries on both the base table and the ontological information in the ontology repository. Our query re-writing uses the implication graph to expand the predicates and then leverages on SQL/X [7] and XPath for subsumption checking. The re-written queries can be processed natively by the DBMS query engine and the results returned to the user with minor re-formatting.

Paper Organization. In Section 2, we introduce virtual views that aim at unifying data and the domain knowledge. Section 3 briefly introduces the key features of a hybrid relational-XML DBMS. Section 4 describes how we support ontology data in a hybrid DBMS. Section 5 describes how we express semantic queries. We review some related work on ontology-based semantic queries in Section 7. Conclusions are drawn in Section 8.

2. VIRTUAL VIEWS

In order to support semantic queries, the RDBMS must be further extended so that knowledge representation can be incorporated into the relational framework, and the manipulation of knowledge can be conducted no differently from the manipulation of data. To satisfy these requirements, we propose the concept of *virtual view*. We adopt a minimalist’s approach to provide the user with a unified view of the data and the knowledge. Through the virtual views, we offer a rich set of functionalities for knowledge inferencing out of the spartan simplicity of SQL.

2.1 Knowledge is a View

Our goal is to express semantic queries in SQL with little divergence from our whimsical desires as shown in Example 1 and Example 2. In this section, we continue to use the wine table as an example.

Imagine the wine table shown in Fig. 1 is appended by two virtual columns, `LocatedIn` and `HasColor`, as shown in Table 2. The meanings of the two virtual columns are as follows.

- For every wine of Origin x , its `LocatedIn` value is a set of locations $\{y_1, \dots, y_n\}$, such that x is a sub-region of y_i , as prescribed by the location property shown in Fig. 1(c).

Id	Type	Origin	Maker	Price	LocatedIn	HasColor
1	Burgundy	CotesDOr	ClosDeVougeot	30	{Bourgogne, France}	red
2	Riesling	NewZealand	Corbans	20	{}	white
3	Zinfandel	EdnaValley	Elyse	15	{California, US}	red

Table 2: WineView: A Virtual View

For instance, wine Burgundy originates from CotesDOr, which is a sub-region of Bourgogne, which in turn, is a sub-region of France. As a result, its LocatedIn value is {Bourgogne, France}.

- The other virtual column, HasColor, is introduced from the wine ontology. The ontology includes a set of rules. For example, the following rules are present:

(type = Zinfandel) ⇒ (hasColor = red)

(type = Riesling) ⇒ (hasColor = white)

Thus, for wines of type Zinfandel, we can derive the value of its HasColor column as red.

We can append as many virtual columns as we like onto the original table. The virtual view incorporates both the data and the domain knowledge that associated with the data. However, it is a virtual view, which means none of the values in the virtual columns are materialized. The purpose of introducing this virtual view is (a) to inform the user what data can be queried, and (b) to inform the system how to derive values for the virtual columns from the raw data and the ontology when needed. In this section, we focus on the first issue, and leaves the second issue to later sections.

With this unified view of the data and the domain knowledge, it is no longer difficult for us to ask queries that manipulate both the data and their meaning. In the following, we revisit the two queries in Example 1 and Example 2, but this time we ask the queries against the virtual view instead of the original wine table.

Example 3 (Semantic Query on Location) *To find wines that originate from the US, we issue the following SQL query against the virtual view:*

```
SELECT W.Id
FROM WineView AS W
WHERE 'US' IN W.LocatedIn;
```

Example 4 (Semantic Query on Wine Color) *To find red wines, we issue the following SQL query against the virtual view:*

```
SELECT W.Id
FROM WineView AS W
WHERE W.HasColor = 'red';
```

We can see that Example 4 is the same as Example 2 except that HasColor is a valid (virtual) column in the view, and the only difference between Example 3 and Example 1 is the use of the set-valued virtual column LocatedIn.

2.2 The Virtuality of the View

At the first look, one may argue a schema of the form as shown in Table 2 violates relational normal forms. For example, a location can be a sub region of many other locations. For any wine, the set of its LocatedIn values only depends on the Origin of the wine, which means the two columns Origin and LocatedIn should be isolated and made into a table on their own. Same argument goes against HasColor, which depends on the Type of the wine.

We argue that this is not a concern because Table 2 is a virtual view. The introduction of such a view is solely for the database user, so that she can query the data and the domain knowledge as if they are both in relational tables.

In fact, a user can imagine that the LocatedIn column is created by a join operation.

Example 5 (User’s Viewpoint) *From a user’s point of view, the virtual view can be seen as the result of joining the wine table with a “knowledge” table.*

```
CREATE VIEW WineView(Id, Type, Origin,
                    Maker, Price, LocatedIn) AS
SELECT W.*, R.superRegions
FROM Wine AS W, RegionKnowledge AS R
WHERE W.Origin = R.region
```

In Example 5, we assume there is a “knowledge” table called RegionKnowledge(region, superRegions), which stores for each region all of its super regions as a set. For example, (CotesDOr, {Bourgogne, France}) is a tuple of this knowledge table. We can also create the HasColor column in the same way. Thus, from a user’s view point, the view we introduced in Table 2 is just a shorthand for specifying the joins.

Example 5 shows how the user thinks what the view represents. However, the view never exists in the system as a materialized table. In addition, the join operations shown above will never take place, not even in query time. A virtual view is different from traditional views in that the “knowledge” tables used in creating the view as shown by Example 5 does not exist in real life.

Instead, the system must remember how to derive the values of the virtual columns from the values in the base table. This may involve reasoning over the ontology, which will be carried out automatically when a query is issued against the virtual view. Thus, the process of creating a virtual view is the process of informing the system of how to derive such values when needed. We describe this in detail in Section 2.3.

2.3 Marrying Relational Tables and Ontology

Beneath the virtual view lie the data and the ontology, which, when properly integrated, produce knowledge queryable through the virtual view. The integration is carried out by a CREATE VIRTUAL VIEW statement. It is part of the language extension we introduce to support semantic queries in DBMSs.

In essence, the CREATE VIRTUAL VIEW statement introduces a mapping between relational schema and the hierarchy of the ontology. Following a minimalist’s approach, we use the join syntax of SQL to express the mapping.

Definition 1 *Create a Virtual View*

```
CREATE VIRTUAL VIEW View(Column1, ..., ColumnN) AS
SELECT head1, ..., headN
FROM BaseTable AS T, Ontology AS O
WHERE constructor
    AND p1 AND ... AND pk
    AND m1 AND ... AND mj
```

According to the above definition, a virtual view is derived from a base table (or a set of base tables) and an ontology, which are specified in the FROM clause. If we regard an ontology hierarchy as a class hierarchy in an object-oriented programming language, the join operation can be regarded as using data from the base tables to instantiate specific ontological types. The integration between

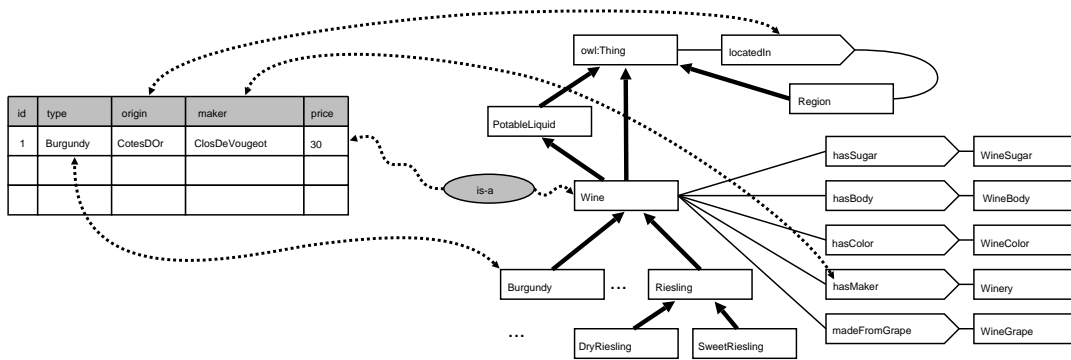


Figure 3: Creating a virtual view

the data and the ontology is specified by the WHERE clause, through its use of predicates of three different types.

(1) **Constructor.** Each CREATE VIRTUAL VIEW statement has one and only one constructor in the form of $O.type = expr$, which instantiate ontology instance of type $O.type$ for a record in the relational table. For example, the constructor $O.type = 'Wine'$ creates a Wine object, and as another example, the constructor $O.type = T.type$ creates an object whose type is specified by the type column in the base table.

(2) **Constraints:** p_1, \dots, p_k . Each p_i can be a traditional boolean predicate on the relational table T . For example, with $T.price \geq 30$ we exclude tuples whose price is less than 30 in the virtual view. Each p_i can also be an ontological constraint, which is a triplet in the form of $(Object_1, Relation, Object_2)$. For example, the constraint $(O.type \text{ isA } 'Wine')$ prescribes that the instance we have constructed must be of type Wine or a subtype of Wine, and $(O.type \text{ madeFromGrape } 'Barbera')$ prescribes that the wine instance we created must be made from grape Barbera. Note that the CREATE VIRTUAL VIEW statement is only responsible for expressing such constraints; the enforcing of such constraints may require knowledge inferencing, and is handled by rule rewriting at query time.

(3) **Mapping:** m_1, \dots, m_k . In the ontology, an instance can have many properties, for example, a wine may have such properties as price, color, origin, etc. The integration enables properties to take values from the relational data. To do this, we create a mapping between the schema of the base table and the properties in the ontology. For instance, $T.origin \rightarrow O.locatedIn$ maps the origin column of the base table to the locatedIn property. Note that a mapping is different from a constructor. A constructor associates a record in the relational table to an instance of a specific type in the ontology, while a mapping associates attributes of the record to properties of the instance created for the record.

We study an example of the CREATE VIRTUAL VIEW statement. In Example 6, the sources of the virtual view are the Wine table and the WineOntology. They are specified in the FROM clause. The predicates in the WHERE clause specify how the wine table and the wine ontology are integrated. The constructor $O.type=W.type$ instantiates an ontology instance whose type is given by $W.type$. Take the first tuple in the wine table as an example. The constructor $O.type='Burgundy'$ creates a Burgundy instance, which is a subtype of Wine in the ontology. The second line, $(O.type \text{ isA } 'Wine')$, prescribes that the newly created instance must be an instance of the Wine class. Thus, if the data in the wine table contains non-wine items, it will not be instantiated. The next two conditions specify that the origin column of the wine table corresponds to Burgundy's locatedIn attribute (which is inherited

from class owl:Thing), and the maker column corresponds to wine's hasMaker attribute. Note here that $O.hasMaker$ is only meaningful when O is an instance of the Wine class.

Example 6 To create a virtual view WineView for integrating the wine table and the wine ontology, the following SQL statement is invoked. After the virtual view is registered, users can issue queries such as Example 3 and Example 4 as if it were a relational table.

```
CREATE VIRTUAL VIEW WineView(
    Id, Type, Origin, Maker, Price,
    LocatedIn, HasColor) AS
SELECT W.*,
    O.locatedIn,
    O.hasColor
FROM Wine AS W, WineOntology AS O
WHERE O.type=W.type /*constructor*/
    AND (O.type isA 'Wine') /*constraint*/
    AND W.origin → O.locatedIn /*mapping*/
    AND W.maker → O.hasMaker /*mapping*/
```

The result of the CREATE VIRTUAL VIEW statement is a schema that includes two virtual columns: LocatedIn and HasColor. This is prescribed by the SELECT list, which has three items.

- Item $W.*$ indicates that the schema of the virtual view contains all the columns (Id, Type, Origin, Maker, Price) in the original wine table.
- Item $O.hasColor$ specifies a virtual column, which is based on the hasColor property of the wine object in the ontology. The attribute value is to be derived using implication rules during query time.
- Item $O.locatedIn$ specifies another virtual column. Note that unlike hasColor, the locatedIn property is transitive, which is indicated in the ontology. Thus, conceptually, the values returned by the SELECT will be the transitive closure of the locatedIn property, which is a set of locations that contain the region specified by $W.origin$.

Fig. 3 illustrates the functionality of the CREATE VIRTUAL VIEW statement. Clearly, the registration of the virtual view merely creates a mapping between values in a relational table and concepts in the ontology. This enables the system to perform knowledge inferencing for queries against the virtual view.

3. HYBRID RELATIONAL-XML DBMSS

The modeling described in Section 2 needs physical level support in a DBMS. In particular, the ontology is modeled as semistructured data, which traditional RDBMSs cannot handle directly.

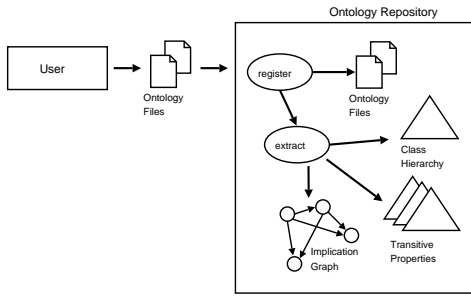


Figure 4: Ontology files are registered in the ontology repository before use and the ontology repository extracts several types of information from the ontology files including class hierarchies, implication rules, transitive properties etc.

We leverage *hybrid relational-XML DBMSs* for physical level support. Some commercial RDBMSs such as IBM DB2 9 PureXML [10, 20] now support XML in its native form. For concreteness, the examples in this paper will be based on IBM’s DB2.

In a hybrid relational-XML DBMS, XML is supported as a basic data type. Users can create a table with one or more XML type columns. A collection of XML documents can therefore be defined as a column in a table. For example, a user can create a table *ClassHierarchy* with the following statement:

```
CREATE TABLE ClassHierarchy
(id integer, name VARCHAR(27), hierarchy XML);
```

To insert an XML document into a table, it must be parsed, placed into the native XML storage, and then indexed. We use the SQL/X function, *XMLParse*, for this purpose:

```
insert into ClassHierarchy values(1, 'Wine',
XMLParse('<?xml version='1.0'>
<wine>
  <WhiteWine>
    <WhiteBurgundy> ... </WhiteBurgundy> ...
  </WhiteWine>
</wine>'));
```

Users can query relational columns and XML column together by issuing SQL/XML query [5, 6]. For example, the following query returns class ids and class names of all class hierarchies that contain the XPath */Wine/DessertWine/SweetRiesling*:

```
SELECT id, name
FROM ClassHierarchy AS C
WHERE XMLExists('$t/Wine/DessertWine/SweetRiesling'
PASSING BY REF C.order AS "t")
```

The SQL/XML [7] function *XMLExists* evaluates an XPath expression on an XML value. If XPath returns a nonempty sequence of nodes, then *XMLExists* is true, otherwise, it is false.

4. ONTOLOGY REPOSITORY

In order to support ontologies as first class citizens of the DBMS, we augment the DBMS with an *ontology repository*. An ontology repository consists of a collection of tables that store all the information associated with the ontologies registered by the users. In this section, we describe how users can manage ontologies with the ontology repository and how these ontologies are preprocessed internally to extract various information such as class hierarchies, transitive properties, implication graph from the ontology files. For the sake of concreteness, we use OWL ontologies for our discussion. Our framework, however, is not restricted to the OWL format.

Managing Ontology Files. From the user’s perspective, the ontology repository is a table of ontology files and their identifiers

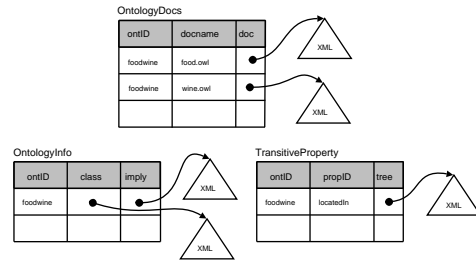


Figure 5: Internal schema of the ontology repository. We use text as identifiers for readability. The *OntologyDocs* table stores a copy of the original ontology files registered by the user. The *OntologyInfo* and *TransitiveProperty* tables store the information extracted from the ontology files for query processing.

(ontIDs). Besides being a storage system for ontology files, another important purpose of an ontology repository is to hide the complexity of ontology related processing from the user.

Our ontology repository provides a simple user interface. The user supplies a unique ontology identifier (ontID) to identify a logical ontology. Each logical ontology is usually encoded in several ontology files. The user registers each ontology file that is part of a logical ontology with a unique identifier (ID) via the stored procedure *registerOntology(ontid, ontology_File)*.

When an existing logical ontology in the repository needs to be removed, the stored procedure *dropOntology(ontid)* is called with the ontology ID. All the ontology files and extracted information associated with the specified ontology ID will be deleted.

Preprocessing Ontology Files. After the user has finished registering the ontology files, the ontology files associated with the same ontID are pre-processed in order to extract a variety of information that could be used in query processing (see Fig. 4). In particular, we highlight several pieces of key ontology information that are extracted and stored to facilitate query processing: the class hierarchies, the transitive properties, and the implication graph. These three pieces of key ontology information that we extract are organized around three tables, *OntologyDocs*, *OntologyInfo*, *TransitiveProperty*, in the ontology repository (see Fig. 5). In an actual system, more ontology information may be extracted: some to support specific query types, others for optimizing query processing.

Conceptually, the three types of information that we extract correspond to three types of rules encoded in the ontology. In particular, all three types of rules are *Horn* rules (defined next).

A *Horn rule or clause* is a logic expression of the form

$$H \leftarrow A_1 \wedge \dots \wedge A_m \wedge \sim A_{m+1} \wedge \dots \wedge \sim A_n$$

where H, A_i are atoms or atomic formulae, and $n \geq m \geq 0$. H is called the *head* (or consequent) of the rule and the right-hand-side (RHS) of \leftarrow is called the *body* (or antecedent) of the rule. The operator \leftarrow is to be read as “if” and \sim stands for negation-as-failure. Each rule is implicitly viewed as universally quantified. A *definite Horn rule* is a Horn rule where the RHS does not contain any negation¹. The implication rules that we consider in this paper are acyclic Horn rules without negations.

Atoms or atomic formulae are represented in two ways in this paper. For example, the atom representing the predicate “wine X

¹In the case of Datalog [24], definite Horn rules are often further restricted to non-recursive rules and “safe” rules where all variables that occur in the head also occur in the body.

has color red” can be written in the following two ways,

$hasColor(X, red)$ or $(hasColor=red)$,

where variable identifiers begin with capital letters, and constants begin with small letters. The first representation is used in the context of logical inference: atoms are represented as logical functions with any arity. The latter representation is used in the context of SQL predicates in a SQL where-clause: atomic formulas are more conveniently written as attribute-operator-value expressions or triples. In the above example, the operator = denote an equality test and the implicit object is a row of the base table associated with the SQL query.

Providing the mapping of the entire OWL syntax into the three types of rules considered in this paper is beyond the scope and space limitations of this paper. Instead we provide a few examples to illustrate the mapping.

Class Hierarchies. The class hierarchies that we extract from the ontology corresponds to subsumption rules dealing with the special *subClassOf* relationship,

$subClassOf(A, C) \leftarrow subClassOf(A, B) \wedge subClassOf(B, C)$

and *isA* relationship,

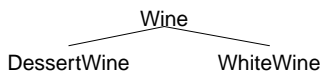
$isA(B, X) \leftarrow isA(A, X) \wedge subClassOf(A, B)$.

The *subClassOf* relationship relates two classes. The *isA* relationship relates an instance to its class. Note that the *subClassOf* and *isA* relationships are special “builtin” relationship that not defined by the ontology-author.

For OWL ontologies, these *subClassOf* relationships that define the class hierarchy can be expressed in several ways. If strict tree structure is required for persistence, non-disjoint *subClassOf* relationships can be flattened into tree structure. In most cases, *subClassOf* relationships are explicitly specified in a *subClassOf* construct and in some cases via restrictions. For example,

```
<owl:Class rdf:ID="DessertWine">
  <rdfs:subClassOf rdf:resource="#Wine" />
  ...
</owl:Class>
...
<owl:Class rdf:ID="WhiteWine">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Wine" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasColor" />
      <owl:hasValue rdf:resource="#White" />
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

where the WhiteWine class is defined to be all wines whose hasColor attribute has the value white. The class hierarchy for the above OWL fragments is:



Transitive Properties. Transitive properties corresponds to subsumption rules dealing with transitive relationships defined in the ontology by the ontology-author. For example, the *locatedIn* property in the wine ontology corresponds to the following rule,

$locatedIn(A, C) \leftarrow locatedIn(A, B) \wedge locatedIn(B, C)$.

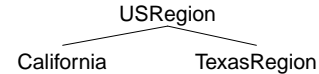
The facts associated with these transitive relationships can be extracted from the ontology into a tree representation to facilitate query re-writing and processing. In OWL, transitive binary relationships (owl:ObjectProperty) are specified using the following construct:

```
<owl:ObjectProperty rdf:ID="locatedIn">
  <rdf:type rdf:resource
    ="&owl;TransitiveProperty" />
  ...
</owl:ObjectProperty>
```

Once we know that the *locatedIn* property is transitive, we scan for all the instances of the property and construct a tree (or forest) from them. For example, suppose the following instances of the *locatedIn* property are found,

```
<Region rdf:ID="USRegion" />
<Region rdf:ID="CaliforniaRegion">
  <locatedIn rdf:resource="#USRegion" />
</Region>
<Region rdf:ID="TexasRegion">
  <locatedIn rdf:resource="#USRegion" />
</Region>
```

The following transitive tree is constructed.



Implication Rules. Both the class hierarchies and the transitive properties are a type of recursive rules. The implication graph, on the other hand, captures non-recursive rules encoded in the ontology. These non-recursive rules are represented internally as an *implication graph*.

An *implication graph* G is a directed acyclic graph consisting of two types of vertices and two types of edges. The vertex and edge set of G is denoted by $V(G)$ and $E(G)$ respectively. The set of nodes adjacent to a given vertex v is defined as $Adj(v) = \{u | (v, u) \in E(G)\}$. An implication graph has two types of nodes. Predicate nodes $P(G)$ are associated with atoms in Horn clauses. Conjunction nodes $C(G)$ represent the conjunction of two or more atoms in the body of a Horn clause. For a vertex $v \in P(G)$, the predicate name (object property name) associated with v is denoted by $pred(v)$, the predicate value by $val(v)$, the operator that relates the predicate name to the predicate value by $op(v)$.

For example, Figure 6 shows the implication graph for the following set of implication rules:

$A=v1 \leftarrow G=v7$
 $A=v1 \leftarrow B=v2 \wedge C=v3$
 $B=v2 \leftarrow H=v8$
 $C=v5 \leftarrow D=v4$
 $C=v5 \leftarrow F=v6$

The construction of the implication graph for an ontology is straightforward. We start with an empty implication graph and scan the ontology files for all implications. After filtering out recursive implications, such as those associated with class hierarchies and transitive properties, we are left with the non-recursive implications. We iterate through each non-recursive implication and insert vertices and edges into the implication graph.

For OWL ontologies, standard logical equivalences can be used to convert definitions into implication rules. Complex implications whose consequent is a conjunction of atoms can in most cases be decomposed into Horn rules. Consider the following OWL fragment from the definition of the Zinfandel class:

```
<owl:Class rdf:about="#Zinfandel">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasColor" />
      <owl:hasValue rdf:resource="#Red" />
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
```

```

    <owl:onProperty rdf:resource="#hasSugar" />
    <owl:hasValue rdf:resource="#Dry" />
  </owl:Restriction>
</rdfs:subClassOf>
...
</owl:Class>

```

The OWL fragment specifies that all instances of the Zinfandel class must also belong to the sub-class of all wines whose hasColor property takes the value red and to the sub-class of all wines whose hasSugar property takes the value dry, i.e.,

$$(isA(Zinfandel, X) \rightarrow [(hasColor(X, Red) \wedge (hasSugar(X, Dry))]$$

which can be decomposed into a collection of Horn rules (the proof using a truth table is trivial):

$$[isA(Zinfandel, X) \rightarrow hasColor(X, Red)] \\ \wedge [isA(Zinfandel, X) \rightarrow hasSugar(X, Dry)].$$

Encoding Extracted Information in XML. After the class hierarchy, transitive properties, and implication graph are extracted from the ontology, they are serialized into XML and stored in the ontology repository.

The class hierarchy and transitive properties all contain subsumption relationships in a tree data structure. Because our query processing component will be relying on XPath for subsumption checking, these tree data needs to be serialized into XML in a way that preserves their tree structure in XML. For example the transitive tree shown previously can be encoded into XML as

```

<USRegion>
  <California/>
  <TexasRegion/>
</USRegion>

```

On the other hand, there is much more flexibility for serializing the implication graph, because we do not need any kind of subsumption testing on it at all. Any standard method for encoding graphs to XML can be used.

5. QUERY PROCESSING

In this section, we describe how queries written against the virtual view can be processed by re-writing into equivalent queries that run on both the base table and the information in the ontology. When describing algorithms in a formal setting, we will use R to denote the set of recursive relationships, where each relationship is either a class hierarchy or a transitive property in the ontology. From the logic inferencing point of view, the class hierarchies and the transitive relationships are both a type of recursive rule. We will also use a set of *view triples* to refer to the mapping information in the create-virtual-view statement. These view triple information will typically be stored in the catalog tables of a DBMS. In general a view triple (b, r, v) encodes a binary association between any pair of a base table b , a property or relationship r in the ontology, and a column v in the virtual view².

Consider a SQL query with a WHERE-clause that consists of conjunctions and disjunctions of atomic predicates. The conjunction and disjunction operators need no re-writing. The atomic predicates in the WHERE-clause can be re-written independently, because (1) predicates on view columns that are not associated with

²For example, the view triples associated with Example 6 are

relational view triples	$(W.id, \epsilon, V.Id),$ $(W.type, \epsilon, V.Type),$ $(W.origin, \epsilon, V.Origin),$ $(W.maker, \epsilon, V.Maker),$ $(W.price, \epsilon, V.Price),$
virtual column triples	$(\epsilon, O.locatedIn, V.LocatedIn),$ $(\epsilon, O.hasColor, V.HasColor),$
ontology triples	$(W.type, O.type, \epsilon),$ $(W.origin, O.locatedIn, \epsilon),$ $(W.maker, O.hasMaker, \epsilon).$

an ontology need only a renaming of the view column name to the column name in the base table, and (2) predicates on virtual columns need to be re-written using rules that are restricted to definite Horn rules in our system.

Algorithm 1 outlines the rewriting algorithm for a WHERE-clause Q in a SQL query on a virtual view. The algorithm takes as input the set of atoms from the WHERE-clause Q , the implication graph G , the set of recursive relationships R (predicate identifiers of all transitive OWL properties and class hierarchies), and the virtual view definition \mathcal{V} , and outputs a rewritten query expression Q' . The algorithm loops through each atom in Q and rewrites each atom independently. Each atom is viewed as a column-operator-value triple. The getViewTriple procedure retrieves from the DBMS catalog tables the view triple associated with the column in the atom. If the column in the atom is not a virtual column, the atom is rewritten using the base table column from the view triple. Otherwise, we call the EXPAND procedure to expand the atom. If the EXPAND procedure returns an empty result, there is no rule that could satisfy the atom and we rewrite the atom to 'false'.

Algorithm 1 REWRITE(Q, G, R, \mathcal{V})

Input: Q is a set of atomic predicates, G is the implication graph, R is the set of recursive implications, \mathcal{V} is the view definition

Output: Q' is the set of expanded predicate expression

```

1: Let  $Q = \{A_1, A_2, A_3, \dots\}$ 
2:  $Q' \leftarrow \emptyset$ 
3: for all  $A_i \in Q$  do
4:   Let  $A_i = (vcol, op, value)$ 
5:    $(b, r, vcol) \leftarrow getViewTriple(\mathcal{V}, vcol)$ 
6:   if  $r = \epsilon$  then
7:     /* vcol is not a virtual column */
8:      $Q' \leftarrow Q' \cup \{(b, op, value)\}$ 
9:   else
10:    /* vcol is a virtual column */
11:     $a \leftarrow findRuleNode(G, R, (r, op, value))$ 
12:    if  $a$  not found then
13:       $Q' \leftarrow Q' \cup \{false\}$ 
14:    else
15:       $A'_i \leftarrow EXPAND(a, G, R, \mathcal{V})$ 
16:      if  $A'_i = \epsilon$  then
17:        /* if rewritten predicate is empty */
18:         $Q' \leftarrow Q' \cup \{false\}$ 
19:      else
20:         $Q' \leftarrow Q' \cup A'_i$ 
21: return  $Q'$ 

```

Note that REWRITE(Q, G, R, \mathcal{V}) only rewrites the predicate expression in the WHERE-clause of a SQL query. Additional post processing is required to add in the retrieval operations for the ontology information needed by the rewritten predicates. For example, if the rewritten WHERE-clause consists of the subsumption-check operator ISSUBSUMED('USRegion', 'locatedIn', Wine.Origin), postprocessing will need to add in the appropriate arguments to the FROM-clause and the WHERE-clause to retrieve the transitive property 'locatedIn' from the ontology repository.

For hybrid relational-XML DBMS, a straight-forward implementation of the ISSUBSUMED boolean operator is to use the SQL/XML function XMLExists [7, 14]. Another possibly less efficient implementation is to use a recursive SQL statement as alluded to in Das et al [4]. For the rest of the discussion, we will assume that the ISSUBSUMED boolean operator can be implemented by re-writing to the SQL/XML XMLExists function.

The heavy-lifting in the inferencing work is actually performed in the EXPAND procedure outlined in Algorithm 2. Predicate expansion work that is similar in spirit has been done in [21] for a different type of rules, but our algorithm is original in the way it deals

with virtual columns and recursive rules. The EXPAND procedure performs inferencing by exploring the implication graph and any relevant recursive relationships. To elaborate on the algorithm, we first define some required concepts.

Algorithm 2 EXPAND(h, G, R, \mathcal{V})

Input: h is the node in implication graph G to be expanded, R is the set of recursive relationships, and \mathcal{V} is the virtual view definition

Output: e is the expanded predicate expression

```

1: if  $h$  is a ground node then
2:    $(b, r, v) \leftarrow \text{getViewTriple}(\mathcal{V}, \text{pred}(h))$ 
3:    $e \leftarrow \{(b, \text{op}(h), \text{val}(h))\}$ 
4:   if  $h$  is a recursive node then
5:      $e \leftarrow e \vee \text{ISSUBSUMED}(\text{val}(h), \text{pred}(h), b)$ 
6:   /* R-Expansion */
7:   if  $h$  is a recursive node then
8:     for all  $s \in \text{subsumedAtoms}(R, h)$  do
9:       if  $s \in P(G)$  then
10:        for all  $\text{rulebody} \in \text{dependentExp}(s, G)$  do
11:           $\text{tmp} \leftarrow \emptyset$ 
12:          for all  $i \in \text{rulebody}$  do
13:             $\text{tmp} \leftarrow \text{tmp} \wedge \text{EXPAND}(i, G, R, \mathcal{V})$ 
14:           $e \leftarrow e \vee \text{tmp}$ 
15:        /* G-Expansion */
16:      for all  $\text{rulebody} \in \text{dependentExp}(h, G)$  do
17:         $\text{tmp} \leftarrow \emptyset$ 
18:        for all  $i \in \text{rulebody}$  do
19:           $\text{tmp} \leftarrow \text{tmp} \wedge \text{EXPAND}(i, G, R, \mathcal{V})$ 
20:         $e \leftarrow e \vee \text{tmp}$ 
21: return  $e$ 

```

Definition 2 (Recursive nodes) A predicate node $n \in P(G)$ from the implication graph G is a recursive node if and only if $\text{pred}(n) \in R$, where R is the set of predicate identifiers of all recursive relationships.

Definition 3 (Ground nodes) For a given virtual view definition \mathcal{V} , a predicate node $n \in P(G)$ from the implication graph G is a ground node if and only if there exists some view triple $(b, r, v) \in \mathcal{V}$ such that $r = \text{pred}(n)$ and $b \neq \epsilon$, i.e., the predicate is associated with a base table column in the virtual view definition.

The EXPAND procedure works as follows. Given a predicate node h , if h is a ground node (line 1), it means that h is associated with a base table column and the predicate h can be checked against the base table column directly. If in addition to being a ground node, the node h is also recursive, then an additional subsumption check needs to be added to the rewritten predicate. An example of a non-recursive ground node for the virtual view in Figure 3 would be “hasMaker=ClosDeVougeot”, and an example of a recursive ground node would be “locatedIn=USRegion”.

For the case where h is not a ground node, it is clear that the algorithm needs to traverse the implication graph. For the case where h is a ground node, the algorithm still needs to continue traversing the implication graph so as to ensure completeness of the inferencing. A ground node is an atom that can be checked against the base table, but does not ensure that the atom is true against the base table; hence, the expansion cannot stop at ground nodes unless there are no more ground nodes reachable from the current ground node.

To further traverse the graph, recursion is used (note that we expressed the traversal using recursion for clarity, a stack can be used for a non-recursive implementation). If h is recursive, then all atoms subsumed by h , denoted by $\text{subsumedAtoms}(R, h)$, will also satisfy the predicate h . The $\text{subsumedAtoms}(R, h)$ function is computed by retrieving the tree associated with h from R (either a class hierarchy or a transitive property) and finding in the tree the

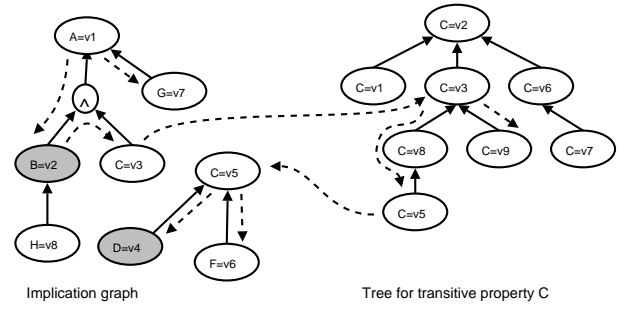


Figure 6: The implication graph and the tree for the transitive relationship C used in Example 7. The two shaded nodes in the implication graph denote ground nodes. Dotted lines indicate traversal of the EXPAND algorithm.

(possibly empty) set of all the atoms subsumed by h . We then recursively call EXPAND on each atom in the subsumed set that has a node in G . Such expansions via the recursive rules are called R-expansions (line 6-7).

In addition to handling expansion via recursive rules, we need to expand h with non-recursive rules contained in the implication graph G , i.e. G-expansions (line 15-16). We iterate over $\text{dependentExp}(h, G)$, the (possibly empty) set of all rules in G that has h as the head. Each element (rulebody) of $\text{dependentExp}(h, G)$ represents the body of a rule and consists of a set of atoms (implicitly joined by conjunction). EXPAND is called on each of these atoms. The re-written expressions of the atoms in a single rule body are joined with a conjunction, and the re-written expressions of different rules are joined with a disjunction in accordance of the semantics of Horn rules.

The EXPAND procedure always terminates because there are no cycles in the implication graph and the transitive trees (by definition).

Theorem 1 With respect to the fragment of horn rules that we support, the view definition, and the query types that are supported, our rewriting procedure is sound and complete.

Proof sketch. The EXPAND procedure rewrites an atom either by traversing paths within the implication graph or by traversing paths in the trees used to store the transitive properties and class hierarchies. Each outgoing edge of a predicate node in the implication graph G denotes a logical implication and each conjunction node is processed without violating the semantics. Each path in the trees denotes recursive application of the transitive rule, $\text{pred}(A, C) \leftarrow \text{pred}(A, B) \wedge \text{pred}(B, C)$. Since each step is an application of some implication rule, the rewriting is sound. For proving completeness, note that our data structures encode each unique atom exactly once. If there is any path from the query predicate to a ground node, our algorithm will discover it. Since our rewriting algorithm examines all rules that could be satisfiable, the rewriting is complete.

Example 7 Consider the following SQL query on the virtual view WineView($id, hasColor$)

```

SELECT v.Id
FROM WineView AS v
WHERE v.hasColor=v1;

```

where the virtual view definition consists of the following triples,

$\{(id, \epsilon, id), (\epsilon, A, hasColor), (type, B, \epsilon), (origin, D, \epsilon)\}$

Further suppose that the implication graph G and the recursive tree for transitive property $C \in R$ are as shown in Figure 6. Rewriting the query using y line 20 is executed, because the query predicate does not involve recursive relationships and there exists a rule in G for the query predicate. Algorithm 1 calls the EXPAND procedure (Algorithm 2) to expand the query predicate. Only $B=v2$ and $D=v4$ are ground nodes in G , so EXPAND (Algorithm 2) tries to traverse G and the tree for C towards the ground nodes. In this case satisfying paths are found and the re-written query is:

```
SELECT W.id
FROM Wine AS W
WHERE W.type=v2 AND W.origin=v4;
```

Optimization. Algorithm 2 has two sources of complexity : expansion via each dependent rule body from the set $dependentExp(G, h)$, and expansion via the recursive relationships from $subsumedAtoms(R, h)$ (G-expansions and R-expansion respectively). Many of these expansions can be avoided if we know that the atoms do not lead to any ground or recursive nodes. This section discusses several ideas for pruning the expansion.

The notion of live or dead nodes (defined next) captures the intuition for whether a node can ever be satisfied from some ground nodes downstream in the inference process.

Definition 4 (Live and dead nodes) For a given virtual view definition, a node $n \in V(G)$ from the implication graph G is a live node if

- (1) $n \in P(G)$ and n is a ground node or a recursive node, or
- (2) $n \in C(G)$ and $\forall v \in Adj(u), v$ is a live node, or
- (3) there exists some $u \in Adj(n)$ such that u is a live node.

Conversely, a node n that is not a live node is called a dead node.

The first optimization is that we mark nodes in the implication graph G that are dead because there is no path from those nodes to any recursive or ground nodes. Note that whether a node is alive or dead is dependent on the view definition. It is clear that if a node does not contain any ground nodes downstream, the expansion algorithm can safely skip it. If a recursive node exists downstream, the algorithm still need to expand to the recursive node and process the recursive node, because the indirect nodes in the recursive relationship tree can trigger G-expansions.

The second optimization deals with the atoms within a rule body, i.e., a conjunction node in the implication graph. The expansion of the atoms in a rule body can be safely skipped if at least one of the atoms is dead. This pruning criteria will still preserve soundness, because atoms in a rule body are joined by logical conjunction that requires every atom to be true.

The third optimization applies to R-expansions. To prune the number of expansions due to R-expansion, we can either mark nodes in the recursive tree R that are not associated with live nodes in G , or we can check if the subsumed atoms are live before calling EXPAND recursively. The former is more efficient because the pruning is done earlier when $subsumedAtoms(h, R)$ is computed resulting in a much smaller set of subsumed atoms.

The fourth optimization uses memoization techniques to avoid traversing nodes in the implication graph more than once.

The fifth optimization deals with pre-computation of the predicate re-writing. If the set of values associated with a virtual column (eg. the hasColor property in the wine ontology) is small, then we can pre-compute the re-writing for each possible value predicate on the virtual column and store these re-written predicates with the view definition in the system catalog tables. During query processing, the system catalog will be consulted first to determine if pre-computed rewriting exists before calling our rewriting procedures.

Note that our optimization strategies on the expansion algorithm

are only dependent on the ontology and the schema of the base table. Updates to the base table data after the creation of view will not affect the correctness of the optimization strategies.

6. EXPERIMENTS

The usefulness of our virtual view framework will depend in part on the performance of the queries against the virtual columns in the virtual view. We have showed that such queries can be re-written into plain SQL/XML queries on the base table and the ontology data in the ontology repository using the re-writing algorithms in Section 5. The performance of the re-written queries are independent of the algorithms proposed in this paper and entirely dependent on the data and the DBMS engine. The performance of commercial DBMS engine is not the focus of this paper; hence we focus on the performance of the re-writing algorithms in this section.

We prototyped our query re-writing algorithms in C++ and measured its performance over synthetic implication graphs and trees. The choice of synthetic data is intentional so as to investigate the performance of our rewriting algorithms under different data sets with different characteristics. With real publicly available ontologies, we would only be able to show single performance numbers that would not shed light on how the rewriting algorithms scale with the complexity of the data.

Data Generation. Random implication graphs are generated by specifying the number of relationships $nvar$, the number of values $nval$ each relationship can take, the depth $nlevels$ of the implication graph, the maximum number of rules $density$ to generate between consecutive levels in the graph and the maximum number of atoms $fanout$ in a rule body. For our experiments the number of values $nval$ is fixed at 10. The maximum number of atoms or nodes in the graph is $nvar \times nval$. The atoms are partitioned uniformly into $nlevels$ groups. The groups are randomly ordered and some number of rules are generated for each consecutive pair of groups. The number of rules $nrule$ generated between two consecutive groups is randomly chosen between one and $density$. Each rule is generated as follows. Randomly pick one atom from group g as the head. Randomly choose f the number of atoms in the body between one and $fanout$. Randomly pick f atoms from group $g + 1$ for the rule body.

Generating tree data for transitive relationships and class hierarchies is somewhat simpler. Each generated tree is specified by the number of values $nval$ and the maximum fanout $fanout$. The number of atoms or nodes in the tree is the same as the number of values, because each tree is associated with one relationship. The generation procedure uses a randomized stack that initially contains only the root node. At each iteration, a node is popped from a random position in the stack for expansion. A random number of children nodes are generated subject to the specified maximum and pushed onto the stack. The procedure terminates when the tree contains the required number of nodes. the stack.

Measuring performance. We measured the performance of the BASELINE algorithm (Algorithm 1 and Algorithm 2) and the OPTIMIZED algorithm, in which memoization (the fourth optimization described in Section 5) is used to optimize the EXPAND procedure. Each run consists of generating an implication graph, generating zero or more trees, measuring the average time to rewrite a query in a workload consisting of all the head atoms of the rules in the implication graph. Such a workload ensures that the rewriting performance on all parts of the implication graph are measured. For each implemented algorithm, and for each setting of the parameters, the performance is averaged over five runs, i.e., five random data sets, to eliminate fluctuations due to randomness. The performance measure is therefore the time to rewrite a single atom or

predicate averaged over the atoms in the implication graph and five random runs.

Varying number of relationships in the implication graph. Figure 7(a) show the rewriting performance as the number of relationships, i.e. *nvar*, in the generated implication graphs is varied. The number of groups is fixed at 5, the *density* at 1600, and the number of trees is zero. Observe that OPTIMIZED is significantly more efficient when the number of relationships is small and the performance of BASELINE approaches that of OPTIMIZED as the number of relationships becomes large. The reason for this unexpected result is because increasing the number of nodes while *density* is fixed, increases the sparsity of the implication graphs. A sparse implication graph less expansion in our rewriting algorithm hence the performance improvement.

Varying the *density* of rules. To confirm the above-mentioned intuition, we fixed the number of relationships to 100 and the number of groups to five, and varied the maximum number of rules generated between consecutive groups. Figure 7(b) and Figure 7(c) show the average rewriting time when 16 trees and zero trees are associated with the implication graph respectively. Observe that as the *density* increases, the performance of BASELINE degrades exponentially, whereas OPTIMIZED scales almost linearly. As the implication graph becomes more densely connected, the opportunity for exploiting duplicate expansions increases; hence the superior performance of OPTIMIZED. Comparing Figure 7(c) and Figure 7(b), we also observe that inferring via the transitive relationships adds an order of magnitude to the rewriting time; however, OPTIMIZED scales very reasonably in both cases.

Varying the tree sizes. To further understand how the number of trees and the size of the trees affect the rewriting time, we fixed the implication graph and varied the size of the 16 trees associated with it. We found that varying the number of trees has an effect very similar to varying the size of the trees, so only one set of results will be presented. Figures 7(d) and 7(e) show the results for two different *density* values. Observe that both algorithms scale linearly with the tree size. The running time of OPTIMIZED grows more slowly with tree size compared to BASELINE. The superior performance of OPTIMIZED is especially dramatic for denser implication graphs.

Varying the depth of the implication graph. In general, we do not expect real ontologies to have implication graphs with a large number of levels. Nevertheless, we investigated how the number of groups of levels in the implication graph affects the rewriting performance by varying the number of groups from three to eleven. Figure 7(f) shows the performance. BASELINE is significantly more sensitive to the number of levels: increasing the number of levels could increase the search space for the expansion exponentially in the number of rules. OPTIMIZED uses memoization to avoid this exponential explosion: it never expands a rule more than once per query.

7. RELATED WORK

Several tools have been developed for building and manipulating ontologies. For example, Protégé is an ontology editor and a knowledge-base editor that allows the user to construct a domain ontology, customize data entry forms, and enter data [23]. RStar is an RDF storage and query system for enterprise resource management [15]. Other ontology building systems include OntoEdit [17], OntoBroker [16], OntologyBuilder and OntologyServer [3], and KAON [22]. Most systems use a file system to store ontology data (e.g., OntoEdit). Others (e.g., RStar and KAON) allow the ontology data to be stored in a relational DBMS. However, processing of ontology-related queries in these systems is typically done by

an external middle-ware (wrapper) layer built on top of a DBMS engine. Two key limitations of this loosely-coupled approach are: (1) DBMS users cannot reference ontology data directly, and (2) query processing of ontology-related queries cannot leverage the query processing and optimization power of a DBMS.

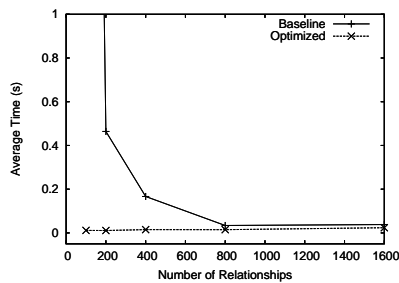
Description logic (DL) and Datalog systems have also been well-studied [8, 9]. These systems are based on translating a subset of DL to Datalog so that efficient Datalog inferencing engines can be used. Efficient Datalog inferencing algorithms have been thoroughly investigated by [24]. Our framework differs from these efforts in that we focus on the integration of relational data and domain knowledge within the DBMS engine; expressivity of the logic fragment is not our focus, even though Datalog optimization techniques can be adapted for our rewriting algorithm. Our rewriting algorithm also differs from previous work on semantic query optimization [13] in that our focus is not on integrity constraints, but on rewriting queries on virtual view into queries on the base tables and the ontology information.

A recent advance in ontology management in DBMSs was introduced by Oracle. Das et al. [4] proposed a method to support ontology-based semantic matching in RDBMS using SQL directly. Ontology data are pre-processed and stored in a set of system-defined tables. Several special operators and a new indexing scheme are introduced. A database user can thus reference the ontology data directly using the new operators. Compared to the loosely-coupled approach, this method opens up the possibility of combining ontology query operators with existing SQL operators such as joins. The ability to manipulate ontology data and regular relational data directly in the DBMS greatly simplifies and facilitates the development of ontology-driven applications.

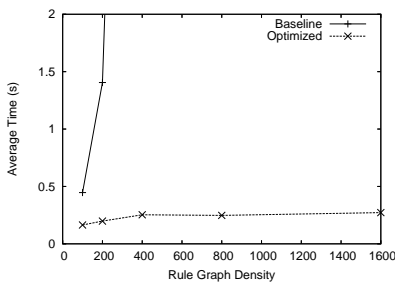
However, due to the “mismatch” between the relational schema and the graphical model of ontology data, this relational-model based approach is still quite limited in its expressing and processing power. From the expressivity aspect, an ontology can encode a broad spectrum of semantics over the base data. The semantics can range from a simple nickname for a value in the base table to some derived values obtained through very complicated reasoning, and the semantic matching operation studied in [4] is just one instance of such semantics. From processing aspect, inference is one of the most expensive operations on ontology data. All the previous approaches except [14] need to pre-compute and materialize all (or a big part of) the inference results (i.e., transitive closures) to achieve reasonable performance at query execution time. This pre-processing not only incurs serious time and storage overhead, but also makes the update of the pre-computed data infeasible when the underlying ontology data change. As an alternative, the authors of [14] proposed using XML trees to encode subsumption relationships and using the `XMLExists` SQL/XML [7] operator to perform subsumption checking. Our framework leverages on the technique in [14] for subsumption checking.

From a theoretical point of view, our framework can be classified as a type of global-as-view (GAV)[12] algorithm. However, our framework has two interesting features: the mapping between global schema (virtual view) and the local schema (base table) is completely determined only at query time, and the mapping is dependent on the data value that is being constrained by the query. Only when the query is specified, does our rewriting algorithm search the implications in the ontology in order to determine if a mapping exists and if it exists, to compute the mapping.

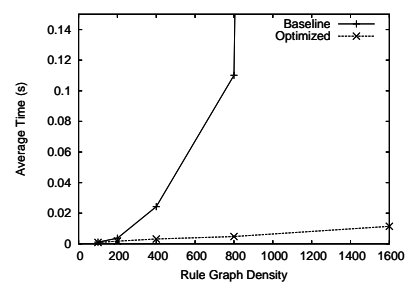
Our framework sets the basis for querying a variety of semantics over the relational data through a simple relational view. We leverage a relational-XML DBMS for manipulating ontology data and for processing semantic queries all within the DBMS engine.



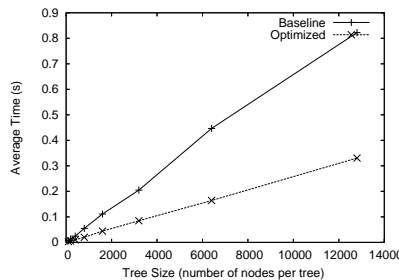
(a) Average rewriting time versus number of relationships in the implication graph.



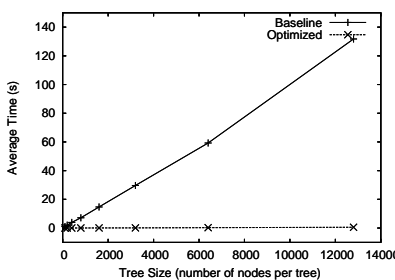
(b) Average rewriting time versus density. Number of trees = 16.



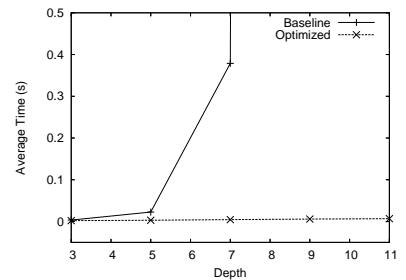
(c) Average rewriting time versus density. Number of trees = 0.



(d) Average rewriting time versus the number of nodes in each tree. Rule density = 100.



(e) Average rewriting time versus the number of nodes in each tree. Rule density = 800.



(f) Average rewriting time versus the number of groups in the implication graph.

Figure 7: The average rewriting time over different configurations of the implication graph and the transitive trees. The plots for BASELINE have been truncated where the running time is prohibitively long.

8. CONCLUSION

In this paper, we propose a framework that aims at supporting a rich class of semantic-related queries within DBMSs in an easy-to-express and potentially efficient-to-process manner. We provide a unified view of data and knowledge so that they can be queried using relational operators. Our framework leverages the recent development in ontology languages and native XML support in DBMSs: the former lays the foundation for semantic representation, and the latter makes it possible to manipulate data semantics with relational data in a DBMS. Our framework opens up new avenues towards supporting efficient semantic data management in DBMSs.

9. REFERENCES

- [1] D. Brickley and R. Guha. Resource description framework schema (RDF/S) specification 1.0, March 2000. W3C Recommendation <http://www.w3.org/TR/rdf-schema>.
- [2] DAML+OIL, March 2001. <http://www.daml.org/2001/03/daml+oil-index>.
- [3] A. Das, W. Wu, and D. L. McGuinness. Industrial strength ontology management. In *The Emerging Semantic Web*, 2001.
- [4] S. Das, E. I. Chong, G. Eadon, and J. Srinivasan. Supporting ontology-based semantic matching in RDBMS. In *VLDB*, 2004.
- [5] A. Eisenberg and J. Melton. SQL/XML is making good progress. *SIGMOD Record*, 31(2):101–108, 2002.
- [6] A. Eisenberg and J. Melton. Advancements in SQL/XML. *SIGMOD Record*, 33(3):79–86, 2004.
- [7] J. E. Funderburk, S. Malaika, and B. Reinwald. XML programming with SQL/XML and XQuery. *IBM Systems Journal*, 41(4), 2002.
- [8] B. N. Groszof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: combining logic programs with description logic. In *WWW*, pages 48–57, New York, NY, USA, 2003. ACM Press.
- [9] U. Hustadt, B. Motik, and U. Sattler. Reducing SHIQ-description logic to disjunctive datalog programs. In *KR*, pages 152–162, 2004.
- [10] IBM DB2 Universal Database viper release. <http://www.ibm.com/db2/udb/viper>.
- [11] O. Lassila and R. Swick. Resource description framework (rdf) model and syntax specification, February 1999. W3C Candidate Recommendation <http://www.w3.org/TR/REC-rdf-syntax>.
- [12] M. Lenzerini. Data integration: a theoretical perspective. In *PODS*, pages 233–246. ACM Press, 2002.
- [13] A. Y. Levy and Y. Sagiv. Semantic query optimization in datalog programs (extended abstract). In *PODS*, pages 163–173. ACM Press, 1995.
- [14] M. W. Lipyeow Lim, Haixun Wang. Semantic data management: Towards querying data with their meaning. In *ICDE*, page preprint, April 2007.
- [15] L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. RStar: An RDF storage and query system for enterprise resource management. In *CIKM*, 2004.
- [16] OntoBroker. http://ontobroker.aifb.uni-karlsruhe.de/index_ob.html.
- [17] OTK tool repository: Ontoedit. <http://www.ontoknowledge.org/tools/ontoedit.shtml>.
- [18] OWL web ontology language guide, February 2004. <http://www.w3.org/TR/owl-guide/>.
- [19] OWL web ontology language. <http://www.w3.org/TR/owl-ref/>.
- [20] F. Ozcan, R. Cochrane, H. Pirahesh, J. Kleewein, K. Beyer, V. Josifovski, and C. Zhang. System RX: One part relational, one part XML. In *SIGMOD*, 2005.
- [21] M.-C. Rousset. Backward reasoning in aboxes for query answering. In *KRDB*, pages 50–54, 1999.
- [22] The Karlsruhe Ontology and semantic web tool suite. <http://kaon.semanticweb.org/>.
- [23] The protege ontology editor and knowledge acquisition system. <http://protege.stanford.edu/>.
- [24] J. D. Ullman. *Principles of Database and Knowledge-base Systems*, volume 1. Computer Science Press, 1988.
- [25] Wine ontology. <http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine.rdf>.