Kragujevac J. Math. 30 (2007) 181–199.

NATIVE XML DATABASES vs. RELATIONAL DATABASES IN DEALING WITH XML DOCUMENTS

Gordana Pavlović-Lažetić

Faculty of Mathematics, University of Belgrade, 11000 Belgrade, Serbia
(e-mail: gordana@matf.bg.ac.yu)

(Received October 30, 2006)

Abstract. When dealing with data-centric XML documents, it is possible to convert XML documents into a relational database, which can then be queried using SQL. Such relational databases are called *XML-enabled databases*. On the other hand, the best choice for storing, updating and retrieving document-centric XML documents is usually a *native XML database (NXD)*. NXDs store XML documents as logical units, and retrieve documents using specific query languages such as XPath or XQuery.

This paper presents different approaches to accessing XML documents from relational databases, as well as from native XML databases. They will be compared based on how general they are in dealing with different types of XML documents and how expressive in stating requests for data, especially recursive queries. Two examples of different types of XML documents are presented. The first one is a part explosion problem as a data-centric example. The second one is a large, highly hierarchical XML document - Serbian language wordnet, a lexical-semantic network, as a document-centric example.

1. INTRODUCTION

XML (eXtensible Markup Language) has been designed as a markup language and a textual file format. It provides for a description of a document's contents, with non-predefined tags, and does not provide for any presentational characteristics.

The following is an example of XML-tagged document (an excerpt from a restaurant menu), contained in the file `simple.xml`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE menu (ViewSourceforfulldoctype...) >
<menu date='5.10.2006'>
    <food>
        <name>Homemade Bean Soup </name>
        <price>100.00 </price>
        <description>
            a bowl of white bean soup with paper and onion
        </description>
        <calories>650 </calories>
    </food>
    <food>
        <name>French Toast </name>
        <price>60.50din </price>
        <description>
            thick slices made from our homemade bread
        </description>
        <calories>300 </calories>
    </food>
    <food>
        <name>Homestyle Breakfast </name>
        <price>195.95din </price>
        <description>
            two eggs, bacon or sausage, toast
        </description>
        <calories>950 </calories>
    </food>
</menu>
```

Still, XML provides for yet another type of data models, which is an ordered tree with typed, named nodes and data in leaf nodes only. An XML document is a linearization of the tree structure.

There are number of advantages of using XML data format over other data formats, e.g., relational one. For example, heterogeneity of data records is supported in a more natural way, extensibility is provided by allowing different data types in a single document, as well as flexibility through variety in size and configuration among

instances of the same data type. Sometimes, manipulating XML data format is significantly more efficient than traditional ones. At the same time, there are obvious disadvantages of XML data format, one of them being inefficiency in record format. Data manipulation is often slower than in traditional formats, and optimization is complex due to richness and expressiveness of query languages.

2. XML DATA MODELS

Unlike relational data model, there is no unique XML data model. Still, all XML data models are extensions of the basic one and some of them will be briefly presented.

The basic XML model includes different types of nodes, such as [1]:

- **element** node, e.g.,
  `<price>195.95din </price>` or
  `<menu date='5.10.2006'> ...</menu>`
- **document** node, one special kind of element node, represented by the `<!DOCTYPE>` node;
- **processing instruction** node, e.g., `<?xml version...?>`
- **comment** (in the form of `<!--c-->`)
- **data** which always reside in leaf nodes, with only one characteristics: data itself, e.g., "`Homestyle Breakfast`" or "`two eggs, bacon or sausage, toast`".

Element node has a **type**, e.g., `price`, `food` or `menu`; it also may have an ordered list of **children** (for the element node of type `food` these are element nodes of type `name`, `price`, `description`, `calories`), and an unordered set of attributes of the form (attribute_name – attribute_value) pairs, e.g., "`date='5.10.2006' `".

The document node has a **type** (`menu` in our example) but no attributes; it also has exactly one element node child, which must have the same type as the document node (in our example, `menu`-typed element node).

XML may also be considered as an abstract data type (ADT) which is very rich, containing Strings and Identifiers, partly ordered in a sequence, partly hierarchical,

partly in an unordered database-like keyword/value system. The ADT XML Node actually imports previously defined data types Identifier, URL, char, int, boolean, nil, and defines mappings between the types (functions) for creating an XML document, setting and getting schema, attributes, manipulating nodes, as well as linearization functions.

Finally, there is an XML data model based on languages for querying them, XQuery and XPath. The so called XDM, XQuery 1.0 and XPath 2.0 Data Model, is a W3C Candidate Recommendation as of the end of 2005.

2.1. DATA-CENTRIC AND DOCUMENT-CENTRIC XML DOCUMENTS

Regarding rigidity of XML document structure, XML documents fall into two broad categories: *data-centric* and *document-centric*. Data-centric documents are those containing structured data, such as price lists. Data appear in a regular order and are usually stored in databases while XML is used just for data exchange and publishing. They may also contain semi-structured data, such as phonebooks or patient records. In general, relational databases are efficient enough in storing data contained in data-centric XML documents.

Document-centric XML documents are those characterized by irregular structure and mixed content, such as in user's manuals and marketing brochures. Storing and manipulating various XML documents in a shared repository usually requires more than a relational database.

3. XML AND DATABASES

Regardless of whether XML is used as a storage or interchange format for data-centric model data, or used for creating semi-structured document-centric model documents, such as XHTML, it is sometimes necessary to store the XML in some sort of repository or database that allows for more sophisticated storage and retrieval of the data, especially if the XML is to be accessed by multiple users.

There are conceptually two different ways to store XML documents in a database. The first is to map the document's data model to a database model and convert XML data into the database, according to that mapping. The second is to map XML model into a fixed set of persistent (database) structures that can store any XML document. Databases that support the first method are called *XML-enabled databases*. Databases that support the second method are called *native XML databases*. XML-enabled databases have their own data model – relational, hierarchical, object-oriented, and they map instances of the XML data model to instances of their data model. Native XML databases use the XML data model directly [2]. Although the choice is somewhat arbitrary, it is usually more convenient and more efficient to store and manipulate data-centric XML documents using XML-enabled databases, and document-centric XML documents using native XML databases.

3.1. XML-ENABLED DATABASES

One way to map XML documents into an XML-enabled database is to create relational views on XML documents stored in columns of a relational database, which can then be queried using SQL. In XML-enabled databases, different XML document schemas correspond to different database schemas. XML is external to the database and invisible inside it.

As an example of such a database we may consider the "`menu`" XML document. It may be represented (either physically or as a relational view) as the following relation:

`food`	`name`	`price`	`description`	`calories`
	Homemade Bean Soup	100.00	a bowl of white bean soup with paper and onion	650
	French Toast	60.50	thick slices made from our homemade bread	300
	Homestyle Breakfast	195.95	two eggs, bacon or sausage, toast	950

A query asking for foods with more than 600 calories may be now stated by the following SQL statement:

```
select name
from food
where calories > 600
```

If we had yet another two relations,

`restaurants(name, address)`, `serving(food-name, rest-name)`

we would be able to formulate an SQL join-query asking for information on where to eat for what amount of money:

```
select restaurant.address, food.price
from food, restaurant, serving
where food.name=food-name and restaurant.name=rest-name
```

3.2. NATIVE XML DATABASES

Native XML database (NXD) is best described as a database that has an XML document (or its rooted part) as its fundamental unit of (logical) storage and defines a (logical) model for an XML document, as opposed to the data in that document (its contents). It represents logical XML document model (not XML document data model), and stores and manipulates documents according to that model [2].

Basic characteristics of an NXD are the following:

- a logical unit of an NXD is an XML document or its rooted part, and it corresponds to a row in a relational database,
- it includes at least the following components: elements, attributes, textual data (PCDATA), and document order,
- physical model (and type of persistent NXD storage) is unspecified.

In a native XML database, XML *is* visible inside the database. There is a unique database for all XML schemas and documents. Native XML databases are especially suitable for storing irregular, deeply hierarchical, recursive data.

Following the definition, an important characteristics of a native XML database is that its physical model is unspecified. It further implies that XML documents persistent storage may be arbitrary, as long as it stores and manipulates an XML

document as a (logical) whole. Different types of persistent storage may be disk files, CLOB fields in relational databases, fixed relational database tables, DOM trees in object-oriented databases, hash tables, etc.

For example, XML `menu` document may be represented in a native XML database stored in fixed relational database tables in the following way:

Documents	Doc_id	Doc_name
	1	simple.xml
	...	...

Elements	Doc_id	El_id	Parent_id	Name	OrdInParent
	1	1	NULL	menu	1
	1	2	1	food	1
	1	3	2	name	1
	1	4	2	price	2
	...	...	...	...	...

Attributes	Doc_id	Atr_id	Parent_id	Name	Value
	1	1	1	date	5.10.2006
	...	...	...	...	...

Text	Doc_id	Text_id	Parent_id	Value
	1	1	3	Homemade Bean Soup
	1	2	4	100.00
	...	...	...	...

Although the example shows how to build a native XML database on top of a relational database, most native XML databases are built from scratch, as stand-alone document management systems. Those systems manage collections of documents, allowing users to query and manipulate those documents as a set, which is similar to the relational concept of a table.

The previous example pertained to a NXD solution for a data-centric XML document ("`menu`"), but the real power of NXDs comes with document-centric XML documents. They provide for XML data model, which is flexible enough to model documents, XML-aware full-text searches, and structured query languages like XQuery.

Advantages of using native XML databases over other types of databases are

numerous. They free users from having to know document schema in advance (prior to designing the database), they support data models that does not fit other databases (e.g., relational databases), provide for extensibility, etc.

Common NXD applications include document management, support for semi-structured data, support for sparse data, catalog data, manufacturing parts databases, medical information storage, etc. [7]

4. XML QUERY LANGUAGES

A number of languages have been created for querying XML documents including XML-QL, XPath, XQL, XQuery. Among them, XPath and XQuery are W3C candidate recommendations [10] and are used for retrieving and manipulating data from most NXD implementations.

XML Path Language (XPath) is a language for addressing parts of an XML document through hierarchical paths similar to those used for a filesystem or URL. It provides built-in functions and is extensible regarding user-defined functions. XPath operates on a single XML document.

Examples of XPath queries against the `simple.xml` document ("`menu`" document) are the following:

a. `/menu/food/name` (selects all name elements that are children of food elements that are children of the root element `menu`).
b. `///price` (selects all `price` elements in the document).
c. `/menu/*` (selects all child elements of the root element `menu`).
d. `/menu[@date]` (selects the date attribute of the menu element).
e. `//*[name()='price']` (selects all elements that are named "`price`").

XML Query Language (XQuery) is an attempt to design a query language with at least the functionality of SQL. XQuery is a functional language where each query is an expression. XQuery expressions fall into seven broad types: path expressions,

element constructors, FLWOR expressions, expressions involving operators and functions, conditional expressions, quantified expressions or expressions that test or modify datatypes. For example, a FLWOR expression is a query construct composed of FOR, LET, WHERE, ORDER BY and a RETURN clauses, with obvious meaning. XQuery supports broader manipulation of document nodes than XPath. The data model of XQuery is also broader than that of XPath and operates on fragments of documents, single XML document, sequence of documents, or sequence of document fragments.

The following are some examples of XQuery queries and expressions:

a. path expressions: `//food[name="Homemade Bean Soup"]/price * 100`

   (From our document containing the restaurant menu, extract the price in cents of the food named "Homemade Bean Soup")

b. element constructors: {`$name`}

   (Generate an element with the content which is specified by the variable that obtained a value in other parts of the query)

c. FLWOR expressions:

```
FOR $b IN document("simple.xml")//food
WHERE $b/calories>"600" AND $b/price<"110"
RETURN $b/name
```

   (List the names of all the foods with more than 600 calories and cheaper than 110 dinars).

d. conditional expressions:

```
FOR $h IN food
RETURN
<cheap>
{$h/name,}
IF ($h/price<"110")
THEN "cheap"
ELSE "expensive"
</cheap>
```

   (Make a list of al the foods on the menu, with the mark "cheap" if cheaper than 110 dinars, and the mark "expensive", otherwise).

e. quantified expressions: The SOME clause is an existential quantifier used for testing if a series of values contains at least one node that satisfies a predicate. The EVERY clause is a universal quantifier used to test if all nodes in a series of values satisfy a predicate.

```
FOR $b IN //book
WHERE SOME $p IN $b//para SATISFIES
    (contains($p, "sailing") AND contains($p, "windsurfing"))
RETURN $b/title
```

(For an XML document `book`, find titles of books in which both "sailing" and "windsurfing" are mentioned in the same paragraph [5]).

f. expressions involving user defined functions:

```
DEFINE FUNCTION depth($e) RETURNS integer
{
# An empty element has depth 1
# Otherwise, add 1 to max depth of children
IF (empty($e/*)) THEN 1
ELSE max(depth($e/*)) + 1
}
depth(document("partlist.xml"))
```

(declaration of a function which finds the maximum depth of a document `$e`, and the function call for the document named "partlist.xml.").

g. Join queries may be formulated using nested `for` constructs. For example, if we had, apart from the document `simple.xml`, two other documents, `restaurant.xml` and `serving.xml`, the following query would find prices charged at specific restaurant addresses:

```
FOR $x IN /menu/food
FOR $y IN /restaurant
FOR $z IN /serving
WHERE $x/name=$z/food_name AND $y/name=$z/restaurant_name
RETURN ($y/address, $x/price)
```

5. DATA-CENTRIC XML DOCUMENT EXAMPLE: PART EXPLOSION PROBLEM

Given a manufacturing company database, there may be a data structure indicating that certain parts include other parts as immediate components (the so-called `bill-of-materials` relationship) [3]. **Part explosion problem** is a well known problem recursively defined on this relationship: Get part numbers for all parts that are components, *at any level*, of some given part, or of each part.

Original data may be represented in a relational database by the table `part_structure`, e.g.:

MAJOR_P	MINOR_P
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6

(Meaning of the data is that the part P1 includes the parts P2 and P3 as its immediate components, that the part P2 includes parts P3 and P4 as immediate components, etc).

The same data can also be represented by the following XML file `part_structure.xml`.

```
<root>
<tuple>
    <major_p> P1 </major_p>
    <minor_p> P2 </minor_p>
</tuple>
<tuple>
    <major_p> P1 </major_p>
    <minor_p> P3 </minor_p>
</tuple>
...
</root>
```

The result of "exploding" the initial `part-structure` data is the set of (MAJOR_P, MINOR_P) pairs represented by the following `subparts` list (pairs in the list

are numbered by numbers in parentheses):

MAJOR_P	MINOR_P	
P1	P2	(1)
P1	P3	(2)
P2	P3	(3)
P2	P4	(4)
P3	P5	(5)
P4	P5	(6)
P5	P6	(7)
P1	P4	(8)
P1	P5	(9)
P2	P5	(10)
P3	P6	(11)
P4	P6	(12)
P1	P6	(13)
P2	P6	(14)

Departing from the relational representation of the initial data (`part_structure` table), one can create the resulting table `subparts` either by non-recursive or by recursive SQL queries.

Non-recursive queries may be formulated first to copy initial data (from the `part_structure` table) into the `subparts` table (query **I** executing only once and adding just direct subparts, resulting in first 7 pairs), and then to add indirect components – subparts at consecutive levels (query **II**, executing as many times as there are levels of indirection):

(I)
```
insert into subparts
    (select major_p, minor_p
     from part_structure);
```

(II)
```
insert into subparts
     (select distinct first.major_p, p_s.minor_p
      from subparts first, part_structure p_s
      where subparts.minor_p=p_s.major_p and
         not exists(select *
             from subparts
             where major_p=first.major_p and minor_p=p_s.minor_p));
```

In our example, the query **II** is executed twice, since there are two levels of indirection, first adding pairs 8–12, and then adding pairs 13 and 14.

Recursive SQL statement that produces the same result (in the `subparts` relation) in one execution is the following:

```
with subparts(major_p, minor_p) as
    (select major_p, minor_p
     from part_structure
     union all
     select subparts.major_p, part_structure.minor_p
     from subparts, part_structure
     where subparts.minor_p=part_structure.major_p )
select distinct major_p, minor_p
from subparts;
```

5.2. PART EXPLOSION PROBLEM: NATIVE XML DB APPROACH

In a native XML DB, the `part_structure.xml` file is put into a *container* `part_structure.dbxml`, and `subparts` result is created and stored in the container `subparts.dbxml`.

The non-recursive relational query (**I**) corresponds to the following XQuery statement:

(I)
```
putDocument
  'for $p in collection ("part_structure.dbxml") /root/tuple
   return {$p/major_p}{$p/minor_p}'
```

The non-recursive relational query (**II**) corresponds to the following XQuery statement:

(II)
```
putDocument
  'for $p in collection ("part_structure.dbxml") /root/tuple
   for $r in collection("subparts.dbxml")/tuple
   where every $k in collection("subparts.dbxml")/tuple satisfies
        not($k/major_p=$r/major_p and $k/minor_p=$p/minor_p)
        and $r/minor_p=$p/major_p
   return distinct-values({$r/major_p}{$p/minor_p})'
```

Recursive XQuery, corresponding to the recursive relational query, may be defined using a recursive user-defined function of the following form:

```
declare function local:sub_part($tuple as node())
as node()*
{
  let $subID := $tuple/minor_p/text()
  for $subpart in $tuple/../tuple[major_p=$subID]
  return
  ($tuple/major_p, if(exists($subpart)) then local:sub_part($subpart) else ())
};
```

6. DOCUMENT-CENTRIC XML DOCUMENT EXAMPLE: SERBIAN WORDNET

Serbian wordnet is a lexical-semantic network of Serbian language [6]. Its development started in the framework of the *BalkaNet* project [8], successor of the Princeton WordNet and EuroWordNet projects [9].

Wordnet is structured around the notion of *synset* (set of synonyms), reflecting the synonymy relation. Nouns, verbs and adjectives are organized in sets of synonyms representing individual lexical concepts. Each synset consists of one or more words (*literals*), used in a specific *sense*, of the same *part of speech*, with *meanings* that may be considered identical and are defined by a *gloss*. For example, a synset pertaining to the computer file concept may consists of a literal *file* in the sense *2* (different senses of the same literal string are numbered), but also of a literal *data file* in the sense *1*, with the *noun* part of speech and defined by the following gloss: *set of records that are stored together*.

Different relations connect synsets (i.e., concepts). These are hypernym / hyponym, near_antonymy, holonymy / meronymy, being derived, be in state, causes, being similar to, being a subevent, etc. Except for the synonymy relation, defining the concept of a synset, the most important is the hypernym / hyponym (is-a) relation used for building a hierarchy of concepts (figure). Certain number of concepts are on the top, not having hypernyms (the so-called *ontological terms*). For example, there are 11 such noun concepts in (English) WordNet, such as *entity*, *psychological feature*, *abstraction*, *event*, *act*, *phenomenon*, etc.

The Serbian wordnet (WNSRP) is an XML file containing more than 11000 synsets at the moment, with about 10% of proper names. It gives rise to different applications on texts (e.g., classification), relying on presence of certain ontology proper names.

An example of a synset from the WNSRP, corresponding to the *deity* ("božanstvo") concept, is the following:

```
<SYNSET> <ID>ENG20-08904620-n </ID> <SYNONYM> <LITERAL>bozxanstvo
<SENSE>1</SENSE><LNOTE>N330 </LNOTE> </LITERAL> </SYNONYM>
<DEF>Natprirodno bicxe koje se obozxava zbog verovanxa da upravlxa nekim
delom sveta ili nekim aspektima zxivota ili zato sxto personifikuje silu.
</DEF><POS>n</POS><ILR>ENG20-08903509-n<TYPE>hypernym </TYPE> </ILR>
<ILR>ENG20-07660421-n<TYPE>holo_member</TYPE></ILR><BCS>1</BCS>
<STAMP>User 2004/01/07 </STAMP> </SYNSET>
```

The meaning of (some of) the elements is the following: ID is the synset identifier unique across different languages, POS is the part of speech, ILR is the ID of a concept (synset) related to this one by a relation of the type TYPE, etc.

Several tools have been designed for manipulating wordnets. One of them is the VisDic [11] which supports choice of different languages, configuring wordnets for selected languages, editing synsets or retrieving them in their hierarchical form, XML form etc. Examples are presented for choice of English and Serbian, on figures and , where figure represents a hierarchy of hypernyms for the synset "Zeus", through eight levels, up to the ontology term "psychological feature", in English and Serbian. Figure represents an expanded list of hyponyms of the concept "Greek deity".

Although basic wordnet representation is the XML one, different alternative representations exist and some of them are implemented in some of the designed tools. One of them is a relational representation comprising of several relations, such as the following:

```
lemma_pos (lemmano, posno, lastno)
lu_lemma (lemmano, lemma)
lu_pos (posno, pos)
lu_relation (relno, relation, reverse)
sense (synsetno, lemmano, posno, senseno)
synset (synsetno, synsetid, base, posno, synset_name, gloss)
synset_relation (parentno, relno, childno, distance)
```

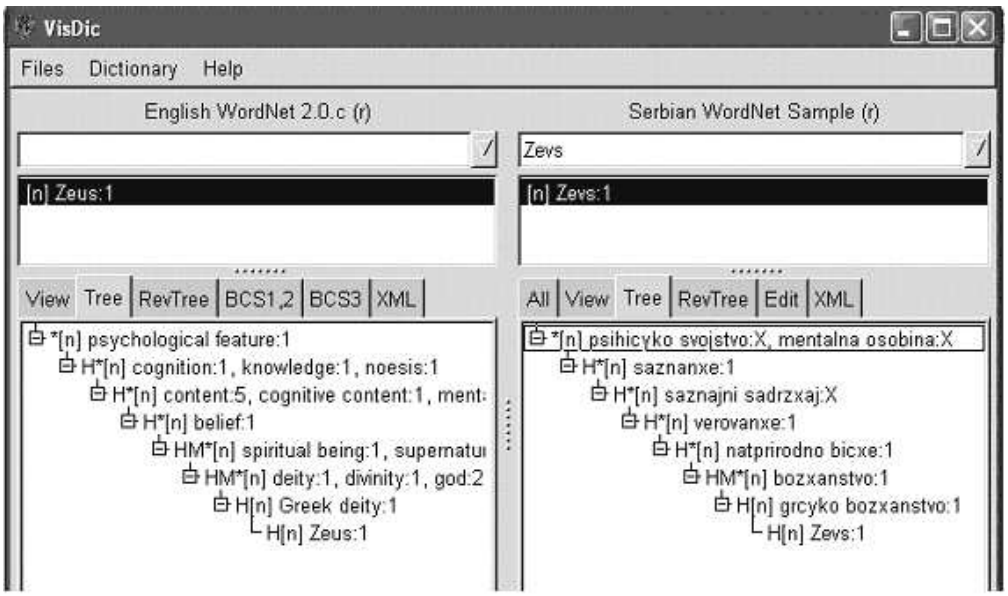

Figure 1: Hierarchy of hypernyms for the synset "Zeus".

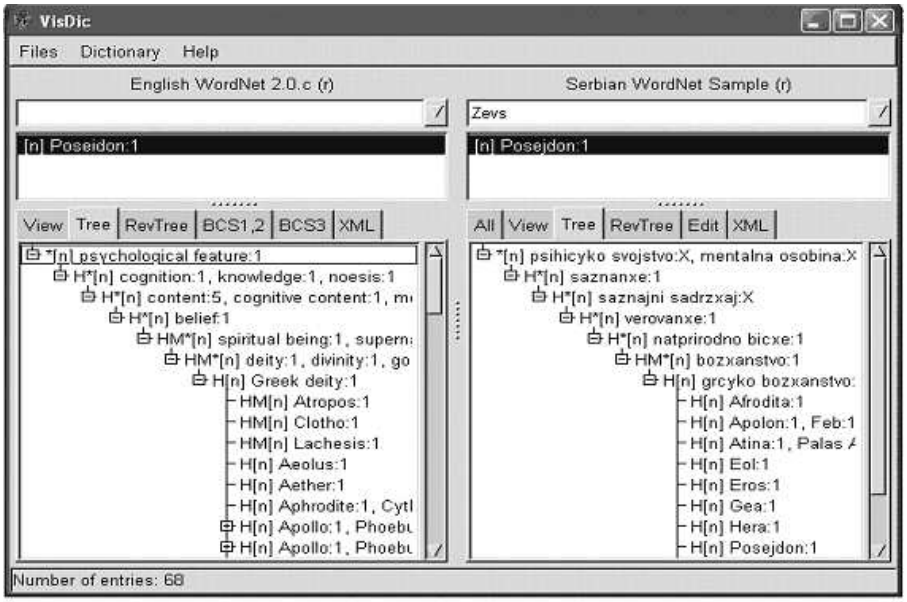

Figure 2: An expanded list of hyponyms of the concept "Greek deity"

Meaning of the attributes are the following: `lemma` is a literal (textual entry), `pos` is part of speech, `lemmano`, `posno`, `relno`, `synsetno`, `senseno`, `parentno` are

numbers, in the corresponding ordering, of a lemma, part of speech, relation, synset, sense etc.

Although relational representation of the WNSRP is obviously possible (with some respects even more efficient), the structure of the document in a container stored in a native XML DB, with synset as a basic node and a complete structure of elements, is much more natural. Both representations may be queried using the corresponding query languages (SQL, XQuery) in a very similar manner as for the part explosion problem. For example, extracting all the Greek deities may be realized by the following XQuery query:

```
'for $e in collection("wnsrp.dbxml")/ROOT/SYNSET
 where $e/SYNONYM/LITERAL/text() = "grcyko bozxanstvo"
 return
    for $r in collection("wnsrp.dbxml")/ROOT/SYNSET
    where ($e/ID/text() = $r/ILR/text())
    return $r'
```

Creating the hierarchy of their hypernyms (up to ontology terms) may be realized by repeatedly executing the following query (and adding the result to the container `hypernym`):

```
'for $e in collection("hypernym.dbxml")/SYNSET
 for $r in collection("wnsrp.dbxml")/ROOT/SYNSET
 where every $k in collection("hypernym.dbxml")/SYNSET
  satisfies not ($k/ID/text()=$r/ID/text()) and $e/ILR/text()=$r/ID/text()
 return $r'
```

7. CONCLUSION

Native XML database is a practical concept addressing pragmatic issues such as functionality, efficiency, expressiveness, and residing at the application side of data modelling. Languages such as XML and XQuery implement a different way of representing and querying a collection of data which is more flexible and less structured than traditional relational data. XML and NXD tend to be a more "practical" solution for data that are irregular, deeply hierarchical, and recursive, in lot of real world

documents. A pure relational model can be built as well as a relatively portable SQL implementation, but a number of practical limitations are faced, such as multiway joins, complex queries, etc. Native XML databases and XML itself allow one to work with such data in a standardized, portable, and reasonably efficient way.

Acknowledgements: The work presented has been financially supported by the Ministry of science and environmental protection of the Republic of Serbia, Project No. 148021.

References

[1] B. Bos, *The XML data model*, http://www.w3.org/XML/Datamodel.html (2005).

[2] R. Bourret, *Going Native: Making the Case for XML Databases*, http://www.xml.com/pub/a/2005/03/30/native.html (2005).

[3] C. J. Date, *An Introduction to Database Systems*, 6th ed, Addison-Wesley Publ. Comp (1995).

[4] M. Kay, *Blooming FLWOR - An Introduction to the XQuery FLWOR Expression*, http://www.stylusstudio.com/xquery_flwor.html (2004).

[5] D. Obasanjo,.*An Exploration of XML in Database Management Systems*, http://www.25hoursaday.com/StoringAndQueryingXML.html (2001).

[6] G. Pavlović-Lažetić, *Electronic Resources of Serbian: Serbian WORDNET*, 36th International Slavic Conference, MSC, Belgrade, Serbia, september 2006.

[7] K. Staken, *Introduction to Native XML Databases*, http://www.xml.com/pub/a/2001/10/31/nativexmldb.html (2001).

[8] S. Stamou et al., *BALKANET: A Multilingual Semantic Network for Balkan Languages*, in: Proceedings of 1st International Wordnet Conference, Mysore, India (2002).

[9] P. Vossen, ed., *EuroWordNet: A Multilingual Database with Lexical Semantic Networks*, Kluwer Academic Publishers (1998).

[10] *W3C XML Query (XQuery)*, http://www.w3.org/XML/Query/ (2005).

[11] http://nlp.fi.muni.cz/projekty/visdic/