Fragmentation of XML Documents

Hui Ma[1], Klaus-Dieter Schewe[2]

[1] Victoria University of Wellington, School of Engineering and Computer Science, Wellington, New Zealand
hui.ma@ecs.vuw.ac.nz
[2] Software Competence Center Hagenberg, Hagenberg, Austria
kdschewe@acm.org

Abstract. The world-wide web (WWW) is often considered to be the world's largest database and the eXtensible Markup Language (XML) is then considered to provide its datamodel. Adopting this view we have to deal with a distributed database. This raises the question, how to obtain a suitable distribution design for XML documents. In this paper horizontal and vertical fragmentation techniques are generalised from the relational datamodel to XML. Furthermore, splitting will be introduced as a third kind of fragmentation. Then it is shown how relational techniques for defining reasonable fragments can be applied to the case of XML.

Categories and Subject Descriptors: Information Systems [**Miscellaneous**]: Databases

Keywords: Distribution, Fragmentation, XML

1. INTRODUCTION

The world-wide web (WWW) is often considered to be the world's largest database and the eXtensible Markup Language (XML) is then considered to provide its datamodel. Of course, this is only partly true, as large parts of the WWW have nothing to do with databases nor do they permit to be considered as an instance of some suitable data model. However, if we adopt this view, then we need (among others) adequate languages for the definition of schemata, for querying this database, and for updating it.

Originally, XML already provides some kind of schema definition language called Document Type Definitions (DTDs), which has been criticised for various deficiencies [Abiteboul et al. 2000; Goldfarb and Prescod 1998]. The gist of these critics refer to the absence of typing, the unspecified targets of references, and the use of global names. It seems to be a matter of taste, whether the right way to overcome them is to extend DTDs or to create separate schema definition languages. Though these may just lead to variations in syntax rather than expressiveness, the approach favoured by many researchers is the second one. This has already led to the proposal of several schema definition languages such as SSD-Schema [Buneman et al. 1997], SOX [Fuchs et al. 1998] and XML-Schema [World Wide Web Consortium (W3C) 2001].

As to querying the database research community has produced various proposals for query languages such as Lorel [Abiteboul et al. 1997], UnQL [Buneman et al. 1996], YAP [Cluet et al. 1998], XML-QL [Deutsch et al. 1999] and XQuery [World Wide Web Consortium (W3C)], most of them following an approach, which could roughly be characterised as creating an SQL-like language that is enhanced by using more sophisticated path expressions. Attempts to promote more expressive query languages in the tradition of DATALOG (see [Abiteboul et al. 2000]) have so far been only discussed among database theoreticians.

In contrast to querying the problem of updating XML documents has only received little attention. The work in [Tatarinov et al. 2001] provides one of the few exceptions dealing with updates in XML.

Another consequence of adopting the view of parts of the WWW being a database is that we have to deal with a distributed database. Thus, in order to design data for the web, we should raise the question, how to obtain a suitable distribution design for XML documents. In fact there are two possible approaches to this problem. The first one would build up a database federation out of existing databases that are described by XML documents. The second one would apply distribution techniques to a global XML database. Here we focus only on the second approach.

In the context of the relational datamodel (RDM) [Ceri and Pelagatti 1984; Özsu and Valduriez 1999] the central issues were the fragmentation of schemata and the allocation of the fragments to the machines in a computer network. Finding a reasonable distribution of the data by using the techniques of fragmentation and allocation is usually accompanied by a cost model for transaction and query processing time. In this paper we concentrate only on fragmentation.

Fragmentation is commonly divided into horizontal fragmentation, which splits a relation into disjoint unions, and vertical fragmentation, which projects a relation onto a subset of its attributes. Horizontal and vertical fragmentation has been discussed intensively [Chu 1992; Chu and Ieong 1993; Cornell and Yu 1987; Lin et al. 1993], and also integrated approaches have been tried [Tamhankar and Ram 1998]. Several authors have approached the generalisation of fragmentation techniques to object oriented datamodels. For instance, horizontal fragmentation is discussed in [Bellatreche et al. 2000; Schewe 2002], and vertical fragmentation in [Malinowski and Chakravarthy 1997; Schewe 2002]. Due to the similarity between object oriented and semi-structured data, some of these techniques have also been applied to semi-structured data [Schewe 2002].

In order to determine reasonable fragments it is quite common to consider just the “most important” or “most frequent” queries and transactions, and to regard the involved basic queries. In this article we take a similar approach to the fragmentation of XML documents. We adapt the fragmentation techniques from [Schewe 2002] to XML and discuss how relational techniques for defining reasonable fragments can be applied to the case of XML.

Outline.. In Section 2 we describe our preliminaries, i.e., we ‘introduce’ XML as a datamodel using an extension of DTDs for describing schemata, and a variant of XML-QL to describe queries. These queries will be used for fragmentation as well as for discussing cost optimisation. Section 3 deals with fragmentation techniques. Following the approach in [Schewe 2002] we distinguish between splitting, horizontal and vertical fragmentation. Finally, Section 4 handles an approach based on frequent base queries to find a reasonable fragmentation.

2. XML AS A DATAMODEL

In this section we describe XML as a datamodel. We use extended DTDs to define schemata. Equivalently, we could have used XML-Schema, but extensions would be needed in both cases. Then it is standard to consider XML documents as databases over such schemata. The third subsection deals with queries, where we use an extension of XML-QL. Equivalently, we could use XQuery, but again extensions would be needed in both cases.

2.1 Schemata and Document Type Definitions

A *document type definition* (DTD) may be considered as some kind of schema. Within such a DTD the regular expressions can be considered as some form of typing. We will make this view explicit and introduce first a typed version of XML. Types to be considered are (using abstract syntax)

$$t \quad = \quad b \mid \epsilon \mid t^0 \mid t^* \mid t^+ \mid t_1, \dots, t_n \mid t_1 \uplus \cdots \uplus t_n \quad .$$

Here, b represents as usual a collection of base types. Among these base types we assume to have a type *ID*, i.e., a type representing a not further specified set of *identifiers*. There may be other base types such as *INT* for integers, *STRING* for character strings, *URL* for URL-addresses, etc. ϵ is a type representing just an empty sequence or tuple. t^* and t^+ represent arbitrary or non-empty sequences, respectively, with values of type t. t^0 represents values of type t or the empty sequence. $t_1, \dots, t_n$ represents sequences or tuples. Finally, $t_1 \uplus \cdots \uplus t_n$ represents a disjoint union.

Note that the basic form of XML DTDs only provides a base type *PCDATA*. Furthermore, we changed the basic notation of XML in order to avoid confusion with the symbols used on the meta-level. Usually, ϵ is denoted *EMPTY*, a question mark is used instead of ^{0}, and alternative is expressed by | rather than using $\uplus$.

More formally, we can associate with each type t a *domain* $dom(t)$, defined as follows:

$$\begin{aligned}
dom(b_i) &= V_i \text{ for all base types } b_i\\
dom(\epsilon) &= \{()\}\\
dom(t^*) &= \{(a_1, \dots, a_k) \mid k \in \mathbb{N}, a_i \in dom(t) \text{ for all } i = 1, \dots, k\}\\
dom(t^+) &= \{(a_1, \dots, a_k) \mid k \in \mathbb{N}, k \neq 0, a_i \in dom(t) \text{ for all } i = 1, \dots, k\}\\
dom(t^0) &= \{(a) \mid a \in dom(t)\} \cup \{()\}\\
dom(t_1, \dots, t_n) &= \{(a_1, \dots, a_n) \mid a_i \in dom(t_i) \text{ for all } i = 1, \dots, n\}\\
dom(t_1 \uplus \cdots \uplus t_n) &= \{(i, a_i) \mid i = 1, \dots, n, a_i \in dom(t_i)\}
\end{aligned}$$

Now we can define a *schema* basically as a sequence of element declarations with one distinguished root-element. An *element declaration* has the form ⟨`!ELEMENT name expression` ⟩, where `name` is an arbitrarily chosen name for the element, and `expression` is either a base type (different from *ID*) or a regular expression e made out of element names n:

$$e \quad = \quad n \mid \epsilon \mid (e)^* \mid (e)^+ \mid (e)^0 \mid (e_1, \dots, e_n) \mid (e_1 \uplus \cdots \uplus e_n) \quad .$$

The schema is then defined by ⟨`!DOCTYPE root [ elements ]`⟩ with a sequence `elements` of element (and attribute) declarations, and the name `root` of the root-element.

We can extend domains in the obvious way to elements. For an element with the name n and the defining expression e we obtain the domain $dom(n)$ as a set of values $\langle n \rangle \bar{e} \langle /n \rangle$ with

$$\begin{aligned}
&\bar{e} \in V_i \text{ , if } e \text{ is the base type } b_i\\
&\bar{e} \in \{v_1 \dots v_k \mid v_i \in dom(n_i) \text{ for } i = 1, \dots, k\}, \text{if } e = (n_1, \dots, n_k)\\
&\bar{e} \in \{v_1 \dots v_k \mid k \in \mathbb{N}, v_i \in dom(n') \text{ for } i = 1, \dots, k\}, \text{if } e = (n')^*\\
&\bar{e} \in \{v_1 \dots v_k \mid k \in \mathbb{N}, k \neq 0, v_i \in dom(n') \text{ for } i = 1, \dots, k\}, \text{if } e = (n')^+\\
&\bar{e} \in \{v_i \mid (i, v_i) \in dom(n_i) \text{ for } i = 1, \dots, k\}, \text{if } e = (n_1 \uplus \cdots \uplus n_k)
\end{aligned}$$

For $e = (n')^0$ $\bar{e}$ must be either in $dom(n')$ or be simply omitted. For $e = \epsilon$ we always omit $\bar{e}$. As usual in XML we abbreviate empty elements by $\langle n/ \rangle$.

Then an instance of a schema with root-element named n is simply a value in $dom(n)$. This is usually called an *XML document*. However, as we consider XML as a datamodel, we could equivalently consider this to be a *database* over the schema defined by the DTD. As we want to consider several schemata at a time, we extend this definition and define an instance of a schema with root-element named n to be a pair (u, v), where u is a URL-address, i.e., a value of type *ID*, and v is a value in $dom(n)$.

Note that these definitions of schemata and documents do not yet include attributes, thus do not capture the full power of XML as a datamodel. In particular, references between elements are missing.

2.2 Attribute Declarations

Let us now add attributes to elements. An *attribute declaration* has the form $\langle$`!ATTRIBUTE element name type key`$\rangle$, where `element` is the name of an element, `name` is an arbitrarily chosen attribute name, and `key` is one of `#REQUIRED` or `#IMPLIED` indicating, whether the attribute is mandatory or not. `type` is either a base type, *ID*, one of the two possibilities `REF name`$'$ or `REFS name`$'$ indicating single or multiple references within the same document, or one of the two possibilities `url.REF name`$'$ or `url.REFS name`$'$ with a URL-address `url` indicating single or multiple references to elements in the XML document at the given URL.

Some comments are due for this form of attribute declarations. A minor change with respect to XML is that we prefer a separate declaration for each attribute instead of an attribute-list declaration. We also omitted further options for keys, but only for brevity. We added the possibility to specify base types other than `CDATA` or `PCDATA`, which is also only a minor extension. The major deviation from XML is with references, which now require a specified target element with an attribute of type *ID* to be included. Extending references to other documents turns out to be necessary for fragmentation.

Example 2.1. The following is a simple example for a schema:

$\langle$!DOCTYPE db [
$\langle$!ELEMENT db (wine $\uplus$ vineyard $\uplus$ region)* $\rangle$
$\langle$!ATTRIBUTE db created *DATE* #`IMPLIED` $\rangle$
$\langle$!ELEMENT wine (name, year^0, (grape* $\uplus$ grapes)^0) $\rangle$
$\langle$!ATTRIBUTE wine w-id *ID* #`REQUIRED` $\rangle$
$\langle$!ATTRIBUTE wine producer `REF` vineyard #`REQUIRED` $\rangle$
$\langle$!ATTRIBUTE wine price *INT* #`IMPLIED` $\rangle$
$\langle$!ELEMENT name *STRING* $\rangle$
$\langle$!ELEMENT year *INT* $\rangle$
$\langle$!ELEMENT grape *STRING* $\rangle$
$\langle$!ELEMENT grapes (grape, percentage)$^+$ $\rangle$
$\langle$!ELEMENT percentage *INT* $\rangle$
$\langle$!ELEMENT vineyard (v-name, owner$^+$, area^0) $\rangle$
$\langle$!ATTRIBUTE vineyard v-id *ID* #`REQUIRED` $\rangle$
$\langle$!ATTRIBUTE vineyard in-region `REF` region #`REQUIRED` $\rangle$
$\langle$!ATTRIBUTE vineyard established *DATE* #`IMPLIED` $\rangle$
$\langle$!ELEMENT v-name *STRING* $\rangle$
$\langle$!ELEMENT owner *STRING* $\rangle$
$\langle$!ELEMENT area *INT* $\rangle$
$\langle$!ELEMENT region *STRING* $\rangle$
$\langle$!ATTRIBUTE region r-id *ID* #`REQUIRED` $\rangle$
$\langle$!ATTRIBUTE region most-famous-wines `REFS` wine #`IMPLIED` $\rangle$
] $\rangle$

In order to define XML documents, i.e., the "databases" for our purposes here, we have to extend the definition of the domain $dom(n)$ for an element named n. If $a_1, \dots, a_k$ are the attributes of n and we obtain suitable values $v_1, \dots, v_k$ for these attributes, we now extend $dom(n)$ to be a set of values $\langle n\ a_1 = v_1 \dots a_k = v_k\rangle \bar{e} \langle /n\rangle$ with $\bar{e}$ being defined as in the previous subsection. Attributes a_i that are declared to be optional can be omitted in such values.

Thus, we only have to be precise about the "suitable" values v for an attribute a. Obviously, if the type of a is a base type b_i including *ID*, v must be in $dom(b_i)$. If the type is `REF` n', we also get an identifier, i.e., $v \in dom(ID)$. If the type is `REFS` n', v must be a sequence of identifiers, i.e., $v \in dom(ID^*)$. The same applies to the types `url.REF` n' and `url.REFS` n' with the difference that the given URL `url` has to be used as a prefix, i.e., we obtain $v = \texttt{url}.v'$ with $v' \in dom(ID)$ or $v' \in dom(ID^*)$, respectively.

With these extensions an instance of a schema with root-element named n is again a pair (u, v) with $u \in dom(URL)$ and $v \in dom(n)$, but the usual restrictions to identifiers and references apply:

—Each value of type ID may appear at most once in v as a value of an attribute of type ID.

—For each value i of type ID that appears in v as a value of an attribute of type `REF` n' (or within a value of an attribute of type `REFS` n') there must exist an element named n' inside v and an attribute of n' of type ID with the value i.

—For each value i of type ID that appears in v as a value of an attribute of type `url.REF` n' (or within a value of an attribute of type `url.REFS` n') there must exist an XML document (u', v') at the URL $\texttt{url} = u'$, and an element named n' inside v' with an attribute of n' of type ID with the value i.

Note that though we extended DTDs, we did not resolve the problem of global element names. On one hand solutions to this particular problem are not very difficult and have been handled in XML-Schema anyway. On the other hand, global names are quite convenient for our purposes here, as they simplify the identification of components, whereas local names would require paths to be used instead.

2.3 Querying XML

In order to query "the web" — more precisely, a collection of XML documents on the web — we need a query language. Each query will be applicable to some XML documents and result in new XML documents. We will follow an approach close to XML-QL [Deutsch et al. 1999].

The basic ingredients for queries are variables, constants and expressions. Variables are elements of some a priori defined set Var. We use the convention to write such variables with a leading \$-symbol. Constants are all the values in some $dom(t)$ for some type t, and expressions are built out of variables and constants using operators defined for the various types. As our focus here is on fragmentation rather than on querying, we dispense with going into details of such expressions.

On this basis we can now define attribute conditions, element patterns, element conditions, comparison conditions, document patterns and queries.

—An *attribute condition* has the form `attribute` θ `expression`, where `attribute` is the name of an attribute and θ is a comparison operator, i.e., $=$, $\neq$, $<$, $\leq$, $>$, or $\geq$. Attribute conditions are used in two ways: to bind variables to values satisfying the condition, or to bind the attribute to the value determined by the expression.

—An *element pattern* has the form $\langle$ `name attribute_conditions` $\rangle$ `element_patterns` $\langle / \rangle$, where `name` is the name of an element, `attribute_conditions` is a sequence of attribute conditions with all left hand sides being attributes on the given element, and `element_patterns` is either a variable or a sequence of element patterns.

—An *element condition* has the form `element_pattern IN document` with an element pattern on the left and a document pattern or a URL on the right. If it is a URL, it is assumed that this is the URL of an XML document.

—An *comparison condition* has the form `variable` θ `expression` with a variable, a comparison operator, and an expression.

—A *document pattern* is either a variable or has the form $\langle$ `name` $\rangle$ `query` $\langle / \rangle$.

—A *query* has the form `CONSTRUCT construction_patterns FROM conditions`, where the first part `construction_patterns` is a non-empty sequence of element and document patterns, and `conditions` is a non-empty sequence of element and comparison conditions. Note that in the construction patterns the attribute conditions bind attributes, whereas in the conditions they bind the variables.

The nesting of element names in element conditions can be abbreviated by paths. Then we can use regular expressions for these tasks and also regular expressions for tags in these tasks. We omit descriptions of such a path language [Abiteboul et al. 2000]. In the end this may lead to using the language XPath.

Example 2.2. The following is an example of a simple query applicable to an XML document with the DTD from Example 2.1. Assume that "xyz" is the URL of such a document.

```
⟨ wines ⟩
   CONSTRUCT
               ⟨ wine ⟩
                     ⟨ name ⟩ $N ⟨ /name ⟩
                     CONSTRUCT
                           ⟨ year ⟩ $Y ⟨ /year ⟩
                     FROM ⟨ db ⟩ ⟨ wine ⟩ $W ⟨ / ⟩ ⟨ / ⟩ IN "xyz",
                           ⟨ name ⟩ $N ⟨ /name ⟩ IN $W,
                           ⟨ year ⟩ $Y ⟨ /year ⟩ IN $W
               ⟨ /wine ⟩
   FROM ⟨ db ⟩ ⟨ wine ⟩ ⟨ name ⟩ $N ⟨ / ⟩ ⟨ / ⟩ ⟨ / ⟩ IN "xyz"
⟨ /wines ⟩
```

Queries are evaluated by evaluating the conditions on the identified XML documents. This leads to a set of bindings for the variables, which are then used in the construction patterns. Thus, a document pattern of the form ⟨ `name` ⟩ `query` ⟨ / ⟩ will result in a new XML document as required.

For our purposes here we want to store these documents at some URL. Therefore, we need even a more general form `New_URL(variable`, ⟨ `name` ⟩ `query` ⟨ / ⟩), which will result in a new XML document and simultaneously bind `variable` to its URL-address. In the following we shall also allow this generalised form to be used as a document pattern.

Similarly, we may use `New_ID(variable)` to create new identifiers and bind `variable` to them. This can be used, whenever we expect an expression of type *ID*.

Example 2.3. The following is another example of a query applicable to an XML document with the DTD from Example 2.1. This query creates references from the new XML document to the original one.

```
New_URL("new", ⟨ wines ⟩
   CONSTRUCT
               ⟨ wine id = New_ID($J) refers_to = "xyz".$I ⟩
                     ⟨ name ⟩ $N ⟨ /name ⟩
                     CONSTRUCT
                           ⟨ year ⟩ $Y ⟨ /year ⟩
                     FROM ⟨ db ⟩ ⟨ wine ⟩ $W ⟨ / ⟩ ⟨ / ⟩ IN "xyz",
                           ⟨ name ⟩ $N ⟨ /name ⟩ IN $W,
                           ⟨ year ⟩ $Y ⟨ /year ⟩ IN $W
               ⟨ /wine ⟩
   FROM ⟨ db ⟩ ⟨ wine w-id = $I ⟩ ⟨ name ⟩ $N ⟨ / ⟩ ⟨ / ⟩ ⟨ / ⟩ IN "xyz"
⟨ /wines ⟩)
```

3. FRAGMENTATION OPERATIONS

Let us now look at the possible fragmentation of XML documents. We follow the work in [Schewe 2002] and distinguish between three different kinds of fragmentation. Split fragmentation results in new identifiers and references, whereas horizontal and vertical fragmentation result from the corresponding operations on the RDM.

3.1 Split Fragmentation

The splitting operation originates from work on object oriented databases. It simply takes a complex expression inside a class definition and replaces it by a reference to a new class.

In the context of XML it is convenient to assume that the root-element has a defining expression of the form $(n_1 \uplus \cdots \uplus n_k)^*$, so that we could refer to each n_i as a class. Splitting would thus result in a new class n_{k+1}, and in some element we would now reference to this new class. We may even think of placing n_{k+1} into a completely new document with a new created URL-address, in which case we would obtain external references.

Let us consider the first case. If we want to split away elements named n, we replace n at the corresponding position in the DTD by a new element name n' with the declaration $\langle$ `!ELEMENT` $n'\epsilon\,\rangle$. Furthermore, we add an attribute declaration $\langle$ `!ATTRIBUTE` n' reference_to_n `REF` n #`REQUIRED` $\rangle$.

We can leave the element definition for n as it is, but add an attribute declaration $\langle$ `!ATTRIBUTE` n n-id *ID* #`REQUIRED` $\rangle$. Furthermore, if the defining expression for the root is $(n_1 \uplus \cdots \uplus n_k)^*$, we replace it by $(n_1 \uplus \cdots \uplus n_k \uplus n_{k+1})^*$.

If we have n^0 instead of n, we can replace it directly by n', but in this case do not require the reference reference_to_n to be mandatory. If we have n^* or n^+ instead of n, we can replace it directly by n', but in this case define the type `REF` n for the reference reference_to_n. However, in all these three cases we could still use $(n')^0$, $(n')^*$ or $(n')^+$, respectively, and not change the attribute definition for reference_to_n.

The construction of a new XML document can be achieved by a query, though the next example shows that this is still cumbersome.

Example 3.1. Refer to Example 2.1. One of the easiest examples for splitting is to separate the names of wines from the wines themselves. For simplicity assume that the DTD contains

$\langle$!ELEMENT wine (name, rest) $\rangle$
$\langle$!ELEMENT rest (year^0, (grape* $\uplus$ grapes)^0) $\rangle$

Otherwise, we have to consider four different cases for the combinations of year, grape and grapes. Now we can define

```
⟨ db ⟩
    CONSTRUCT
        ⟨ wine w-id = $I producer = $P price = $Q ⟩
            ⟨ name_ref ref_to_name New_ID($N) / ⟩
            ⟨ rest ⟩ $R ⟨ / ⟩
        ⟨ /wine ⟩
        ⟨ name n-id = $N ⟩ $M ⟨ / ⟩
    FROM
        ⟨ db ⟩ ⟨ wine w-id = $I producer = $P price = $Q ⟩
            ⟨ name ⟩ $M ⟨ / ⟩
            ⟨ rest ⟩ $R ⟨ / ⟩
        ⟨ / ⟩ ⟨ / ⟩ IN "xyz"
        ⋮
  ⟨ /db ⟩
```

The dots indicate the parts dealing with carrying over vineyard- and region-elements to the new document.

The process for the second case is similar. In this case we have to create a new DTD with the root element definition ⟨ `!ELEMENT` root_new $(n)^*$ ⟩. Furthermore, the attribute definition for reference_to_n has to change to ⟨ `!ATTRIBUTE` n' reference_to_n "new".`REF` n `#REQUIRED` ⟩, where "new" is the URL of the new created document.

3.2 Horizontal Fragmentation

In [Schewe 2002] two versions of generalising horizontal fragmentation from the RDM to the object oriented case have been discussed. The first version addresses horizontal fragmentation on the level of classes, whereas the second one addresses the problem on the level of bulk types inside the structure definition of classes. However, at the end it turns out that the second version only leads to a fragmentation, if it is followed by a splitting fragmentation. In this case, however, the same result could be achieved by applying the splitting first.

Therefore, we will concentrate only on the first version. In the XML context this means to assume that the root-element has a defining expression of the form $(n_1 \uplus \cdots \uplus n_k)^*$. Horizontal fragmentation then aims at replacing one of these n_i by $(n_{i1} \uplus \cdots \uplus n_{ik_i})^*$ such that in all documents the result of the fragmentation would correspond to renaming each n_i to exactly one of the n_{ij}.

This can be realised by selection queries. Using queries on XML for fragmenting elements n_i we would simply need element conditions of the form ⟨ n_i ⟩ $\$N_i$ ⟨ / ⟩ and further conditions $\varphi_1, \dots, \varphi_n$ on the variables used in these conditions such that their disjunction is always true, whereas the conjunction of two of them is always false.

EXAMPLE 3.2. We want to fragment the wines in the schema of Example 2.1 into cheap wines with a price below 10 dollars and expensive wines. This horizontal fragmentation can be achieved by the following selection query:

```
⟨ db ⟩
   CONSTRUCT
     ⟨ cheap_wine w-id = $I producer = $P price = $Q ⟩ $W ⟨ / ⟩
   FROM
     ⟨ db ⟩ ⟨ wine w-id = $I producer = $P price = $Q ⟩ $W ⟨ / ⟩ ⟨ / ⟩ IN "xyz"
          $Q < 10
   CONSTRUCT
        ⟨ expensive_wine w-id = $I producer = $P price = $Q ⟩ $W ⟨ / ⟩
   FROM
        ⟨ db ⟩ ⟨ wine w-id = $I producer = $P price = $Q ⟩ $W ⟨ / ⟩ ⟨ / ⟩ IN "xyz"
          $Q ≥ 10
     ⋮
⟨ /db ⟩
```

Same as in Example 3.1 the dots indicate the parts dealing with carrying over vineyard- and region-elements to the new document.

We would also like to separate the document into several new ones. This can be easily achieved by creating new URLs. Same as for splitting fragmentation it is possible to create separate documents in two steps: first reorganise the XML document as described above, then separate the document into a collection of documents.

3.3 Vertical Fragmentation

Vertical fragmentation in the RDM corresponds to projecting to subsets of attributes. For an XML-element vertical fragmentation can be defined, if the defining expression is a sequence. So let us assume to be given an XML-element named n that is defined by an expression of the form $n_1, \dots, n_k$. Let $\{n_1, \dots, n_k\} = X_1 \cup \cdots \cup X_\ell$ be a decomposition — not necessarily a partition — of the set of names of successor elements of n. Then we replace n by $n'_1, \dots, n'_\ell$ with new elements n'_i ($i = 1, \dots, \ell$). Furthermore, the defining expressions for the new elements n'_i have the form $n_{i1}, \dots, n_{ik_i}$ for $X_i = \{n_{i1}, \dots, n_{ik_i}\}$.

As to the attributes of n, assume that these are $a_1, \dots, a_m$. Then decompose also the set of attributes $\{a_1, \dots, a_m\} = A_1 \cup \cdots \cup A_\ell$ and turn the attributes in A_i into attributes of n'_i.

This can be realised by projection queries. Using queries on XML for fragmenting elements n_i we would simply need element conditions of the form $\langle n \rangle \langle n_{ij} \rangle \$N_i \langle / \rangle \langle / \rangle$ for all $i = 1, \dots, \ell$ and $j = 1, \dots, k_i$. Note that splitting cannot be expressed as a special form of vertical fragmentation.

Example 3.3. The following projection query defines a vertical fragmentation of the element wine from Example 2.1 into new elements wine and wine-info:

```
⟨ db ⟩
    CONSTRUCT
        ⟨ wine w-id = $I price = $Q ⟩
            ⟨ name ⟩ $N ⟨ / ⟩
            ⟨ year ⟩ $Y ⟨ / ⟩
        ⟨ / ⟩
        ⟨ wine-info for_wine = $I producer = $P ⟩
            ⟨ grapes ⟩ $G ⟨ / ⟩
        ⟨ / ⟩
    FROM
        ⟨ db ⟩ ⟨ wine w-id = $I producer = $P price = $Q ⟩
            ⟨ name ⟩ $N ⟨ / ⟩
            ⟨ year ⟩ $Y ⟨ / ⟩
            ⟨ grapes ⟩ $G ⟨ / ⟩
        ⟨ / ⟩ ⟨ / ⟩ IN "xyz"
      ⋮
⟨ /db ⟩
```

Note that we introduced a new attribute for_wine for the new element wine-info. In the DTD this would lead to the attribute definition ⟨ !ATTRIBUTE wine-info for_wine REF wine #REQUIRED ⟩. We will discuss below why this reference is needed.

There are two problems to be discussed with this form of vertical fragmentation. The first one concerns order. When defining the new element n'_i, we may take the element names in X_i in any order, whereas sequences in XML elements are ordered. However, from a database point of view the content is not affected by the order, and any order on X_i is acceptable.

The second problem is more serious. The vertical fragmentation of n into $n'_1, \dots, n'_\ell$ should be reversible. However, this is only possible, if we can define a join query on the fragmented document, which results in the original document.

The first solution is to ensure that the intersection of the X_i contains a key. Though we did not formally introduce keys here, it is clear what is meant by a key: the value of the element n is uniquely determined by this key. An obvious disadvantage would be that the sets X_i must have a significant overlap.

Another solution would be to exploit identifiers and references. Choose one of the new elements $n'_1, \dots, n'_\ell$, say n'_1 and give it an attribute n_1-id of type *ID*. If such an attribute already exists due to the distribution of the attributes of n to the elements $n'_1, \dots, n'_\ell$, this is not necessary. For the other elements n'_i $(i = 2, \dots, \ell)$ we define a new attribute for_n'_1 with type `REF` n'_1.

Same as for horizontal fragmentation we would like to separate the document into several new ones, which can be achieved by creating new URLs in the projection query. Alternatively we can use a two-step process for this.

4. FRAGMENTATION DECISIONS

According to our discussions in the previous section we can think of fragmentation of XML documents as a three-phase process: first use splitting fragmentation to define new subelements of the root-element of an XML document, then define fragments using horizontal and vertical fragmentation techniques without introducing a new XML document, finally fragment the XML document into several new ones.

The third step has to be done in connection with the allocation of fragments to the nodes of a network. As we may easily recombine documents, each node should receive exactly one of the new XML documents. As we do not deal with allocation in this article, we will not further discuss this step.

We also do not put any effort into discussing the first step. As discussed in the previous section the splitting operation is mainly used for preparing the horizontal or vertical fragmentation. Thus, if an element is to be fragmented, we turn it first into a subelement of the root-element. If no horizontal nor vertical fragmentation is planned, there is no need for splitting.

Thus, in principle we can reduce ourselves to assume that the root-element is defined by an expression of the form $(n_1 \uplus \cdots \uplus n_k)^*$, and that we intend to fragment some of the elements n_i. However, we will see that this restriction is not really needed, as we will first define a physical view of XML documents, which can be used to describe a relational or network-based implementation.

4.1 A Physical View of XML Documents

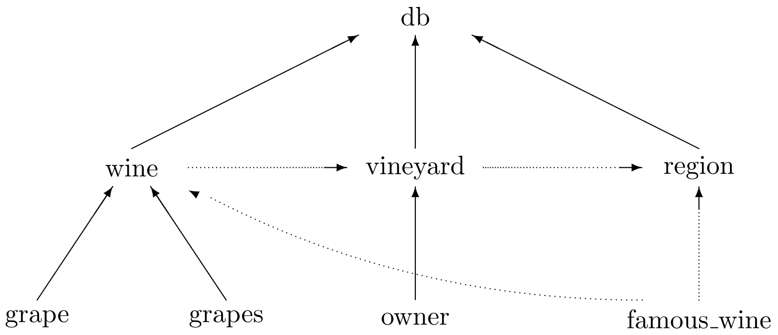
Decisions on fragmentation and even more on allocation depend to some non-negligible degree on the physical implementation and the access structures. Therefore, let us briefly discuss how XML documents could be realised using relational (or also network) database technology.

There are several approaches to storing XML documents in relational databases [Bohannon et al. 2002; Deutsch et al. 1999; Florescu and Kossmann 1999; Schmidt et al. 2000; Shanmugasundaram et al. 1999], which all there merits and limitations. In particular, some information about the XML document, e.g. dependencies, might get lost in the relational representation. In our context we use a relational representation only to obtain a better estimate for query and transaction processing costs. Such a cost model including a heuristic horizontal fragmentation algorithm that addresses cost optimisation has been worked out in detail in [Ma 2003].

According to the definition of a schema in Section 2 each element is reachable by a unique path from the root — ignoring for the moment the references. Even if the same element name can be used in different paths we will refer to an element is being identified by a path. Globally unique element names will be helpful to underline this convention.

We start defining a directed graph for an XML DTD (as in Section 2) with two types of edges. The nodes of the graph correspond to elements. Such nodes are associated with a relation schema, i.e., with a set of attributes. Subelements that may occur at most once are used to define this relation schema. A solid edge from n' to n occurs, if n' is a subelement of n that may occur more than once. If only a regular expression is given, we create a new element. All other subelements are subsumed in the associated relation schema. A dotted edge from n to n' occurs, if n contains a reference to n'. In case the element n has an attribute of type `REFS` n', we create a new node with dotted links to n and n' and an empty associated relation schema. Attributes that are not references are subsumed in the associated relation. We will call such a graph the associated graph of the *XML document*.

Example 4.1. Let us consider the DTD from Example 2.1. This defines the following directed graph:



The associated relations are the following: db = { created }, wine = { w-id, name, year, price }, grape = { grape }, grapes = { grape, percentage }, vineyard = { v-id, established, name, area }, region = { r-id, name }, owner = { name }. The domains for these attributes are as defined in the DTD.

We can treat the edges as in the network model and define the head of an edge as the owner of the edge and the tail as the member. We can then imagine ring or spider structures to implement the edges.

If we are looking for a relational implementation, we can add key attributes to all relations that are owners of edges, unless these are already defined, and add these key attributes as foreign keys to the relations of the members. We can take artificial keys such as db-id, w-id, etc. with domain *ID*. Then we obtain the following relations: db = { db-id, created }, wine = { w-id, name, year, price, db-id, v-id }, grape = { grape, w-id }, grapes = { grape, percentage, w-id }, vineyard = { v-id, established, name, area, db-id, r-id }, region = { r-id, name, db-id }, owner = { name, v-id }, and famous_wine = { w-id, r-id }.

In case of a relational implementation the navigation along edges is realised by relational joins.

4.2 Basic Queries and Fragmentation

Queries in XML consist of two parts: binding variables to XML elements and using these bindings to construct new XML elements. For the purpose of fragmentation it is only relevant to consider the first part. Thus, we consider the `FROM`-conditions in queries.

Such conditions are either element conditions or comparison conditions. The element conditions refer to a path and thus affect one or more of the relations in the graph defined in the previous subsection. Following the good old tradition of algebraic query optimisation we can assume that some of the comparison conditions can be evaluated on these relations.

Thus, we may derive basic queries from a given query. Such a basic query consists of selection and projection conditions on the relations in the associated graph. More precisely, a *basic query* is defined by a subgraph of the associated graph of the XML document plus selection predicates and projection subrelations for all the relations associated with the graph.

A basic query is interpreted as applying first all the selections on the relations, then all projections and finally all joins defined by the edges in the graph. As known from the RDM we can define a query tree for each basic query. Furthermore, we can assume that the known techniques of query optimisation (SIP graphs, algebraic optimisation, etc.) have been applied to define these query trees.

As to horizontal fragmentation we may thus apply the techniques from the relational model [Özsu and Valduriez 1999]. We consider only the most important and most frequent queries and derive the basic queries for them. This gives us a set of basic queries, each of which determines a set of selection predicates and frequencies for these selections. For a relation in the associated graph — and thus for an element n in the XML document — we consider the selection predicates $\varphi_1, \dots, \varphi_k$.

We then consider the 2^k formulae $\varphi_1^* \wedge \cdots \wedge \varphi_k^*$, where φ_i^* is either φ_i or $\neg\varphi_i$ except those that are inconsistent. These determine selection predicates on n and thus fragments for horizontal fragmentation. For vertical fragmentation we consider, which attributes in the relation to n are affected by the basic queries, and define the affinity matrix for the attributes.

Details of these relational techniques are described in [Özsu and Valduriez 1999]. As we now use a relational representation of XML documents, these techniques can be applied directly to determine the fragments.

5. CONCLUSION

In this article we approached the fragmentation of XML documents. The rationale behind this work is that XML is designed to support data management on the web. Thus, it can be considered as a datamodel, but the databases we should have in mind are distributed.

We generalised horizontal and vertical fragmentation techniques from the RDM to XML and introduced a third simple fragmentation operation based on splitting. Then distribution design can be considered as a three-phase process:

(1) Use splitting fragmentation to define new subelements of the root-element of an XML document.
(2) Define fragments using horizontal and vertical fragmentation techniques without introducing a new XML document.
(3) Fragment the XML document into several new ones and allocate them to the nodes of a network.

The determination of a reasonable set of fragments can be based on a physical view of XML documents. Then the relational techniques can be applied.

The work presented in this article, however, is only a first step towards distribution of XML documents. We believe that the open problem of allocation will raise serious problems, which cannot be solved by easy transformation of the relational techniques.

REFERENCES

Abiteboul, S., Buneman, P., and Suciu, D. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers, 2000.

Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. The LOREL query language for semi-structured data. *International Journal on Digital Libraries* 1 (1): 68–88, 1997.

Bellatreche, L., Karlapalem, K., and Simonet, A. Algorithms and support for horizontal class partitioning in object-oriented databases. *Distributed and Parallel Databases* 8 (2): 155–179, 2000.

Bohannon, P., Freire, J., Roy, P., and Simeon, J. From XML-schema to relations: A cost-based approach to XML storage. In *Proceedings of the International Conference on Data Engineering.* San José, California, pp. 64–75, 2002.

Buneman, P., Davidson, S., Fernandez, M., and Suciu, D. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory.* Lecture Notes in Computer Science, vol. 1186. Springer-Verlag, Delphi, Greece, pp. 336–350, 1997.

Buneman, P., Davidson, S., Hillebrand, G., and Suciu, D. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* Montréal, Canada, pp. 505–516, 1996.

Ceri, S. and Pelagatti, G. *Distributed Databases Principles and System.* McGraw-Hill, New York, 1984.

Chu, P.-C. A transaction oriented approach to attribute partitioning. *Information Systems* 17 (4): 329–342, 1992.

Chu, P.-C. and Ieong, I. T. A transaction-based approach to vertical partitioning for relational databases. *IEEE Transactions on Software Engineering* 19 (8): 804–812, 1993.

Cluet, S., Delobel, C., Siméon, J., and Smaga, K. Your mediators need data conversion! In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* Seattle, Washington, pp. 177–188, 1998.

Cornell, D. and Yu, P. A vertical partitioning algorithm for relational databases. In *Proceedings of the International Conference on Data Engineering.* Los Angeles, California, pp. 30–35, 1987.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. A query language for XML. *Computer Networks* 31 (11-16): 1155–1169, 1999.

Deutsch, A., Fernandez, M., and Suciu, D. Storing semistructured data with STORED. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* Philadelphia, Pennsylvania, pp. 431–442, 1999.

Florescu, D. and Kossmann, D. Storing and querying XML data using an RDBMS. *Bulletin of the Technical Committee on Data Engineering*, 1999.

Fuchs, M., Maloney, M., and Milowski, A. Schema for object oriented XML. http://www.w3c.org/TR/NOTE-sox, 1998.

Goldfarb, C. F. and Prescod, P. *The XML Handbook.* Prentice Hall, New Jersey, 1998.

Lin, X., Orlowska, M., and Zhang, Y. A graph-based cluster approach for vertical partitioning in databases systems. *Data & Knowledge Engineeering* 11 (2): 151–170, 1993.

Ma, H. *Distribution Design in Object Oriented Databases.* M.S. thesis, Massey University, 2003.

Malinowski, E. and Chakravarthy, S. Fragmentation techniques for distributing object-oriented databases. In *Proceedings of the International Conference on Conceptual Modeling*, D. W. Embley and R. C. Goldstein (Eds.). Lecture Notes in Computer Science, vol. 1331. Springer, pp. 347–360, 1997.

Özsu, M. T. and Valduriez, P. *Principles of Distributed Database Systems.* Alan Apt, New Jersey, 1999.

Schewe, K.-D. Fragmentation of object oriented and semi-structured data. In *Proceedings of the 5th International Baltic Conference on Databases and Information Systems.* Tallinn, Estonia, pp. 1–14, 2002.

Schmidt, A., Kersten, M., Windhouwer, M., and Waas, F. Efficient relational storage and retrieval of XML documents. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases.* Lecture Notes in Computer Science, vol. 1997. Springer-Verlag, Dallas, Texas, pp. 137–150, 2000.

Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., De Witt, D. J., and Naughton, J. F. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases.* Edinburgh, Scotland, pp. 302–314, 1999.

Tamhankar, A. M. and Ram, S. Database fragmentation and allocation: An integrated methodology and case study. *IEEE Transactions on Systems Management* 28 (3): 194–207, 1998.

Tatarinov, I., Ives, Z., Halevy, A., and Weld, D. Updating XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* Santa Barbara, California, pp. 413–424, 2001.

World Wide Web Consortium (W3C). XQuery. http://www.w3c.org/TR/xquery.

World Wide Web Consortium (W3C). XML Schema. http://www.w3c.org/TR/xmlschema-0, http://www.w3c.org/TR/xmlschema-1, http://www.w3c.org/TR/xmlschema-2, 2001.