Integrated Querying of XML Data in RDBMSs

Albrecht Schmidt Stefan Manegold Martin Kersten

CWI Amsterdam
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

first.last@cwi.nl

ABSTRACT

This paper proposes a way to integrate cleanly relational databases and XML documents. The main idea is to draw a clear line of demarcation between the two concepts by modelling XML documents as a new atomic SQL type. The standardised XML tools like XPath, XQuery, XSLT are then user-defined functions that operate on this type. Well-defined interoperability is guaranteed by, on the one hand, defining a standard way to markup SQL relations as XML documents and, thus, to make them accessible to the XML tools; on the other hand, XPath and XQuery queries run against the XML portion of the database can use the same predefined schema to make their results accessible to the SQL language for further processing. Additionally, a method for set-oriented evaluation of regular path expressions is presented that integrates into our implementation framework.

Keywords

XML, Database, Data Integration, Query Optimisation

1. INTRODUCTION

So far, a plethora of work on storing and querying XML documents in relational, object-relational, object-oriented or special-purpose native XML databases has become available (see, *e.g.*, [7, 10, 15, 16]). Most of these proposals focus on the technical or implementation aspects of queries and tend to neglect the modelling and software-engineering aspects. The contribution of this paper is to build on already known physical XML-to-relational mappings and to propose a sound model of XML databases so that essential database design principles are adhered to. We introduce a perspective of XML documents that allows for the co-existence with relational tables in the same database without violating the data independence principle, which is often a point of concern in the previous work cited above. The main idea to achieve this is to draw a strict line of demarcation between the XML and the SQL world and by allowing data interchange between them only through well-defined interfaces. From a SQL point of view, the XML documents in the database are all instances of an atomic datatype *XML document*; the inner structure of XML values is accessed and queried through user-defined functions which implement the XPath and XQuery specifications. To extract SQL tables from XML documents through XPath or XQuery, queries have to return results that adhere to an XML Schema [17] that is supplied by the DBMS and also make use of a special namespace; if these rules are obeyed, then the results are automatically made available as SQL tables or SQL views without any further user interaction. In the reverse direction, to access SQL tables from XQuery, we use a standard publishing technique from the literature [14]. In fact, for reasons of simplicity, transparency and efficiency, we require that the schema of published relations in XML is the same as that of the extracted XML data mentioned above. Should users desire data to be in a different format, they can use (indexed) view mechanisms to derive and use differently structured data.

Once the authors of this paper embarked on the approach of clear separation just described, the next question was how to make best use of the SQL-based infrastructure that was already in place like the query optimiser and query execution engine; building on them would be preferable to having to implement facilities for XPath and XQuery processing from scratch, having to make fundamental modifications to the SQL engine or duplicating functionality. These approaches would be objectionable from a software engineering and manpower point of view. While implementing these ideas, it turned out that one of the greatest technical challenges was the presence of wildcards in path expressions. We propose to tackle it with so-called *path summaries*, *i.e.*, structural summary information that we gather while shredding documents when we bulkload them into the database [12]. This technique can enable large scale processing in many practically interesting cases.

The rest of this paper is structured as follows: After reviewing some related work, we present a system architecture that implements our proposal. We then embark on outlining how we can make use of the SQL engine to query XML documents without implementing a dedicated XQuery processor from scratch; special emphasis is laid on the evaluation of regular path expressions. Finally, we conclude with a summary and an outlook on future work.

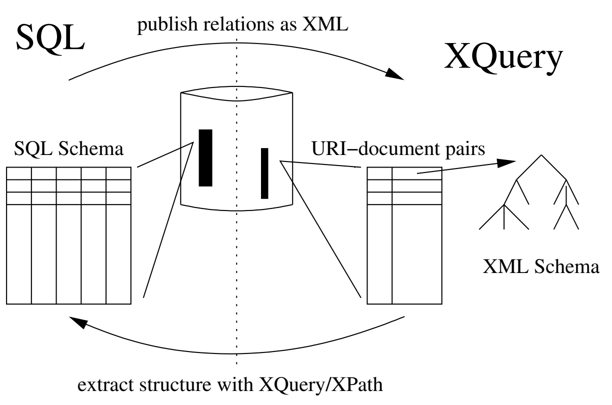


Figure 1: System architecture

2. RELATED WORK

From the plethora of XML research literature, the works on publishing SQL tables as XML documents [8, 14] are of particular relevance in our context, since they provide the basis for accessing SQL relations from XPath and XQuery. On the other hand, to query XML documents with a SQL optimiser and engine, we use the storage schema presented in [13]. The technique to eliminate path expressions is akin to the 'path expansion' technique for graph databases as deployed in Lore [11]; however, our technique is XML-specific, allows complete compile-time optimisation and does not require fixed-point calculations. Although we make use of previous work, the approach presented in this paper is geared towards removing implicit constraints inherent in many techniques that use relational technology to store and query XML and that often violate physical data independence principle. In [9], the authors present a general method to represent relational tables as XML documents; in the sequel, we assume that our system implements an approach such as this one.

3. SYSTEM ARCHITECTURE

Figure 1 sketches the architecture of our system. First we focus on the XML part on the right side of the figure. From a user's point of view, XML documents are stored as pairs of Uniform Resource Identifiers (URIs) and black-box XML documents, which may only be accessed through XPath and XQuery but not SQL. Internally, the documents are stored in a shredded schema that allows fast associative access with the relational algebra of the SQL engine; to achieve this we chose to implement the approach of [13]. Additionally, the native SQL tables on the left side of the figure are exported as XML views and may be queried in XPath and XQuery just like the XML documents on the right side. Details of the approach are described in the next section.

The left half of the figure is the SQL universe with its relational tables, views, triggers *etc.* that can be used as usual. In addition to that, the URI-document table can be queried with user-defined functions that implement XQuery and XPath: selections on the URI column identify tuples of interest to which expressions in XML query languages are applied. Should the user want to query the results in SQL, then the thus extracted information must adhere to a

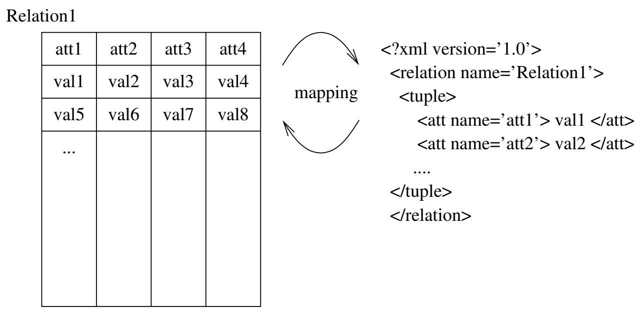


Figure 2: Translation between SQL tables and XML documents

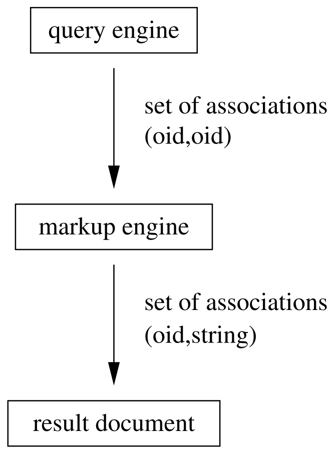


Figure 3: Data Flow in Query Processing

predefined schema which enables automatic conversion into SQL tables. The translation process is illustrated in Figure 2. Note that the translation process places restrictions on the document structure which we found useful to guarantee something like a 1 : 1 mapping; it turned out that a true bijective (1 : 1) mapping is not achievable due to numerous incompatibilities between relations and XML documents. We do not have the space to enumerate and discuss them in detail but they range from whitespace handling and font encodings to attribute naming conventions. In practise, the most promising approach seemed to be to introduce restrictions that should help avoid problematic cases before they can occur. Efforts like Canonical XML [4] should be a good starting point for future research. For our purposes, it should be sufficient to assume that the system implements a method to map relations to XML documents as the arrows in Figure2 indicate; such a method is presented, for example, in [9].

Note that, from the SQL side, the XML documents themselves are only identifiable through their URI. This is reasonable for two basic reasons: (1) Since XQuery [6] is document-oriented – XQueries are constructed with blocks of FLWR expressions of the form `FOR $x IN document(''foo.xml'')` *etc.* – and not document collection-oriented, this choice of representation captures the semantics of XQuery. The logical scheme of an XQuery processor is depicted in Figure 3.

(2) To be able to enforce the semantic and modelling integrity of the XML documents independently of the physical representation used in the storage engine, the documents need to be black boxes from a SQL point of view. In a later section, we briefly mention some implementation techniques we used to enforce the strict separation of the SQL and the XML part.

4. QUERY ALGEBRA

In this section, we discuss how documents stored in the XML part of the database can be queried with the relational algebra such as presented in [1] and be accessed with the machinery that the underlying storage engine provides [3]. Basic queries, *i.e.*, queries without wildcards in path expression, can be expressed by assigning the usual SQL bag semantics to the tables in which XML documents are stored and that were generated by the XML bulkload scheme procedure described in [12]. Unfortunately, the probably most salient feature of XML query languages, regular path expressions with wildcards, is not expressible this way. To overcome this lack of expressiveness of the underlying algebra, we now introduce a preprocessing technique which translates regular path expressions containing wildcards into the plain relation algebra given a structural summary of a database instance that was generated while the document was bulkloaded into the database. This approach enables the use of existing query optimisers and execution engines on XML queries without extensions or modifications. Note that by building on the relational algebra, we primarily aim at bulk retrieval; other types of queries are supported as well but maybe not executed as efficiently as bulk queries. A further design advantage we would like to mention is that we inherit the simplicity and minimality of the relational algebra along with its rule set for query transformation and optimisation.

4.1 Features of XML Query Languages

Although this paper is not the place to discuss the requirements and features of XML query languages in detail, it is still useful to look at what makes them different from SQL [2], which is the standard interface to query processors based on the relational algebra and the basis for our implementation. Then we analyse the requirements to identify how the algebra we present can be extended to act as a fully functional low-level implementation language for XQuery and XPath.

In [5], the authors define some general requirements of XML query languages. These requirements reflect the tendency to extend the role of query languages beyond what they have been in past settings. For example, relational databases are queried through SQL or Query By Example (QBE) interfaces, both of which are made for human users. The role of XML as a machine-readable data interchange format also necessitates a machine-readable version of the query language. Therefore, it makes sense to define more than one syntax for the logical query model to support both machine-readable and human-readable formats. Furthermore, to deserve the designation *query language* an XML query language has to be declarative; this means that it should describe queries on the logical level rather than by algebraically enforcing a particular strategy of evaluating the query.

On the technical level, a query language should be independent of protocols so that queries do not depend on the physical infrastructure of the World Wide Web or the database server. Since so much effort has been made to provide and standardise lowest common denominators like URIs, on which XML and its infrastructure depend, it is not desirable for the query language to depend on more than these lowest common denominators.

As one important use of query languages is the role of embedded languages in program code, an XML query language should provide a set of standard error conditions like exceptions to signal to the host application that expressions cannot be processed. This could be due to syntactical or logical errors in the expression or the unavailability or failure of external resources such as network or external functions.

Additionally, an XML query language should be extensible in the sense that it is open for additional functionality that goes beyond querying. Updates and transactions are critical for many applications but still not part of any standard. Since some XML data models can define infinite document instances, fixed point computations are useful for these cases; however a query language is only required to be defined for finite instances.

Although XQuery appears to be the current frame of reference in XML querying, a number of alternative approaches are available and may catch on in the future. Therefore, we do not pretend to present an algebra that implements all the features of XQuery but that rather can serve as a basis for an implementation of XQuery. So we try to focus on the features that separate XML query languages from relational query languages and show how to implement the former with a basic relational algebra and the additional information provided by our physical XML mapping. This means that the algebra which we present in the next subsection has two important properties: First, it is closed under composition, *i.e.*, the result of a sub-query can be bound to a variable in the enclosing query. Second, it is set-oriented, *i.e.*, it aims at processing of large data volumes.

The reader can easily verify that, despite these impedance mismatches, a SQL engine provides the functionality that is necessary to implement the features that were discussed in the previous paragraphs.

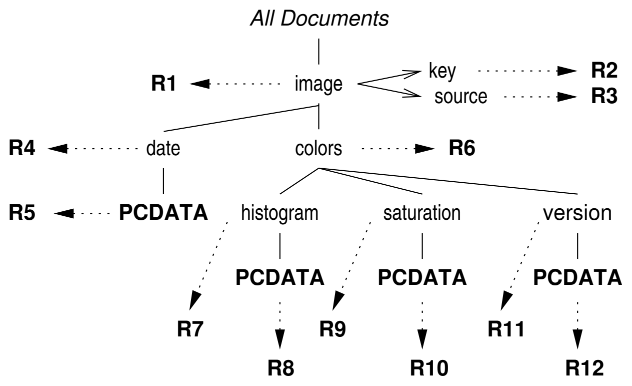
4.2 Example Database

We use the XML document displayed in Figure 4 as an example to illustrate the concepts we explain in the following subsections. The document is one example from a whole series of analyses done in the context of multimedia feature detection. They describe images and are output in XML format by programs called ‘detectors’ that extract features from raw image data.

Figure 5 displays the schema tree as maintained by the database engine [12]. In the sequel, it is useful to keep this tree in mind as the query rewriting ideas follow naturally from the *tree* shape of the schema tree. In our XML mapping all relations $\mathbf{R}_i$ are binary and contain the parent-child or node-attribute relationships in the XML syntax tree of the document.

```
<image key="18934" source="/cdrom/images1/23493.jpeg">
  <date> 999010530 </date>
  <colours>
    <histogram> 0.399 0.277 0.344 </histogram>
    <saturation> 0.390 </saturation>
    <version> 0.8 </version>
  </colours>
</image>
```

Figure 4: Example document



R1: /image **R3:** /image[source] **R5:** /image/date/PCDATA
R2: /image[key] **R4:** /image/date ...

Figure 5: Schema tree of example document

4.3 Overview of the Query Algebra

This subsection gives an overview of the query algebra by outlining its syntax. Figure 6 presents a listing of the grammar productions. The algebra is a simple extension of basic relational algebras [1] with function application, intersection and path expressions.

$$
\begin{array}{rcl}
\mathit{expr} & \rightarrow & (\mathit{expr}) \\
 & | & \mathit{pathexpr}\ (\mathit{expr}) \\
 & | & \sigma_p(\mathit{expr}) \\
 & | & \pi_A(\mathit{expr}) \\
 & | & \mathit{expr} \cup \mathit{expr} \\
 & | & \mathit{expr} \cap \mathit{expr} \\
 & | & \mathit{expr} \bowtie_p \mathit{expr} \\
 & | & \mathbf{map}_f(\mathit{expr}) \\
 & & \\
\mathit{pathexpr} & \rightarrow & \mathit{step}?\ \mathit{tag}?\ (\mathit{step}\ \mathit{tag}) *\ \mathit{attr}? \\
\mathit{step} & \rightarrow & /\ |\ // \\
\mathit{attr} & \rightarrow & [\mathit{tag}]
\end{array}
$$

Figure 6: Operators of the algebra

The semantics of most of the operations are standard and straight-forward. Like in many algebras in the database world, the functions are either singleton or binary. The selection operator $\sigma_p(R)$ filters out those tuples in a relation R for which the predicate p does not hold. The projection operator $\pi_A(R)$ only keeps the attributes contained in the set A from the tuples in the relation R. The binary operators $\cup, \cap$ and $\bowtie_p$ are the well-known union, intersection and equi-join operators, where p again is a predicate; $\mathbf{map}_f(R)$ applies the side-effect free function f to all tuples in R. It will mainly be used to convert tuples to XML notation and to cast the type of attributes. The salient feature of the algebra is the production *pathexpr*, which can be interpreted as follows: while the other operators work in the document tree in a horizontal manner, path expressions help to query the tree vertically, *i.e.*, along the tag hierarchies. Note that the set of operations we presented is not minimal. This means that it is possible to express certain operators by combinations of other operators. Also note that the algebra does not allow expressions to be substituted by database relations; this is done automatically by the query compiler. The only way to navigate through the hierarchies of documents is by means of path expressions.

Example. Consider the example document of the previous section in Figure 4, whose schema tree is displayed in Figure 5. Suppose we want to extract all histograms and the key of the corresponding image. The following expression could be used to do just that (since all relations are binary we use *hd* to denote the first component – the head – of the binary tuple and *tl* to denote the second component – the tail):

$$\mathit{image}([\mathit{key}] \bowtie_{hd=hd} (\mathit{colour}/\mathit{histogram}/\mathit{cdata}[\mathit{string}]))$$

This translates to the plain algebra in a straightforward manner:

$$\mathbf{R1} \bowtie_{tl=hd} (\mathbf{R2} \bowtie_{hd=hd} (\mathbf{R6} \bowtie_{tl=hd} \mathbf{R7} \bowtie_{tl=hd} \mathbf{R8}))$$

The structure of the plain algebra expression resembles that of the original query. For orientation, note the following way of reading the original query: *In the database, navigate to all nodes which carry an image tag. Then join the key attribute with the string found at the end of the paths along the tags colour and histogram.* Also note, that, in this case, all head-head and head-tail correspondences in joins happen to be 1 : 1 relationships. We only need two different kinds of join attributes in the query. If we follow a hierarchical path, we join parent-child relations on the OIDs (object identifiers) that refer to each other, this is denoted by $\bowtie_{tl=hd}$ following Monet speak [3]. If we want to combine objects with a common ancestor we use the join $\bowtie_{hd=hd}$ to compute the intersection with respect to the head elements.

We now turn our attention to regular path expressions and how they can be replaced with operators from the plain relational algebra.

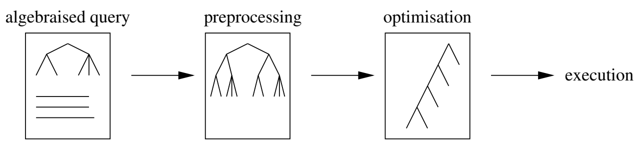


Figure 7: Phases of query compilation

4.4 Compilation of Regular Path Expressions

While Regular Path Expressions (RPEs) are one of the most powerful features of XML query languages, there is still research going on how to evaluate them efficiently in general settings. We now present a two-step evaluation scheme that (1) enables efficient execution of a restricted class of RPEs in the relational algebra and (2) opens up this class of RPEs to query optimisation.

Once a query front-end produces an algebraised version of the input query for example from an XQuery input, query execution consists of the steps outlined in Figure 7: during preprocessing all wildcards in regular path expressions are eliminated and replaced with join and union operations; the query can then be handed on to the conventional SQL query optimiser and execution engine. In the preprocessing step, we eliminate wildcards in regular path expressions by keeping track of the current context of a query node and replacing a wildcard with the paths that match the wildcard in the current context. The following algorithm eliminates all wildcards in path expressions from the input tree:

```
procedure eliminate ($context\ c, expr\ e$) : $expr$
  if $e$ is pathexpr then
    replace $e$ with union of all matching relations in $c$
    new context $c_n$ is union of $c$ and expanded $e$
  else
    the new context $c_n$ is concatenation of $c$ and $e$
  endif
  $\forall child\ e_c$ of $e$ : eliminate($c_n, e_c$)
end
```

Figure 8: Algorithm to eliminate regular path expressions schematically

Example. This example query is an extension of the previous one. Now we are not only interested in the colour histogram but also in all string data below the colour node:

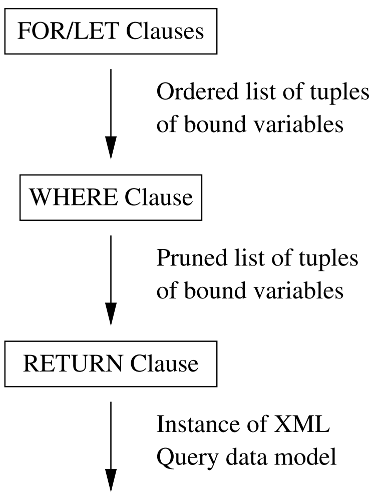
$$image([key] \bowtie_{hd=hd} (colour//cdata[string]))$$

Using the algorithm in Figure 8, this translates to:

$$\begin{aligned}\mathbf{R1} \bowtie_{tl=hd} (\mathbf{R2} \bowtie_{hd=hd} (&\mathbf{R6} \bowtie_{tl=hd} \mathbf{R7} \bowtie_{tl=hd} \mathbf{R8}\\ &\cup \mathbf{R6} \bowtie_{tl=hd} \mathbf{R9} \bowtie_{tl=hd} \mathbf{R10}\\ &\cup \mathbf{R6} \bowtie_{tl=hd} \mathbf{R11} \bowtie_{tl=hd} \mathbf{R12}))\end{aligned}$$

Following the previous example, the regular path expression *colour//cdata[string]* expands to the three paths

1. *colour/histogram/cdata[string]*,
2. *colour/saturation/cdata[string]*, and
3. *colour/version/cdata[string]*.



FOR/LET Clauses
Ordered list of tuples of bound variables
WHERE Clause
Pruned list of tuples of bound variables
RETURN Clause
Instance of XML Query data model

Figure 9: Data flow in XQuery

Note that once one wildcard in an RPE evaluates to more than one path, all its child-wildcards in the RPE possibly also evaluate to more than one path.

During the translation, new union statements are only introduced if not all paths are fully specified or otherwise non-unique. After the elimination of the wildcards, the query is optimised for efficient execution. Since our extended relational algebra is compiled to a plain relational algebra, relational optimisation techniques can be applied and the query is handed on to the SQL optimiser. Note also that the presence of path expressions makes query optimisation feasible for a larger class of queries than a plain translation of path expressions into joins, which would be functionally equivalent. This is because large numbers of joins tend to enlarge the number of optimisation and reordering opportunities beyond what current optimisers are capable of handling.

Figure 9 shows the data flow in an XQuery processor as anticipated by the designers of the language. In such an engine, path expressions come in two flavours: just as we found it useful to introduce strong and weak associations [13] to capture the semistructured nature of XML data, we also distinguish between strong and weak path expressions. Whereas strong path expressions are evaluated with join semantics, weak path expressions bear outer join semantics. For the purpose of this algebra, path expressions in the first phase of query execution, *i.e.*, the evaluation of FOR and LET clauses, are all strong, whereas those in the second part, *i.e.*, the RETURN clause, are all weak. Note that this restriction does not reduce the expressiveness of the algebra.

4.5 Some Implementation Issues

As we mentioned before, from a conceptual point of view, the only entry point to the XML documents in the database is the URI key of the binary relation. This means that there is exactly one document root that corresponds to this URI. To guarantee that the database obeys to this integrity constraint, we use the SQL mechanism of *triggers* to guarantee integrity of shredded data. When update operations are applied to the documents, it is, for example, necessary to enforce the following constraint: for each node that is deleted, the rules defined by the triggers must make sure that the corresponding sub-trees are deleted as well.

To guarantee the strict division between the SQL and the XML world on the one hand, but to still be able to use the same query processor for querying both on the other hand, we make sure that the relations in the database, which are either native SQL tables or XML indexes but never both, belong to different namespaces. Thus SQL tables cannot be accessed with XPath or XQuery expression and, vice versa, the internal structure of XML document cannot be accessed with pure SQL. If the database engine does not support namespaces the same effect can be achieved, for example, by either prefixing relation names with appropriate distinguishing strings or by putting them in different databases altogether, if this is supported. A final decision depends on the architecture of the underlying engine.

Of course, there still remains the issue how derived data in indexed views can be kept up to date efficiently; in the scheme we presented triggers do not work well across the SQL/XML boundary. Currently we consider this a research issue and future work.

5. CONCLUSION

We presented a method to integrate SQL tables and XML documents in the same database engine. The focal features of our proposal are that (1) it is logically sound, (2) it can be implemented with reasonable effort by building on existing SQL optimisers and execution engines, and, (3) that it is conceptually simple. From a SQL point of view, the main idea to achieve this was to model XML documents as a separate user-defined type with user-defined functions implementing XPath/XQuery functionality. From an XML point of view, the mapping between SQL tables and XML views was the guarantee for co-existence of the two schools of thought. To make the mappings possible, it was necessary to introduce a dedicated schema, for which the $1:1$ mapping could be realised between relations and XML documents. Furthermore, we presented a technique to eliminate wildcards in regular path expressions at compile time.

Concerning future work, we plan to work on query optimisation for XQuery. The execution plans generated during the path elimination may be hard to optimise since they may gain great complexity if many path wildcards are expanded. We also intend to work on more flexible XML storage strategies that automatically adapt the storage structure to the query profile.

6. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] American National Standards Institute. The database language SQL, 1986.

[3] P. Boncz and M. Kersten. MIL Primitives for Querying a Fragmented World. *VLDB Journal*, 8(2):101–119, 1999.

[4] J. Boyer. *Canonical XML Version 1.0*, March 2001. available at `http://www.w3.org/TR/xml-c14n`.

[5] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. XML Query Requirements. working draft, available at `http://www.w3.org/TR/xmlquery-req`, February 2001.

[6] D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery: A Query Language for XML, February 2001. available at `http://www.w3.org/TR/xquery`.

[7] L. Fegaras and R. Elmasri. Query Engines for Web-accessible XML Data. In *Proceedings of the International Conference on Very Large Data Bases*, 2001.

[8] M. Fernandez, A. Morishima, and D. Suciu. Efficient Evaluation of XML Middleware Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2001.

[9] M. Fernandez, W. Tan, and D. Suciu. SilkRoute: trading between relations and XML. *Computer Networks*, 33(1–6):723–745, 2000.

[10] C.-C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *Proceedings of the IEEE International Conference on Data Engineering*, page 198, 2000.

[11] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *Proceedings of the International Conference on Very Large Data Bases*, pages 315–326, 1999.

[12] A. Schmidt and M. Kersten. Bulkloading and Maintaining XML Documents. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2002)*, pages 407–412. ACM Press, 2002.

[13] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient Relational Storage and Retrieval of XML Documents (Extended Version). In *The World Wide Web and Databases – Selected Papers of WebDB 2000*, volume 1997 of *Lecture Notes in Computer Science*, pages 137–150, 2000.

[14] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proceedings of the International Conference on Very Large Data Bases*, pages 65–76, 2000.

[15] J. Shanmugasundaram, E. Shekita, J. Kiernan, R. Krishnamurthy, E. Viglas, J. Naughton, and I. Tatarinov. A General Technique for Querying XML Documents using a Relational Database System. *ACM SIGMOD Record*, 30(3), 2001.

[16] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *Database and Expert Systems Applications*, pages 206–217. Springer, 1999.

[17] The World Wide Web Consortium. XML Schema Part 0: Primer. available at `http://www.w3.org/TR/xmlschema-0/`, May 2001.