RAPID XML DATABASE APPLICATION DEVELOPMENT

Albrecht Schmidt
Aalborg University
9220 Aalborg Øst, Denmark
Email: al@cs.auc.dk

Kjetil Nørvåg
Norwegian University of Science and Technology
7491 Trondheim, Norway
Email: Kjetil.Norvag@idi.ntnu.no

Key words: XML, databases, prototyping, design

Abstract: This paper proposes a rapid prototyping framework for XML database application development. By splitting up the development process into several refinement steps while keeping the application programming interface stable, the framework aims at rapid implementation of a prototype with a well-defined interface and a subsequent implementation of more advanced concepts like business rules in several steps. The refinement process takes the form of incrementally adding domain-specific information to the application. This is achieved by transgressing from general-purpose XML tools that do not support the definition and enforcement of constraints to frameworks that support domain-specific models and constraints such as E/R modeling. We have employed this method in the development of an example application, and we give performance numbers that illustrate the incremental improvements of each step.

1 Introduction

Since XML assumed the role as the premier data exchange format on the Internet, application designers have increasingly been showing interest in coupling XML technologies and large scale data management techniques. One path to achieving this goal in the development of new Web services is to modularize tasks and to build on mature components: because Web services are accessed through a well-defined interface that hides the actual implementation of the service, it is possible to split the front-end, *i.e.*, the client applications, from the database back-end.

Internet information systems are usually implemented as complex multi-tier architectures. Virtually any such system that has to deal with significant amounts of data will utilize some kind of mass storage system, most probably a database management system (DBMS). The overall system architecture depends very much on the kind of services the mass storage back-end can deliver. Because, unfortunately, the implementation of this very back-end is a time-consuming task, it is desirable to split up development into several steps. The first step usually comprises the definition and export of an the interface that client applications can use. If a SOAP interface (Box et al., 2000) is to be implemented, standard XML tools can be leveraged; they greatly facilitate setting up a prototype that provides the necessary services but without taking into account issues like efficiency and consistency, which can be dealt with later. We refer to such an approach as *rapid prototyping*. According to (Kordon and Luqi, 2002) a prototype is *an executable model of a system that accurately reflects a chosen subset of its properties, such as display formats, computed results, or response times.* In the context of our work, this implies that the prototype implements an abstract programming interface (API) but uses only standard, non-performance oriented tools for the back-end, which is treated as a black-box. In subsequent steps, the back-end is then improved until it scales up to production levels.

Furthermore, prototyping (Kordon and Luqi, 2002) is a technique which is generally desirable in software engineering for a number of reasons. It helps to abstract from low-level details and to blend the different components of a system to work together. A prototype is then refined until it reaches production level. To summarize:

1. Prototyping helps to understand the requirements of a software systems early in the development process: unnecessary requirements can be removed or altered while other desiderata might be discovered.
2. Prototyping permits early feedback from users and

implementors alike; this can then be used for improvements in planning and implementation. Ideally, the process will result in a feedback loop.

3. A point that is particularly important is that prototyping eases the integration of subsystems. Since software systems, and especially Web-based ones, consist of various logical and physical layers, it is highly desirable to develop and improve the different subsystem independently of each other as far as possible.

In this paper we outline a framework for XML database application development which addresses these issues. In a stepwise refinement prototyping process, we move from general-purpose XML tools deployed in the first step to tools that allow domain modeling in the refinement steps. The first step in the development chain consists of using XQuery (Chamberlin et al., 2001) on a file-based storage backend. Subsequent steps include switching to a DBMS with automated XML-to-database mapping, annotations with domain knowledge, and, eventually, using modeling techniques to both ensure efficient data access as well as data integrity through the specification of constraints. Thus, prototyping goes through several refinement steps and employs more and more specific tools, which is made possible by increasingly drawing benefit from domain-knowledge. The final step is then a database which does not consist of automated mappings with surrogate identifiers anymore but of a E/R-type data model (Thalheim, 2000). To achieve this, we have defined a mapping language that ensures smooth interaction of XML tools and relational databases. We have employed this development process during development of an example application, and we give performance numbers and improvements for the prototype through the steps in the development process. The implementation was carried out in a number of student projects.

The organization of the rest of this paper is as follows. In Section 2 we give an overview of related work. Section 3 describes the general layout of our framework. In Section 4 we describe in detail the different steps of the development process. In Section 5 we present a number of measurements that reflect the performance characteristics of the different steps and hints at some trade-offs. to be considered. In Section 6 we discuss the use of the framework in the context of document-centric XML documents. Finally, in Section 7, we conclude the paper and outline topics for future research.

2 Related Work

The general validity of rapid prototyping for XML applications has been demonstrated in various industry projects (see, *e.g.*, (e-XMLmedia,)), although usually only one prototype is developed in order to demonstrate both the feasibility of the undertaking and the user interface. In our framework, this is equivalent to the first step as laid out below.

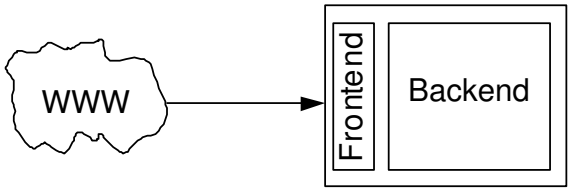


Figure 1: General setting of our research.

In (Orsini and Celentano, 2002), Orsini and Celentano propose a development environment that can aid data engineers in mapping between database schemas and XML DTDs. This process is essentially bidirectional: it enables data transfer between the two sides, and the generation of programs and DTDs for executing, validating and safe-guarding the data exchange process. Furthermore in (Florescu and A. Grünhagen, 2003), the authors present a language for implementing middleware functionality like Web services that could also play a role in the API specification that is part of our framework.

With respect to databases, there have been several studies on mapping from XML to relational tables, and how to query and store in a RDBMS based on these mappings. For example, in (Florescu and Kossmann, 1999), Florescu and Kossmann present mappings from XML to general relational tables; in (Schmidt et al., 2000), Schmidt *et al*. present a data and an execution model that allow for efficient storage and retrieval of XML documents in a relational database based on binary associations. The main problem of mapping from XML to relational tables, is in order to achieve good performance different mappings are needed for different data and workloads. In order to solve this problem, Bohannon *et al*. (Bohannon et al., 2002) developed a cost-based XML storage mapping engine that is based on models of XML schema, data statistics and workload tries to find the best mapping for a given application according to a cost model. In (Freire and Simèon, 2002) the authors propose an implementation framework for the implementation of these considerations. In (Shanmugasundaram et al., 1999), a mapping that 'imitates' E/R modeling on top of XML documents is presented; it is a variation of one of the mapping we also use in our implementation and performance study.

The reverse process, generating and publishing XML data from relational sources in addressed, for example, in the Agora system (Manolescu et al., 2000); there, XML is employed as the user interface

format, while relational tuples are used to represent the data inside the query processor. This approach resembles the final refinement step of our architecture. SilkRoute (Fernández et al., 2002) is a middleware system for publishing XML data from relational databases. The XML view is defined using a declarative query language. It accepts XML-QL queries over the XML view, and translates them into SQL queries. The results are tagged before being delivered to the user as XML data. An efficient publishing technique with a detailed discussion is also presented in (Shanmugasundaram et al., 2000). (Grabs et al., 2002) present a complementary study of how to extend an arbitrary XML-to-relations mapping with transactions.

3 General System Architecture

This section describes the general setting of our research. We are concerned with the implementation of very general Web-service architectures. The general model can then easily be adapted to more specific settings.

We assume a general client-server architecture, as illustrated in Figure 1: clients issue requests to servers by means of XML-based SOAP documents. The important feature of our architecture is now that the front-end is well-defined: it is exactly the set of documents allowed by the XML request or input language. The implementation of the back-end can now be done in a black-box fashion; we are free to change it as long as it still implements the specification imposed by the input documents. In the rest of the paper we will focus on a particular way of successively and systematically altering the implementation of the back-end.

The tasks of the individual components are as follows:

1. The *front-end* parses the incoming XML-documents,
2. verifies certain basic constraints such as those imposed by XML Schema or others that may be checked without the application context, and
3. generates input for the back-end by pre-processing the XML documents and converting them to custom data structures which are forwarded to the back-end. In later steps, the front-end has to provide certain basic transactional services and thus plays a role in ensuring the transactional integrity of the distributed system.
4. The *back-end* provides storage of data, query capabilities, and possible additional database features. Again, the back-end has to be front-end-aware to some degree, so that, *e.g.*, it is able to report back whether a transaction was successful or not.

While the transmission of data is done entirely in XML, we needed to extend XQuery slightly to add basic transactional functionality.

4 The Prototyping Process in Detail

This section describes the step-wise refinement process in more detail. The basic idea is to start out with a very general framework to which we add more and more knowledge in order to obtain better performance and ensure data integrity.

The prototyping process consists of a number of steps which are sketched in Figure 2. The focal point is to move from an architecture that fulfills basic conformance requirements imposed by the SOAP language and that is *fast to implement*, to an *optimized system* with many bells and whistles that can be tuned for maximum performance, maintainability, and integrity. In each step, the codebase of the prototype is refined by switching from general tools to tools that require additional semantic modeling. Ideally, the additional functionality results in more control over the system and data. In the spirit of many software engineering methodologies, it also provides opportunities for identifying problems, bottlenecks and insufficiencies of the architecture so that this feedback can be used to a constant improvement of the codebase even before moving on to the next step. It can also be implemented as an iterative process, where a solution to the problems is proposed and evaluated at the current step before entering the next step.

Technically, the four different development stages of our framework can be outlined like this:

1. XML data are stored in flat files; against these an XQuery processor issues interface-compliant queries as demanded by the API specification. Thus, the individual components are only very lightly coupled.
2. This is the first step where a relational database management system is used as back-end. XML documents are shredded and inserted into relations using a mapping technique in the spirit of (Shanmugasundaram et al., 1999). Queries are executed on the tables generated by the mapping.
3. The main purpose of this step is to add domain knowledge for enforcing data integrity and optimizing query execution. The range of technologies that are employed in this step comprise indexes, constraints, views and triggers. Note that in contrast to Step 2 the generation of the database schema is not fully automatic anymore. The domain knowledge has to be added by the database administrator.

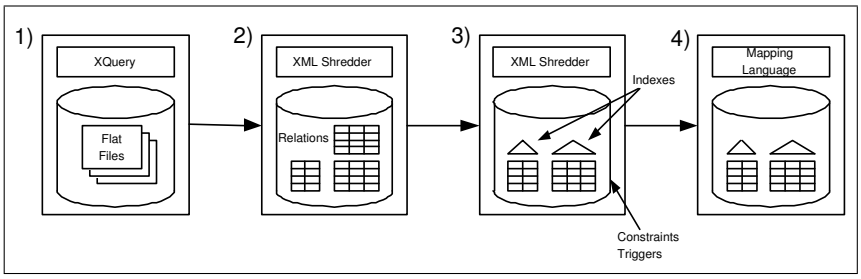


Figure 2: System architecture in the different development steps.

4. In this step, the database administrator uses the knowledge gathered in the first three steps to design a database lay-out that reflects the requirements of applications. In contrast to the first step, which only leverages native XML tools, there is no use of XML internally; this step could therefore be called 'native relational' with full leverage of relational tools and opportunities for storage and query optimization techniques common in the relational world. With this step, all automation with respect to deriving database schemas is abandoned. The database administrators have full control over the database and is free to deploy the tools of their choice.

Since the system uses relational back-ends in Steps 2–4, database administrators can use the knowledge they gathered in previous projects and, especially in Step 4, do not require additional training. So the overall idea of our framework results in a smooth transition from document-oriented XML technology to the more scalable relational technology in the back-end of the system. This proves useful since many XML products currently do not scale to massive data and, on the other hand, because in this way the integration of legacy relational systems is facilitated.

In the remainder of this section we discuss the individual steps in more detail.

1. Flat-File Storage In our context, the solution that is fastest to implement is to store the XML data in a flat file repository, and use a freely available XQuery processor for querying the data. By storing the XML data in flat files, the way to add information to a database is to create a file containing the data and registering it. Updates are performed by deleting and creating a new file. This is simple to implement but expensive in terms of performance and integrity maintenance. On the other hand, it captures well the spirit of XML query languages like XQuery because the individual XML document is the point of reference in these languages.

One of the most obvious disadvantages of this solution is query efficiency. Every time a query is run, the actual files have to be scanned and parsed into an internal representation. Although indexing could be possible, it is not well supported, and will typically not be a part of this step. Transactions can be supported by the locking mechanisms that the repository provides. However, the semantics and granularities of the locking system are usually not aligned with the requirements of XML.

An alternative or complementary approach to implementing this step would be to take advantage of RDBMS data types that are database equivalents files, *i.e.*, BLOBs, CLOBs or even XML objects. This way, the standard tools of RDBMSs can be leveraged to implement recovery, indexing, replication, *etc*. However, updates still tend to be expensive because the storage granularity is not fine enough to update only document fragments.

2. Decomposition into Relations This refinement step features a different storage model. XML data are now decomposed into relations. An XML shredder decomposes XML documents into rows and tables. Currently, the decomposition scheme is independent of the query workload. In the future, this step may also generate workload-aware storage schemes (Freire and Simèon, 2002).

The benefits of the architecture in this step compared with the architecture in Step 1, are the availability of a query language like SQL which scales better, and that the ACID properties are supported and can be used. This includes support for recovery and support for fine-granularity locking of data and thus higher concurrency.

A potential problem with the architecture in this step is that, according to the storage scheme used, an XML document is decomposed into many tables, and many joins are needed in order to reconstruct a document. However, in many cases reconstruction of documents will not be necessary, so that this problem will not be an issue.

At this stage in the prototyping process a few queries will have been issued so that further optimization will be possible in the next step:

3. Optimization of Decomposition The architecture of the previous step has potential for both high concurrency and scalability. However, in order to achieve high query performance and consistency, additional features supported by the RDBMS should be employed. These include indexes, constraints and triggers. This step offers opportunities for adding them. Typically constraints like database-wide uniqueness of XML attributes and reference constraints are candidates for declaring domain-knowledge to the database and thus ensure some important integrity constraints. Although indexes probably are also to be used in this step, their main use is not to improve query performance but constraint enforcement. In this sense, they are used on an ad-hoc basis.

4. No Use of XML Internally For some projects, the prototype developed in Step 3 will be the final one. It will satisfy many requirements. However, in general it will not scale up to the requirements of a production system. When an application manages large amounts of data or features a query-intensive workload, it is probable that more fine-tuning is needed than possible in the framework of Steps 1–3. In such a case, it will be necessary to develop the prototype into a system that does not use XML internally but that makes semantic data modeling possible and that can take advantage of the semantics.

To bridge this gap, we have designed a mapping language between the XML and the relational database schemas. In practice, the process resembles the way E/R CASE tools are used and can be supported by a Graphical User Interface (GUI). The language is used to glue the relational database schema to the elements of XML documents and, at the same, to enforce database-wide constraints on the documents. For example, if information from an XML person record is to be inserted into the RDBMS but the Social Security Number of the person is already present in the database, then the XML document has to be rejected. In this way the language is used to enforce constraints that are difficult to enforce in XML-only scenarios.

5 Performance Impressions

Figure 3 sketches some performance numbers from Steps 1–3 when different database schemas are implemented, illustrating the cost of loading data into different schemas (top figure) and the cost of querying (bottom figure).

Note that adding domain knowledge to ensure integrity does not always enhance performance by itself. Especially, Figure 3(a) shows that automatic constraint enforcement brings about additional update costs. However, the gain is certainty that the database is in a consistent state. Transaction-oriented application are a prominent case when this is useful. In practice, domain-specific modeling in Step 4 is when performance gains are most probable.

6 Document-Centric Documents

XML documents are frequently divided into two categories: *document*-centric and *data*-centric. Document-centric XML document are often documents meant for human consumption, like books, papers, *etc*., while data-centric are typically documents meant for computer consumption/data transport, and that are highly structured.

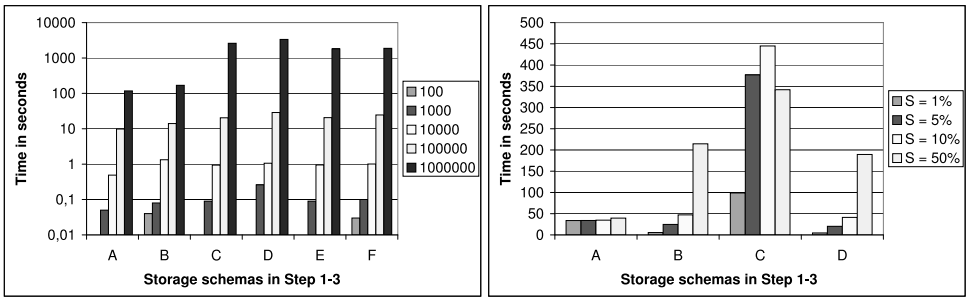
Our focus in this paper has been data-centric documents. However, it should be noted that the proposed framework is also applicable in the case of mainly document-centric documents and/or repository services.[1] In that case, it can be beneficial to store the documents in BLOBs in the database (as one of the alternatives in Step 1), and rely on associated indexes to improve performance. Thus, Step 2 is not applicable, but instead performance improvements similar to some of those proposed for Step 3 can be implemented. For XML documents special indexes are provided by the actual commercial database systems. Supported indexes typically include path indexes, as well as text-index variants.

7 Conclusions and Future Work

We have described a framework for XML database application development. The focal point of the development framework is that it enables rapid prototyping by deploying easy-to-setup general purpose tools in the first steps and then refines the application by adding more and more domain knowledge in subsequent steps until it is possible to use semantic modeling. Technically, the four steps comprise flat-file back-end storage, automatic XML document shredding, custom XML document shredding, and, as a final step, the transition to a relational back-end.

Directions for future research include investigations into how to utilize XML Schema and Semantic Web information for optimizing of Steps 2–4 in our framework. Of particular importance is the verification of queries and mappings in the fourth step where manual interaction can introduce errors. Furthermore,

[1] In a repository service a stored document is returned, in contrast to a query-generated document as in the more general case.



(a) Bulkload

(b) Sequential Scan

Figure 3: Two performance figures.

the mapping language mentioned in Step 4 currently produces more locks than necessary and therefore impedes on parallelism. A more detailed analysis of the declarative locking mechanism could lead to improved code after a query rewriting phase.

Acknowledgments We would like to thank our students Jens Gorm Rye-Andersen, Lasse Jensen, Jimmy Nielsen, Søren Nøhr Christensen and Mads Wiederholt Jensen for their implementation work and the performance measurements.

REFERENCES

Bohannon, P., Freire, J., Roy, P., and Simèon, J. (2002). From XML Schema to Relations: A Cost-Based Approach to XML Storage. In *Proceedings of the IEEE International Conference on Data Engineering*.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H., Thatte, S., and Winer, D. (2000). Simple Object Access Protocol (SOAP) 1.1. Available at `http://www.w3.org/TR/SOAP/`.

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

e-XMLmedia. Services summary. Version 3.0. Available at `http://www.e-xmlmedia.com/sol/`.

Fernández, M., Kadiyska, Y., Suciu, D., Morishima, A., and Tan, W.-C. (2002). SilkRoute: a framework for publishing relational data in XML. *ACM TODS*, 27(4).

Florescu, D. and A. Grünhagen, D. K. (2003). XL: a platform for Web Services. In *Biennial Conference on Innovative Data Systems Research*.

Florescu, D. and Kossmann, D. (1999). Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3).

Freire, J. and Simèon, J. (2002). Adaptive XML Shredding: Architecture, Implementation, and Challenges. In *Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web, VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DTWeb. Revised Papers*, volume 2590 of *Lecture Notes in Computer Science*. Springer.

Grabs, T., Böhm, K., and Schek, H.-J. (2002). XMLTM: efficient transaction management for XML documents. In *Proceedings of the Eleventh International Conference on Information and Knowledge Management*, pages 142–152.

Kordon, F. and Luqi (2002). An Introduction to Rapid System Prototyping. *IEEE Transactions on Software Engineering*, 28(9).

Manolescu, I., Florescu, D., Kossmann, D., Xhumari, F., and Olteanu, D. (2000). Agora: Living with XML and Relational. In *Proceedings of the International Conference on Very Large Data Bases*.

Orsini, R. and Celentano, A. (2002). A workbench for prototyping XML data exchange. In *Proceedings of Sistemi Evoluti per Basi di Dati (SEBD)*.

Schmidt, A., Kersten, M., Windhouwer, M., and Waas, F. (2000). Efficient relational storage and retrieval of XML documents. In *The World Wide Web and Databases, Third International Workshop WebDB 2000*.

Shanmugasundaram, J., Shekita, E., Barr, R., Carey, M., Lindsay, B., Pirahesh, H., and Reinwald, B. (2000). Efficiently Publishing Relational Data as XML Documents. In *2000*, pages 65–76.

Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D. J., and Naughton, J. F. (1999). Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the International Conference on Very Large Data Bases*, pages 302–314.

Thalheim, B. (2000). *Fundamentals of Entity-Relationship Modeling (Foundations of Database Technology)*. Springer.