Benefits and Techniques of Mapping XML onto Relational Databases

Theodore Johnson
Mathematics and Computer Science
St. Olaf College
johnsotm@stolaf.edu

Abstract

The storage and exchange of information are commonly the most important and difficult parts to develop of any software. The creation of Extensible Markup Language (XML) in 1998 sought to solve these two programming challenges. To move to XML, legacy programs had to be scraped or reprogrammed. The investment with money and resources into programs created before XML hindered a migration toward XML. The switch to XML had to be slower. The switch to XML datastore from traditional relational database (RDB) was far to costly. The solution was to map XML data onto existing RDB solutions. Due to the volume of data that needs to be stored by institutions today, automated mapping was the only way to achieve this. Up until now, no fast, easy, automated, and most importantly, *free* solution existed. This paper focuses on the projects, problems, and research that lead to the design and development of this solution for mapping XML to a RDB.

Introduction

Extensible Markup Language (XML) differs from traditional definitions of computer language. Unlike C++, a programing language, XML is a markup language. A markup language focuses on structuring and expressing data about the data, meta-data. In fact, one could express English with XML. The flexibility of XML continues to be one of its advantages in managing data over other languages. Aside from being flexible, XML allows for very human readable markup and helps separate the structure and logic of the data from the rendition or view of the data.

Unfortunately, the advantages listed above bring with them large performance costs when parsing, searching, manipulating, or validating. For this reason, XML's predecessor and superset Standard General Markup Language (SGML) found use only in huge companies that had to process large multi-gigabyte documents. SGML was not simple enough for computers or humans to be employed for smaller tasks, such as expressing a book, although some authors did so anyway.

Furthermore, small- and medium-sized companies could not afford computers fast enough to process SGML in a timely fashion. Two major changes made it possible for the computing world to move to Markup Languages. First, the price of CPU speeds ratio was low enough to process XML and SGML quickly enough for on-demand applications. Second, the demand for an easy to use and flexible means of structuring information emerged from the growth of the Internet and Electronic Data Interchange (EDI) needs of large institutions. The new potential market for SGML needed something simpler. The solution was XML.

Despite XML's gains from increased CPU speeds and the removal of SGML's complexity, the migration from complex and cryptic data formats was slow. The new road block for using XML was that the data was already being stored in relational databases. The systems and programs around the relational databases could not be converted or replaced with XML without large costs. The migration from a relational database management system (RDBMS) to a XML database management system (XDBMS) or datastore was and is too costly for some institutions.

Even with the costs of switching, XML continues to move forward, and to keep up with the momentum, the creation of conversions between XML and relation databases has taken center stage. Despite the focus on conversions, many narrow and incomplete solutions exist. The few good solutions are proprietary, do not follow standards, or are not automated. Realizing that there was no solution available, I decided to research creating a free automated, yet configurable, XML to relational database (RDB). Using that research, this paper will cover the different methods, applications both commercial and non-commercial, challenges, and finally a solution I will implement.

Overview of technology

To understand the core of the paper there are two sections below covering what XML is and the value of XML. If you are comfortable with XML, feel free to skip the sections *What is XML?* and *Why XML?*.

What is XML?

Extensible Markup Language, perhaps known more commonly as XML, is not just another language. XML provides the logic and structure to express complex data structures and data types and keeps them human readable. XML was derived, as was Extensible Hypertext Markup Language (XHTML), from a superset called SGML. XML, actually all three (XHTML, SGML and XML), are languages built from elements called tags.[i] A tag is just a "<" followed by the name of the tag and terminated by a ">"; the actual definition has more rules and variance.[ii] However, just as in XHTML, some tags may have other elements or content (body) between start and end tags. To handle both content and content-less tags, the markup languages use self-termination and end tags.

Self-terminated tag that has no content: <empty-tag />

Tag with contents needs matching end tag: <author>Ted Johnson</author>

In both cases a "/" is used either in the end tag when content is between the tags, or at the end of the start tag to show that there is no content. The ability for tags to have content XML allows nesting of tags to create a tree like structure. Also, XML's structure shows us, because it is nested within a start and end author tag, that Ted Johnson has something to do with the author.

The example of the author tag does not say what the content "Ted Johnson" is however, although it is safe to assume that he might be the author. In reality, "Ted Johnson" could be an extremely intelligent pet, capable of authoring documents. Because the definition of the data is our own to control, XML allows the author tag to instead contain another more descriptive tag.

```
<author>
  <person>Ted Johnson</person>
</author>
```

The <*person*> tag allows this example to be more detailed than the previous <*author*> tag example. Another way for an element to store more information is by using

attributes. Attributes are contained with using "<" and ">" but after the name. An attribute is defined and assigned a value with the form *attribute*="*value*", where *attribute* is name and *value* is the value assigned to the name. A common XHTML attribute is *src,* which is frequently used in the <*img*/> tag. Combining the attribute *src* and tag *img* together the following is created.

Attributes allow tags to store more information, just as with adding elements in the content (body). XML is much far too feature rich to be explained in a page or two, however, this primer gives you an idea of the recursive hierarchical structure and syntax. To learn more, there are many excellent tutorials and books on XML (Bray, "Extensible Markup Language (XML) 1.0," 2004), (Bradley, "The XML companion," 2002), (w3schools.com, "Introduction to XML," 1999).

Why XML?

Both the *Introduction* and the *What is XML?* sections have alluded to the reasons for wanting to use XML. Terms such as *recursive* and *flexible* were used when introducing XML and now must be justified. The strengths of XML fit into three main categories which will be discussed below.

- Data structure definition validation
- Data separated from rendition
- Human readable data

In 1999 when XML was first created by the World Wide Web Consortium (W3C) the structure of XML could be expressed in the form of a Document Type Definition (DTD).[iii] Furthermore, the DTD would (hopefully) impart a logical structuring on the data. Another way of expressing XML was with an XML Schema (Fallside, "XML Schema," 2001). The strength of a Schema is simply that a Schema is an XML document that expresses structures in XML. The DTD or Schema can be used to verify if a document fits its definition. Validation helps insure data integrity and data quality if, before storing the XML document, it is first checked to make sure it matches the definition (DTD or Schema).

XML documents only contain meta-data (information about the meaning and structure of the data) and data. The benefit of keeping the data and meta-data separate from the program implementation is that the implementation details can change without interfering with the data. So the desired rendition, be it a web page, PDF, database table, or whatever, can change without effecting the XML document. Something stored in XML, if found to not be defined correctly, can always be mapped via a rendition to a new language or a new XML definition.

Understanding the data is far easier in XML than a format that has been optimized to be compact for transportation or encoded for display. XML provides human readability for the data just by the hierarchical structure and element and attribute names. The attention to abstractness in XML's design allows for the definition of the elements to be independent from the language that they are written in. The semantics (meaning) of XML is not lost in the syntax (code or form).

Motivation

The initial interest in converting XML to an equivalent storage within a relational database began with a class project called the Academic Contract Explorer (ACE). The ACE project set out to change the way mathematics, statistics and computer science students applied, edited, and updated their major, minor, or concentration proposals. The class focus was on client and server applications. It was decided that ACE would communicate using XML and store the documents into a custom database. To update, delete, or query the database required us to implement fifty or so Simple Querying Language (SQL) queries. The four thousand lines of code accomplishing the queries, as it occurred to me, would not be maintainable. Because every two years new students, would need to edit the Schema and change the code for converting it to a database. Thus, the conversion model was insufficient.

At the same time, St. Olaf College was looking at a new database and interface for accessing the data. The future direction of both projects remained unclear and to have to recode or patch the system for every potential change was not practical. Further reading made it clear that a system for automating the mapping between XML and a Relational Database (RDB) could be applied to many different problems.

I was further motivated when I read about companies struggling with design and budget limitations mandating both a move toward XML while continuing to storing data in a RDB. (Goldfarb, "The XML Handbook," 2001) Companies could improve business in business electronic communication and expand partnerships by moving to XML. The RDB solutions in place hindered the movement to pure XML. The lure of improving Electronic Data Interchange (EDI), Enterprise Application Integration (EAI), and component based enterprise systems made XML impossible to ignore. Benefits from implementing EDI to faster implementation depended on converting old RDB technology into XML. This dependence although important to companies had not been automated and consequently this slowed the migration to XML.

I determined that many solutions existed but were either incomplete or proprietary. For institutions or students the costs are not practical and incomplete solutions posed similar problems to implementing a custom coded solution. From these limitations, an idea was

born: develop a free, completely automated system of storing XML documents in a RDB using an XML Schema to format and constrain the data. There were, however, many models and approaches to consider and analyze before jumping head first into implementation.

The Problem

At the end of the *Motivation* section, I outlined the general features and requirements a system should have to convert between XML and a RDBMS. To better understand the approaches when creating an implementation, a firm set of the desired functionality and features should be discussed.

The first major feature desired for usability and ease is automation of the conversion between the database and XML. If automation was not implemented, the user would have to write a unique mapping file to tell the system how to convert between the two formats. The use of unique mappings eliminates many possible dynamic uses that a XML converter would more than likely be used for. Because the mapping is dependent on the Schema or DTD, it would have to be rewritten every time the Schema or DTD changed. Finally, XML lends itself to automation because the Schema or DTD for that document contains all the information needed to perform the conversion.

Another desired feature for an XML to RDB converter would be to created unified security and permissions for both XML and RDB forms. Also, the XML stored in a database should be able to use fine- and course-grained permissions. Giving XML the control of security and permissions within the database removes redundancy in security choices between XML and RDBMSs. As with automation, the XML Schema provides a good map from which to build and communicate the desired security.

A solution system should also consider all the standards associated with databases. To provide connectivity and compatibility with as many RDBMSs as possible, the program should strive to follow the Open Database Connectivity (ODBC). When mapping between XML and database data types the SQL standards (Technical committee JTC 1/SC 32 "ANSI X3.135-1992", 1992) would also be used to increase compatibility. The XML community has developed many new languages such as XPath (Clark, "XML Path Language," 1999) and XQuery (Boag, "XQuery 1.0: An XML Query Language," 2003) for accessing and processing documents; these languages are widely supported and should be supported in an ideal solution.

Finally, the design should keep the speed of the DBMS and the flexibility of XML to create a fast, versatile system. An implementation of the system should leverage the recursive and tree-like structure of XML by using dynamic programming[iv]. The tables and columns of the database should allow for quick update or retrieval of XML documents. A slow system defeats many of the reasons for wanting to store XML in a

database in the first place.

Designs for converting between XML and RDBMS

Now that we have a outlined what features go into a well-designed system, we will look into what solutions are currently available. Researching the problem I uncovered many different approaches employed by programmers and researchers. To analyze the methods and implementations discussed below, for each one, I will compare the strengths and features of a good systems design.

Home brewed approaches

There are two naïve ways to implement a XML to RDBMS solution. One approach used is to put the XML data in one entry (column), perhaps as a string or blob data type. Although, the *blob* technique is simple and ease, from a database design perspective, there are numerous drawbacks. Using the *blob* method does not allow the database to put the data into smaller parts. The data has no structure within the database. Querying the database for a XML document in this fashion will only return the whole document which then has to be parsed to build its structure. Furthermore, because of the single entry design, the access control that databases enforce over tables and columns cannot impose granular control on the XML structure. The SQL and ODBC standards would be used, just minimally, but XQuery and XPath have no applications. The *blob* approach does not utilize any of the advantages that a database offers other than database concurrency.

The other naïve approach *shreds* the XML elements (tags) into custom hand designed database. This approach is the common free solution suggested in books. The largest drawback is that the code is not generalized and must be changed when the Schema changes. The code consists of very complex and error prone SQL queries. When hand-designing a database mistakes, will happen more often. A recursive complex Schema properly implemented in a RDB could contain twenty or more tables with many complex foreign key relationships. For non-technical persons, designing a good database and maintaining stable custom code would not be feasible. Again, just as with *blobs,* the *shred* method would use SQL and ODBC. Shredding would support XQuery and XPath, if custom implemented for the specific Schema used. The s*hred* approach takes advantage of all of the features that a rich DBMS would offer, but the trade-off comes in implementation and maintenance of the system.

Existing Solutions

Many of the programs and projects created fit into general types of solution techniques.

For that reason, only project representative of a design will be discussed. The section titled *partial* are all free software and the *complete* section looks more at proprietary and research solutions. It should be said that the programs below are analyzed from the specific focus of this paper. The analysis should not reflect poorly on the programs as they are all very appropriate for certain uses.

Partial

Both *Xindice* and *eXist* are XML databases. An XML database (XDB) differs from a RDB in that it can only store XML documents. XDB uses many techniques to speed up the querying and updating process. However, even with improvements to search an XML document in a XDB, the XML must first be loaded and parsed. The extraction process will not be as slow as the *blob* technique, but there are faster designs. The querying and updating is achieved using XML standards of XPath and Xupdate(Laux Andreas, "XML:DB Initiative: Xupdate," 2000) which is a good design choice. Also, an XDB lacks fine-grained permissions on an element level. For legacy applications that use a RDB, there is no ODBC or SQL support. *Xindice* and *eXist* are emerging programs with many new applications. The lack of legacy support makes switching to an XDB unfeasible.

Castor provides conversion from XML to Java and vice-versa. The conversion uses Java objects and code generating methods to produce the code for Java objects that represent the XML document structure. *Castor* uses the XML Schema to learn information about the structure for a given document. Castor only provides the code you must compile for each new XML Schema that you want to support. This is most disadvantageous in a dynamic setting where the Schema could change at any time. It is important to point out another program called *Kleen*. The important features of *Kleen* are the powerful GUI and modeling concepts used to help aid in the designing of the conversions to and from XML. However, *Castor* and *Kleen* only solve the XML to Java conversion, not XML to RDB.

Complete

An interesting reasearch project called XRel() is a complete conversion from XML to RDB. The conversion uses a generalized relational structure to accommodate and document structure into four tables. The simplicity makes implementation easy, and can store any document in the same relational structure. The compact design allows for easy database design, but hinders easy extraction of data using SQL or a DBMS. The four tables store any structure of a document. To achieve the four table design all the documents no maker the Schema or DTD are stored in the same tables. During extraction, the attributes and body must be queried from a separate tables. The use of indexing with binary trees helps, however, the traversal would not be as fast as just searching the table of only matching XML documents. The Schema of a XML document

is not used. A consequence is that important structural information stored in the Schema is not used that could have be used to speed up searching. Most significantly, the constraints assign in the Schema on the elements and attributes are lost and unenforced within the RDB. The absence of the constraints on the database could introduce data type errors if edited outside to the conversion program, as with the DBMS.

The most feature rich and sophisticated solution I found was by *Oracle*®. *Oracle XML DB*® attempts to bridge the XML technological bridge for companies and institutions. Oracle provides both RDB and XDB access to XML content. To balance advantages of both *blobbing* and *shredding,* Oracle allows the user to define which way the data should be stored in. Also, *shredding* is both automated and controllable for whatever the need. The connectivity is very extensive, providing SQL, XPath, and ODBC. The only feature laking was the fine-grained data typing and XML Schema defined RDB permissions. It is important to note that the maturity of features exists due to the rewriting of the DBMS itself. Oracle expanded on SQL and other standards in order to achieve such a rich system.

A quick disclaimer: during my research, I discovered that *Oracle XML DB* already used many design decisions that I had come up with. However, *Oracle XML DB* is part of the Standard edition which runs $15,000 per 4xCPU, only with *Oracle* databases, and making modifications to the code is impossible. That said, let us continue to the solution I have devised.

The Design

I have an advantage over the existing implementations in that I was able to look at their work, extract the good ideas, and learn from the bad ones. I am not, however, ignoring my failed implementation with ACE. I learned from that as well. Once the research was done, I had a good idea of the values pertaining to a XML and RDB conversion system, and had to choose which technologies to use. For ease and familiarity I chose Java. Java has a standard (Application Programming Interface) API for ODBC connections which which for this project maximizes compatibility. The *Apache*® projects *Xercies* and *Xalan* were chosen as XML parser and transform, respectively, for their stability and conformance with the Java API. For a test database, I chose *PostgreSQL*® because of the advanced features such as transactions and locking, which other free databases did not offer. Because of Java, I was able to avoid pulling together obscure software packages. I hoped that Java would speed development.

To allow for maximized compatibility with software, I chose to obey all the SQL, ODBC, XML, XML Schema, and XPath standards. This design decision was crucial, so that my solution would be able to easily work with existing software. It might seem backwards, but using standards should help encapsulate the problem into manageable development

and programing tasks. For example, XPath is a rich developed language used to search XML Documents. Using XPath helps define the project and keep the project focused on implementation and not creating new languages to support it.

Besides using industry standards, my projects feature set includes the following.

- Automated
- Configurable through the Schema
- Unified security permissions

Most importantly, automation of the entire conversion for extractions, searching, and insertion. Even with automation, I will allow customizations by specifying within the XML Schema options such as , structural, access/permissions, *blob* or *shred* storage, and data type changes. In fact, the options can be applied at any point at the hierarchy so apply to the whole branch. The XML Schema also contains constraint information for elements and attributes which can automatically be mapped to the database. This design choice allows the Schema to be the single structural and data specification for both XML and RDB formats. Data and meta-data duplication/redundancy is something XML strives to avoid, so it seems appropriate to apply it to this project. Below are some conversions of XML constraints to RDB constraints.

Restriction in XML	<=>	Constraint in SQL
<xsd:restriction base="xsd:integer"> <xsd:minInclusive value="200"/> <xsd:maxInclusive value="60000"/> </xsd:restriction>	<=>	y integer CHECK (200 > price and price > 6000)
<xsd:restriction base="xsd:string"> <xsd:pattern value=".(b|d)."/> </xsd:restriction>	<=>	y SIMILAR TO '%(b|d)%'

The next step after laying out the languages and features is to outline the mappings between every data type in XML to its equivalent RDB data type. Although the set of Schema data types is large (numbering about 42 with some overlap), the mapping for each only has to be defined once. More importantly, the mapping only needs to be written in Java once, which is better than having to rewrite the code every time any Schema changes. To understand the process better examine example conversions below.

XML Schema data type	<=>	SQL equivelent data type
<xsd:simpleType name="ID"> <xsd:restriction base="xsd:decimal"/> </xsd:simpleType>	<=>	CREATE TABLE ID (body text SIMILAR TO '[0-9]*\.[0-9]*');
<xsd:simpleType name="SKU"> <xsd:restriction base="xsd:string"> <xsd:pattern value="\d{3}-[A-Z]{2}"/> </xsd:restriction> </xsd:simpleType>	<=>	CREATE TABLE SKU (body text SIMILAR TO '[0-9]{3}-[A-Z]{2}');
<xsd:complexType name="USAddress" > <xsd:sequence> <xsd:element name="name" type="xsd:string"/> <xsd:element name="street" type="xsd:string"/> <xsd:element name="city" type="xsd:string"/> <xsd:element name="state" type="xsd:string"/> <xsd:element name="zip" type="xsd:decimal"/> </xsd:sequence> <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/> </xsd:complexType>	<=>	CREATE TABLE USAddress (name text NOT NULL stree text NOT NULL city text NOT NULL state text NOT NULL zip text NOT NULL);
<xsd:simpleType name="SixUSStates"> <xsd:restriction base="xsd:string"> <xsd:length value="6"/> </xsd:restriction> </xsd:simpleType>	<=>	CREATE TABLE SixUSStates (body char(6) NOT NULL)

Now that we mappings for the data types, XPath must be able to traverse the RDB via mappings. Again, as with the data types. The XML parsers allow for easy implementation. Here are some more mappings to give you an idea of the logical conversions for XPath.

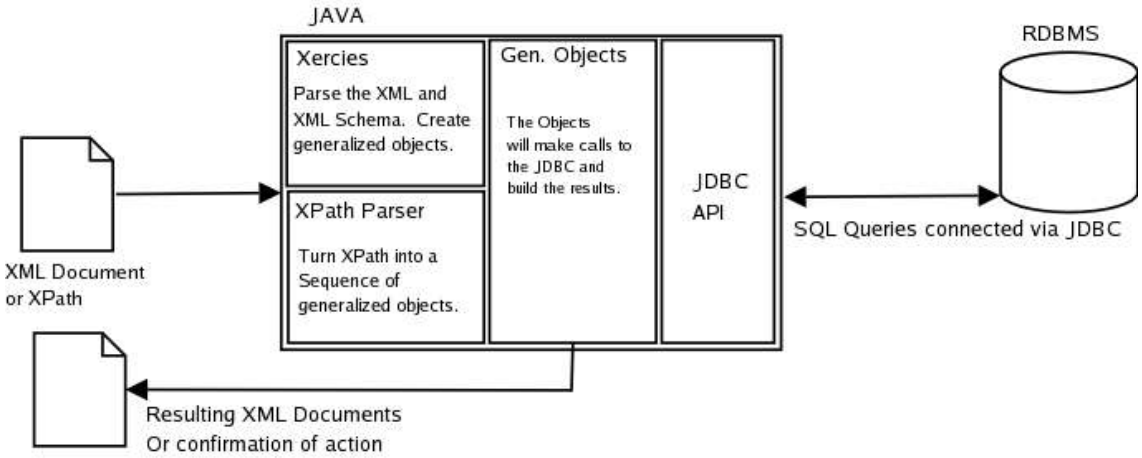
XPath	<=>	SQL equivelent
child::para[attribute::type="warning"]	<=>	From the current element result set. Assume current set is from table BOOK. SELECT element_para FROM BOOK; Using the result from the previous query. SELECT * FROM para WHERE type="warning"; The result set from query above returns the proper matching elements.
child::*	<=>	From the current element result set. Assume current set is from table BOOK. SELECT element_para ... FROM BOOK; The result set from query above returns the proper matching elements.
child::chapter/descendant::para	<=>	From the current element result set. Assume current set is from table SECTION. SELECT element_chapter1 element_chapter2 FROM SECTION; Use result set from above. SELECT element_para FROM chapter WHERE id = ...; Use result set to get para elements SELECT * FROM para WEHERE id = ...; The result set from query above returns the proper matching elements.

The overlap in functionality between the XPath and data type mappings lead me to use one object for each data type. This eliminates code duplication between XPath and data type mappings. The methods for these objects perform all of the operations for XPath and data type conversions and are listed below.

- Traverse – Builds the set of the node contained in the specified current node. A node can be attributes, elements, or character data.
- Query – Returns a set of nodes that matched from the given set. A set might be as small as one row, one column, or as large as an entire table.
- Update – Will act on a set of nodes specified from XPath.
- Create – Used to add new tables and insert new document into those tables.

The benefit of using a standard set of methods is that when a set needs to be processed it does not matter what objects are in it. The technical term is polymorphis. I will explain further however. Imagine we have a result set with attribute objects, simple element objects, and complex element objects. The four methods can be called on any of these objects no matter which one they are. An attribute.Query method requires the same arguments as complexElement.Query. What happens to the arguments within the method is different, but calling them is simplified.

All of the mappings described above will be part of a pipeline. As you can see from the diagram of the overall system design that the complexity was minimized.



The design section gave you an outline of the complexities within the mappings for this implementation. However, the complete implementation and programming design choices would in themselves be another paper.

Future Implementation

The solution outlined above at the time of the writing of this paper has just begun. I hope to demo the implementation at the Midwest Instruction and Computing Symposium in 2004. Because of the projects size, after the initial development I plan on moving the project into the open source community to find support in creating a production quality application. Once complete, the paper will be revised to include the problems and eventual final product of this project.

References

Bray, Tim, Paoli, Jean, Sperberg-McQueen, C.M., Maler, Eve, & Yergeau, François (2004). Extensible Markup Language (XML) 1.0 (Third Edition) - W3C Recommendation 04 February 2004. Retrieved March 12, 2004 from http://www.w3.org/TR/2004/REC-xml-20040204/.

Bradley, Neil (2002). The XML Companion. UK: Person Education Limited.

w3schools.com (unknown). XML Tutorial. Retrieved March 10, 2004, from http://www.w3schools.com/xml/default.asp.

Fallside, David C. (2001). XML Schema Part 0: Primer – W3C Recommendation, 2 May 2001. Retrieved March 3, 2004 from http://www.w3.org/TR/xmlschema-0/.

Goldfarb, C. F., Prescod, Paul (2001). The XML Handbook (Third Edition). US: Prentice Hall PTR.

Technical committee JTC 1/SC 32 & The Open Group (1996), Technical Standard. Data Management Structured Query Language (SQL) Version 2. Retrieved March 12,2004 from http://www.opengroup.org/publications/catalog/c449.htm.

Clark, James & DeRose, Steve (1999). XML Path Language (XPath) - Verion 1.0 – W3C Recommendation 16 November 1999. Retrieved March 12, 2004 from http://www.w3.org/TR/xpath.

Boag Scott, Chamberlin Don, Fernández, M. F., Florescu Mary, Robie Jonathan, & Siméon Jérôme (2003). XQuery 1.0: An XML Query Language – W3C Working Draft 12 November 2003. Retrieved March 12, 2004 from http://www.w3.org/TR/xquery/.

Laux Andreas & Martin Lars (2000). XML:DB Initiative: XUpdate – XML Update Language. Retreived March 13, 2004 from http://www.xmldb.org/xupdate/xupdate-wd.html.

Huzii Singo, Toshiyuki Amagasa, Yoshikawa Masatoshi, & Uemura Shunsuke (2000). XRel: a path-based approach to storage and retrieval of XML documents using relational databases. ACM Transactions on Internet Technology. Volume 1 Issue 1, 110-141.

Acknowledgments

I would like to thank Professor Richard Brown for offering classes with the flexibility to research and explore ideas that product papers like this one. Thanks to Michael Zahniser who was a co-author of the ACE database design and 4000 lines of code. Thanks to XRel team for being most helpful in sending their implementation to me that could not be found on the Internet.

i The terms tag and element have different meanings, however in practice and this paper no distinction will be made between them.

ii The actual definition of a tag involves precise rules, as does XML on a whole. For definition and specification of XML see ("Extensible Markup Language (XML) 1.0," 2004).

iii The W3C is the leading developer of standards for the Internet. The W3C web site can be found at http://www.w3.org/.

iv Dynamic Programming is the concept of solving a problem by breaking it up into small and smaller logical pieces to make it easier. A very simple example. Take the mathematical formula (4 + 5) – (6 + 2). Humans naturally split the problem up into 4 + 5 then 6 + 2 and then subtract. More complex examples can be found on the Internet.