Indexing XML with Relational Tables

by

Jeffrey H. Yu

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science at the

Massachusetts Institute of Technology

May 24, 2002

Author

Department of Electrical Engineering and Computer Science
May 17, 2002

Certified by

Kurt Fendt, Peter Donaldson
Thesis Supervisors

Accepted by

Arthur Smith
Chairman, Department Committee on Graduate Theses

Indexing XML with Relational Tables

by

Jeffrey H. Yu

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science at the
Massachusetts Institute of Technology
May 24, 2002

Abstract

XML content management is still in its infant stages as new ways of storing and retrieving XML are constantly being developed. This paper explores an XML content management solution that provides a query language richer than XPath. The purpose is to provide the ability to query data from interrelated XML documents, which is a feature unavailable in XPath. The solution utilizes a relational database to index XML documents with simple XPath expressions and queries data with SQL statements that map onto a subset of XQuery.

Thesis Committee:

Thesis Supervisor: Kurt Fendt, Research Associate, Comparative Media Studies
Thesis Supervisor: Peter Donaldson, Professor of Literature
Technical Supervisor: Christopher York, Technical Director, MetaMedia Project

Acknowledgments

I would like to specially thank Kurt Fendt for the opportunity he has given me. Your willingness to work out the difficult conflicts with me and other research assistants made everything run smoothly. I also thank Peter Donaldson for his supervision of my thesis. My most heartfelt thank you goes to Christopher York for going above and beyond his duty to provide technical advice, comments on my thesis, and most importantly, for providing a healthy work environment and being my friend.

I would also like to thank the rest of the Metamedia group for their help and for an enjoyable year.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

There are many programs available that allow users to store and query XML documents. Unfortunately, most of these programs lack the ability to run complex queries as the query syntax is limited to XPath. The programs that do provide this ability require steep licensing fees. These are the problems with which the Metamedia group at MIT had been faced. This paper describes a solution for storing and querying XML documents with a richer query syntax while eliminating budget concerns by using open–source software.

To help fully understand the problems and the motivation for creating this solution, the following subsections will provide provide background information on XML and the Metamedia group's project.

1.1 XML

XML (Extensible Markup Language) was introduced back in 1997 during the Internet boom. XML was developed specifically for the Web by the W3C (World Wide Web Consortium) in an effort to provide important features that are not available in HTML (HyperText Markup Language) **[1]**. HTML is still the most commonly used format to store and transmit data over the Web. Both HTML and XML are subsets of SGML (Standard Generalized Markup Language), which has been in existence since 1986.

XML is described as being a "metalanguage," which means that it is a language used to describe other languages. This provides the ability to create an unlimited number of languages, each one specific to one's needs. This would not be possible with HTML as it is limited to a set number of tags. The three main features that XML provides over HTML are: extensibility, structure, and validation **[2]**.

- Extensibility: XML allows users to specify their own tags and attributes.

- Structure: XML supports the specification of structures deep and complex enough to represent object oriented hierarchies and database schemas.
- Validation: XML provides the ability for users to validate the structure of documents upon retrieval.

The use of XML documents can be used in two ways. One is the simple exchange of data of XML documents, much like how two users would exchange Excel spreadsheets. The second is the management of XML data to create a database. The scope of this paper is only concerned with the second use, and will therefore focus on that issue.

1.1.1 XML Database Systems

An XML database is defined to be a collection of XML documents and their parts, maintained by a system having capabilities to manage and control the collection itself and the information represented by that collection **[3]**.

The application of traditional database technologies to XML has proved to be problematic over the past years. The problem is that the data in XML documents often have special characteristics that are not found in traditional databases **[3]**. For example, an XML document can have a complex structure in which the data consists of both natural languages and multimedia entities. Traditional databases typically do not represent such documents, and thus it would be difficult to apply its concepts to XML databases. The following sections will present the advantages and disadvantages of using XML documents to store data.

1.1.1.1 Advantages

One of the greatest advantages of using XML documents to store data is that they are self–describing in that the markup describes the structure and types names of the data

[4]. This basically means that external data, such as a filename, is not needed to describe the data. All of the data needed from an XML document is stored in its content.

Another important advantage of XML is that it is written in Unicode, which makes it portable across multiple platforms and languages. In addition, the contents of the XML database (basically the XML documents) can be migrated to another XML database with ease.

1.1.1.2 Disadvantages

XML documents, when stored as text files, need to be parsed before the data can be accessed. The parsing of XML documents is costly and thus degrades performance. This means that the data stored in the XML documents is not always ready to be extracted, whereas the data stored in a relational database can be immediately extracted with a simple SQL (Structured Query Language)[1] query.

Another disadvantage is that XML is generally regarded as being verbose, which leads to inefficient storage. Relational databases, on the other hand, are considered to be extremely space efficient. This is not always the case as certain documents are better represented by an XML document than by tables in a relational database. For example, an XML document would represent every act, scene, and line of dialogue in Shakespeare's Hamlet more easily than a relational database. Nevertheless, most database applications do not require the representation of Shakespeare's Hamlet. As a result, XML databases are only useful in special cases such as where one is building a literature archive.

[1] SQL (Structured Query Language): A standard query language used to retrieve data stored in relational tables.

1.1.2 Data vs Documents

As explained earlier, using a XML database is only suitable in certain scenarios. These scenarios can be easily differentiated once one understands the concepts of data–centric and document–centric documents.

1.1.2.1 Data–centric Documents

Data–centric documents are those that use XML as a means of transporting data. For example, a sales order containing data such as price, date, and item has a regular structure that can be represented in an XML document, Excel spreadsheet, or a traditional regular database all with ease. In other words, it is not critical that a sales order be stored in a specific format for it to be understood. When storing data–centric documents, it is necessary to utilize an XML–enabled database[2], such as a relational database [4]. Relational databases are usually tuned for efficient data storage and retrieval, and thus would be an ideal choice for storing data–centric documents. Figure 1 illustrates how the contents of a data–centric XML document can be directly mapped to a table in a relational database.

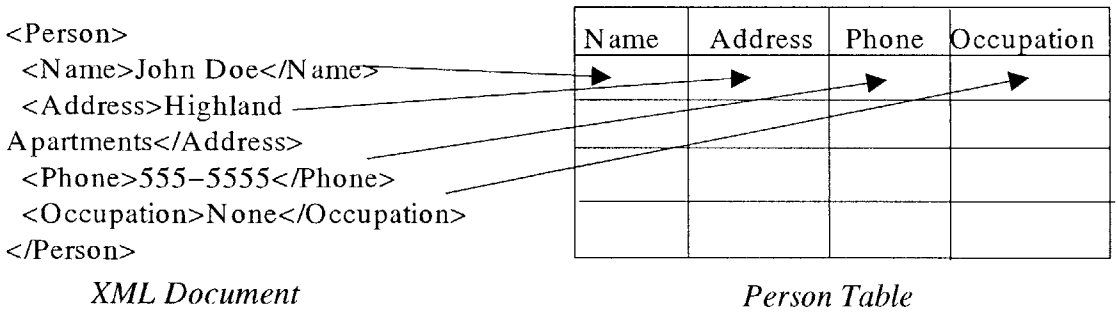

Figure 1: Mapping a data–centric document to a relational table

1.1.2.2 Document–centric Documents

Document–centric documents are characterized by irregular structure and mixed content,

[2] XML–enabled database: a database that is not specifically designed to store XML documents, but can recognize XML and has limited capabilities in storing and manipulating XML. The internal model is not based on models such as relational and hierarchical models, not XML.

designed to be read by humans **[4]**. An example of a document–centric document is the aforementioned Shakespeare play, "Hamlet." The contents of such a document–centric document cannot be easily migrated to another format like data–centric documents are able to as it is uncommon for two distinct formats to represent the same irregular structure of a document–centric document. It is ideal to use XML–native databases[3] when storing document–centric documents as they designed specifically to store XML documents with the XML model intact. XML–native databases store XML documents as documents while XML–enabled databases store the data in the documents. This allows XML–native databases to run XML–specific tools on the documents directly. On the other hand, XML–enabled databases need to reorganize the data before any XML–specific tools can be used.

1.2 Project Background

The goals of the Metamedia group at MIT is to create an archive of myriad Humanities disciplines, accessible through the Web. The archive is to be available to professors and students in different universities so that they can browse through and share their thoughts and opinions on the various topics. XML was chosen as the format to store the literature documents because of its high extensibility. XML can have pointer tags so that annotations can be made to reference specific sections of the literature documents. An annotation can be something as simple as a reader comment or as complex as a movie clip referring to a certain page or line number. The ability to create and view annotations is the highlight feature of the project as it provides an online community for people to share information.

1.2.1 Requirements

In order to achieve the goal of creating an archive that can be properly queried, a number

[3] XML–native database: a database that is specifically designed to store XML documents; it has XML specific tools for querying/searching and manipulating stored XML documents. The internal model is based on XML.

of requirements must be satisfied. The following subsections will present and describe these requirements.

1.2.1.1 Context Querying

Any database must have the ability to perform searches and return data based on the search parameters. For example, a user must be able to find all of the documents written by Shakespeare. Without this functionality, it would be impossible to implement a useful archive.

1.2.1.2 Joins

Joins are conditionals used in querying that links the attributes of different types of documents to find exactly what the user needs. Joins are needed when different documents cross reference each other and the user needs data from a document of one type that is dependent on another document. Joins are necessary in the project because the annotations are dependent on the literature documents and a user may choose to search for annotations based on certain aspects of the literature documents that they annotate.

1.2.1.3 Grouping

Grouping is a function used in querying that, like its name suggests, groups the queried data based on a certain attribute. This is particularly useful when trying to group the results of an aggregate function by a certain attribute. For example, if there were a database of employee/salary information where each employee belongs to a department, the user would be able to obtain the total salary for each department by using the grouping function. Without the grouping feature, the user would only be able to obtain either the total salary for every department combined or the individual salaries, which would require the user to manually sum up the salaries for each department.

2. XML Languages

Two important XML languages that play a significant role in our solution will be presented in this section.

2.1 XPath

XPath is a language, developed by the W3C, used for addressing different parts of an XML document **[5]**. XPath is mainly used to transform documents into different forms, such as HTML. However, in terms of XML databases, XPath can used to extract different parts of a document that match the pattern in the XPath query. XPath uses a path notation that works well with the tree structure of XML documents. Also, its syntax allows the use of predicates which strengthens its ability to match a part of a document.

The following is an example that illustrates the XPath syntax:

```
//a/b[/c = 'x']
```

This XPath example will match nodes that are labeled 'b' that have an ancestor node labeled 'a' and have a child node labeled 'c' whose value equals 'x'.

Unfortunately, XPath was never intended to be used as a querying language like SQL. It was specifically designed to be used on only one document at a time by XSLT and XPointer, an XML Transformation language and an XML pointer language **[5]**. As a result, XPath's querying ability is limited; it does not offer joins and groupings. This is unacceptable for an XML database system should be built on a model that supports collections of inter–related documents **[3]**. Without joins, it becomes extremely difficult to perform queries on documents that reference others. Without groupings, developers will be forced to write inefficient code for reasons explained in section 1.2.1.3.

2.2 XQuery

XQuery is a query language that was developed and in the process of being finalized by the W3C. At present, there does not exist a standard query language for XML. Many proposals for an XML query language have been made, but they are designed to accommodate for specific data sources and not all **[6]**. XQuery is supposed to be able to applicable across all XML data sources.

XQuery's specification includes the essential features of joining and grouping, making XQuery highly anticipated as XPath does not offer them. XQuery also provides the ability to query data from multiple documents. Its syntax uses the same path expressions in XPath, but it is much richer. XQuery's greatest improvements over XPath are the use of variables and FLWR (for, let, where, and return) expressions. FLWR expressions allow for iteration and the binding of variables to intermediate results. This translates to the ability to join between documents. A simpler way to think about XQuery is that it allows the use of multiple XPath queries where each query line can be assigned to a variable and returned. The following is an example that illustrates the XQuery syntax:

```
for $a in document('foo.xml')//bar
for $b in document('foo2.xml')//bar
let $var1 := $a/x[y = 'z']
let $var2 := $b/x[y = 'z']
where $var1 = $var2
return $var1
```

This XQuery example goes through two documents, foo.xml and foo2.xml, to obtain two nodes based on a path expression similar to XPath (section 2.1). It then returns the nodes where the values in both documents are equal. This example illustrates the use of variables to store intermediate results, the use of multiple path expressions, and the joining between two documents.

3. Related Work

This section will describe some of the ideas and tools related to our solution that attempt to solve the same issues.

3.1 Oracle XML SQL Utility

The Oracle XSU (XML SQL Utility) models XML document elements as a collection of nested tables [7]. The purpose of representing XML documents in relational tables is to use the features and advantages of relational databases. These advantages include the requirements in section 1.2.1 as SQL provides all of these features: context querying, joining, and grouping.

There are many programs other than the Oracle XSU that employ this "XML document to relational tables" conversion process. Unfortunately, this scheme is only useful when working with data–centric documents, which have fixed structures that can be easily represented by relational tables. Document–centric documents, on the other hand, have unfixed structures that are extremely inefficient to migrate to relational tables. Since the Metamedia project focuses around document–centric documents, the Oracle XSU is not sufficient.

3.2 XRel

XRel [8] is a path–based approach developed by Masatoshi Yoshikawa and Toshiyuki Amagasa for storing and retrieving XML documents with indexes using relational databases. The main idea is to decompose the documents into nodes and then store them according to node type. Each node's path information, relative to the root, is also stored. An algorithm for translating a subset of XPath expressions to SQL queries is also provided.

Different nodes are stored in different tables: *element*, *attribute*, and *text*. There is also a *path* table that contains the path expressions to every node so that its entries can be referenced bye the different node tables. Every node entry has a reference to a document and a path expression. A node entry also has a pair of start and end integers, which correspond to regions, to denote the position of a node. XRel uses an the concept of a region to preserve the precedence and ancestor/descendant relationship among nodes.

XRel essentially provides a full–fledged indexing scheme that provides the user access to every node in a document. XRel's performance in evaluating XPath expressions is impressive. In addition, it eliminates the need to parse an XML document to a DOM before performing an XPath query. Unfortunately, its querying capabilities are limited to a subset of XPath, which as explained earlier, is inadequate for our purposes. The authors plan on extending XRel to provide better querying features in the future.

3.3 XISS

XISS (XML Indexing and Storage System) [9] is an XML indexing system that utilizes a numbering scheme for elements. The purpose of XISS is to provide a means for efficiently evaluating regular path expressions by decomposing them into several simple path expressions and then joining them. The developers of XISS provide the path join algorithms necessary for the join process.

Given an optimized decomposition of a path expression, the author's claim performance figures up to an order of magnitude greater than other methods. Unfortunately, XISS's query performance is substantially affected by the way in which the path expressions are decomposed, and the author's have yet to find an optimal decomposition procedure.

Even with an optimal decomposition method, XISS only provides an interface to evaluate regular path expressions, which, like XRel, is inadequate for our purposes.

4. Implementation

Section 2 highlighted the two XML languages, XPath and XQuery. It is clear that XPath does not provide a query syntax rich enough for our purposes. While XQuery does provide a richer query language, its unavailability suggests a need for a solution that offers the same features. The solution that will be presented from hereon utilizes both XPath and XQuery to provide a way of querying XML data with a syntax that can be mapped directly to a subset of XQuery.

4.1 Design

The solution utilizes relational tables to store and index XML documents. The idea is to run preset XPath queries on XML documents before they are stored in the database, and then store the results as a way to index the documents. This process can be considered to be caching, however, this is not the case as will be shown later. The indexes are then used by SQL queries to retrieve the necessary data. The advantage of using SQL is that it provides the previously absent features of joins and groupings. With these features, it is possible to create SQL queries where each sub–query maps onto a line in an XQuery query, which will also be shown in a later subsection.

Indexing XML with relational tables is an idea that was conceived as different methods of caching XPath queries were being brainstormed. After much analysis, it became apparent that an XPath query simply returns a substring of an XML document, which can be represented by an offset and length. This property is extremely useful for it allows for efficient XPath query caching. Instead of storing the entire substring returned by an XPath query, which may require large amounts of data space, only the offset and length (both integers) need to be stored. The idea of using an offset and length is similar to XRel's (see Section 3.5) use of regions for they are also used to determine ancestor/descendant relationships, as will be shown later. The following subsections will

describe the implementation details of the Metamedia group's program that takes advantage of this property to create a system that can store and query XML with a rich syntax.

4.2 Relational Tables

As explained earlier, our solution utilizes a relational database to store the XML documents and index data. In order to achieve the highest level of efficiency from a relational database, it is essential that the tables are designed to store data that is not redundant. For example, in this case an XPath query result is stored as an offset and its respective length, as opposed to the entire query result. Our solution only requires three tables: a document table, an XPath table, and a node table.

4.2.1 Document Table

The document table is self–explanatory for each row represents a document where one column contains a document's contents and another contains a unique *docid* (Document ID) that is assigned to each one. An additional unique identifier column is used for future consideration. The idea is to use a document's absolute path in the file system, before it is stored in the database, for future reference when a different database is used and the documents need to be migrated.

The *docid*'s serve as the table's private key[4] for each one is unique throughout the entire table. As such, the document id can be referenced by foreign keys[5] in other tables. The following figure illustrates the document table.

[4] Private key: A column in a relational table that uniquely identifies a record/row.

[5] Foreign key: A column in a relational table that references a private key, but is not required to be unique. Records/rows that have foreign keys are meant to complement the record/row that is identified by the referenced private key.

Docid	URI	value
1	/usr/local/file1.xml	<Root> ...
2	/usr/local/file2.xml	<Root> ...
3	/usr/local/file3.xml	<Root> ...
4	/usr/local/file4.xml	<Root> ...

Figure 2: The doc table

4.2.2 XPath Table

The XPath table contains a list of preset XPath queries that are run against every document that is stored in the database to create the index. The preset XPath queries are determined by compiling a definitive list of all the XPath queries that are needed to retrieve the necessary data for a given project. Compiling such a list is possible as a given project uses a set number of XML document types as well as a set number of XPath queries. The XPath queries are then broken down into simpler ones that can later be combined with SQL to perform the original complex queries. The reason for breaking down the XPath queries is because many of them use the same simpler parts. Thus, by using simpler XPath queries, it provides for higher flexibility in constructing a wider range of more complex XPath queries while storing a smaller number of simple queries. For example, take the following two XPath queries:

//Person//Name

//Item//Name

The first XPath matches the "Name" nodes whose ancestor node is "Person" while the second XPath also matches the "Name" nodes, but only those whose ancestor node is "Item." The two XPath's can be broken down into three parts: //Person, //Item, and //Name (//Name is used twice). These three parts can be put into the XPath table and SQL can be used to concatenate the individual XPath's to create the original XPath

queries. Section 4.3 will provide a number of examples that will help illustrate this.

Much like the *docid* in the document table, the XPath table has an *xpathid* (XPath ID) column that serves as the table's private key. In addition to the *xpathid* column and the XPath string column, there is a namespace column. A given XPath query must be complemented with a set of namespaces so that the query can match nodes based on their local names and relevant namespaces **[5]**. A node in an XML document is identified not only by its local name, but also a namespace. Refer to Appendix A for information on namespaces. The figure below illustrates the XPath table.

xpathid	path	namespace
1	//a:name	a=http://www.w3...
2	//a:address	a=http://www/w3
3	//b:book/title	b=http://www...
4	//b:book/author	b=http://www...

Figure 3: The XPath table

4.2.3 Node Table

The node table represents the heart of our solution for it contains the entire index data for all the documents. The following figure illustrates the node table.

docid	xpathid	position	length
1	2	25	42
1	3	156	31
2	1	52	10
3	3	154	32

Figure 4: The Node table

Each record/row represents a result from a given XPath query and document on which the query was run. The XPath table is consisted of the following columns: *docid*, *xpathid*, *position*, and *length*. *Docid* and *xpathid* are both foreign keys for they refer to private keys in *doc* and *xpath* tables, respectively. A given *docid* and *xpathid* does not denote a unique record for a number of results can be returned when running an XPath query on a document. Position and length represent one of the substrings of the XML document that is returned by the XPath query. For example, a row in the node table that has the values 1, 2, 3, and 4 (docid, xpathid, position, and length) translates to the following: the substring with offset 3 and length 4 in the document with *docid* equal to 1 is a result, or one of the results, from running the XPath query with *xpathid* equal to 2.

The results of the XPath queries stored in the XPath table are cached in the node table. However, as explained in section 4.1.2, the cached information is not directly used, but rather they are combined to obtain results for more complex queries. For this reason, the cached information acts like an index and not a simple cache look up.

4.3 Storing and Indexing Documents

Storing and indexing documents is the first step that needs to be taken before any query

can be made. A document is first added by inserting it into the document table. When a document is added, a unique *docid* is assigned to that document by the database. Afterwards, the program scans through the XPath table and runs every single XPath query on the document that was just stored. Every result returned is inserted into the node table as a record, making sure that the proper *docid* and *xpathid* values are used. The following flow diagram illustrates the steps involved in storing and indexing a document.

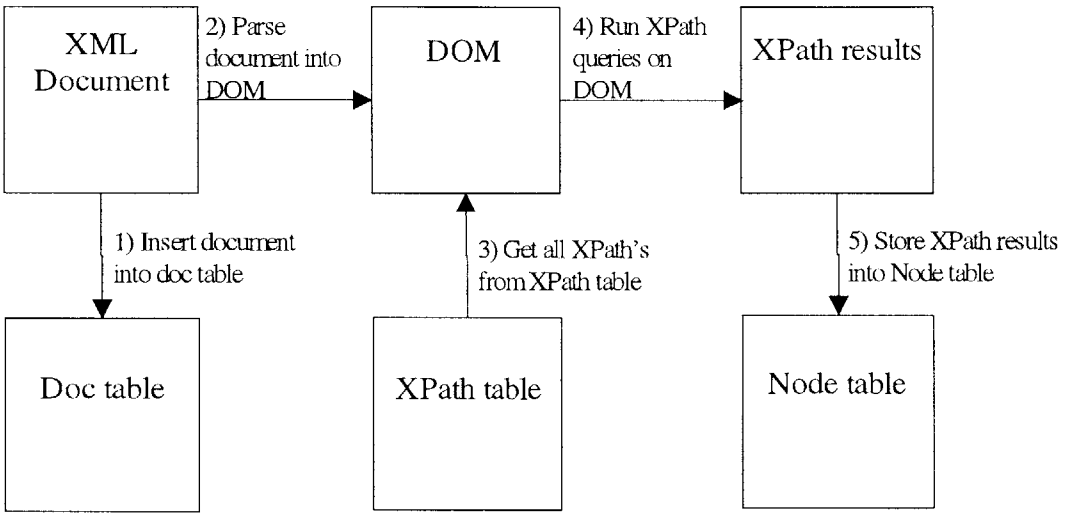

Figure 5: Flow chart for inserting documents

4.4 Querying

Querying is the most complex portion of the program. All of the basic components stored in the relational tables need to be manipulated to retrieve specific data. It was already established that XPath is insufficient as a querying language. To remedy this, our solution is to mimic the features of XQuery by using SQL and our indexing scheme. XQuery provides the many features unavailable in XPath (see Section 3.3), such as joins and groupings. Furthermore, much of XQuery's syntax is carried over from XPath, which makes it possible to utilize the indexing scheme (uses XPath) in our solution. This section will provide a number of examples to illustrate how an XQuery query is

translated to a SQL query that uses our indexing scheme.

4.3.1 Simple Query

This section will use a simple query as an example to help understand the basics of how to construct a query using our indexing scheme. Consider the following XQuery query:

```
for $a in document('foo.xml')//Person
return $a
```

This XQuery simply returns all of the nodes in the XML document, foo.xml, that are labeled "Person." The following pseudo–SQL query (refer to Appendix B for the complete SQL translations) is a translation from the XQuery above that checks all documents in the database, instead of just foo.xml:

```
SELECT get(node) FROM nodeTable WHERE path(node) = '//Person'
```

This query, as well all of the other queries, uses two custom functions *get* and *path*. The *get* function returns the substring for a given document, identified by a node. The *path* function returns the XPath query string that was used to return the substring that is identified by a node. This is possible for a given substring can only be obtained by one XPath query, or an identical one. Refer to Appendix B for more detailed information on the *get* and *path* functions.

The SQL query in this example simply returns all of the substrings from all documents that are results from the XPath query that equals "//Person." The following figure circles the nodes in an example XML document that would be returned by the above query.

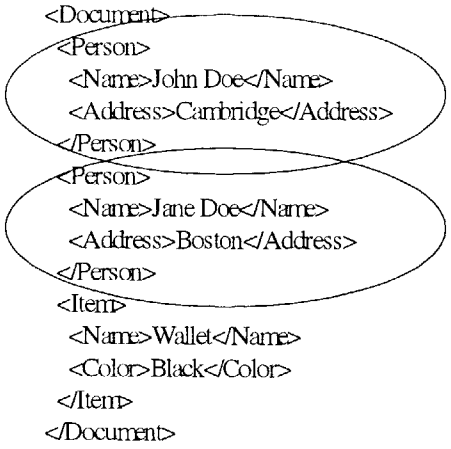

Figure 6: Nodes returned by simple query

4.3.2 Context Query

The following is an XQuery example is more complex than the previous one for it makes use of predicates:

```
for $a in document('foo.xml')//Person [./Name = 'Cornelius']
return $a
```

This query returns all nodes labeled "Person" where a child node named "Name" equals "Cornelius." The equivalent pseudo-SQL query (refer to Appendix B for the complete SQL translations) is as follows:

```
SELECT get(nodeTable.node) FROM nodeTable,
        (SELECT node FROM nodeTable WHERE path(node) = '//Name' AND get(node) =
'Corenelius') AS Name
WHERE path(nodeTable.node) = '//Person' AND contains(nodeTable.node, Name.node)
```

The addition of the predicate translates to a much more complex SQL query that involves a subquery. The outer select clause remains similar to the previous example in that it returns the results from an XPath in which the query equals "//Person." However, there is an additional condition that uses the custom function *contains*. The purpose of this function is to verify that a given XPath result is within the result of another XPath query. In context to the example above, the contains function ensures that the node returned by

the XPath query, //Name, that is equal to Cornelius is a child node of the //Person node that is returned. Refer to Appendix B for more detailed information on the *contains* function.

The subquery, labeled "Name," is simple when viewed separately. It returns nodes (or substrings from a technical point of view) that are returned by the XPath query that equals "//Name" and whose value equals "Cornelius." The results of this subquery are used by the outer select clause to complete the query. Figure 7 illustrates a document and the nodes that would be returned by this query.

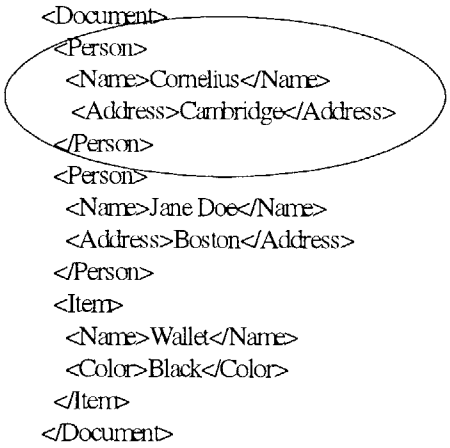

Figure 7: Nodes returned by context query

Note how the XPath query is decomposed into two different parts and then later joined by the where clause. Although not as complex, this decomposition/join process is similar to that of XISS (see Section.3.6).

4.3.3 Complex Query

The previous two XQuery examples were simple enough to be represented by XPath. Consider the following XQuery example, which beyond the capabilities of XPath:
The following is an XQuery example that cannot be represented by a simple XPath

query:

```
for $People in document('foo.xml')//Person[./City = 'Boston']
let $Name := $People//Name
let $Age := $People//Age
let $Occupation := $People//Occupation
return $Name, $Age, $Occupation
```

This XQuery returns the names, ages, and occupations for people who live in the city of Boston. It is not possible for an XPath query to return multiple individual entries as its syntax only allows the matching of one type of node, given by the path expression. This can be accomplished in SQL by using the join functionality. In an XQuery that returns multiple tuples based upon a for statement, such as this example, SQL joins must be performed across all subqueries. In this example's case, the subqueries representing the for statement and three let statements must be joined with one another. The following is a pseudo–SQL query and the complete SQL query can be found in Appendix B.

```
SELECT get(Name.node), get(Age.node), get(Occupation.node) FROM
        (SELECT nodeTable.node FROM nodeTable,
                (SELECT node FROM nodeTable WHERE path(node) = '//City' AND
get(node) = 'Boston')
                AS City
        WHERE path(node) = '//Person' AND contains(nodeTable.node, City.node))
        AS Person
        JOIN

        (SELECT node FROM nodeTable WHERE path(node) = '//Name')
        AS Name ON contains(Person.node, Name.node)
        JOIN

        (SELECT node FROM nodeTable WHERE path(node) = '//Age')
        AS Age ON contains(Person.node, Age.node)
        JOIN

        (SELECT node FROM nodeTable WHERE path(node) = '//Occupation')
        AS Occupation ON contains(Person.node, Occupation.node)
```

Although this select query is long, there is a fair amount of repetition as the last three subqueries are almost identical except for the XPath query that they are resolving. The first subquery represents the for statement in the XQuery in which it is obtaining the people that reside in Boston. The other three subqueries are simply retrieving all of the

names, ages, and occupations without any conditionals. The join statements ensure that the names, ages, and occupations returned are for their respective people. This is achieved by the *contains* conditional that checks whether or not the nodes returned by the subqueries belong to a certain person (see section 4.3.2). Without the join, there is no way of knowing if a given name, age, occupation combination is valid.

Figure 8 illustrates the nodes that would be returned from the previous two SQL statements.

```
<Document>
 <Person>
  <Name>John Doe</Name>
  <City>Cambridge</City>
  <Age>25</Age>
  <Occupation>Janitor</Occupation>
 </Person>
 <Person>
  <Name>Cornelius</Name>
  <City>New York</City>
  <Age>35</Age>
  <Occupation>Investment Banker</Occupation>
 </Person>
 <Person>
  <Name>Jane Doe</Name>
  <City>Boston</City>
  <Age>19</Age>
  <Occupation>Student</Occupation>
 </Person>
</Document>
```

Figure 8: Nodes returned by complex query

5. Performance Analysis

In order to gauge the suitability and practicality of our indexing scheme, this section will will provide performance analyses, with respect to time, in three different categories: inserting, querying, and reindexing documents.

5.1 Inserting Documents

Before any data can be queried, the solution requires that a document first be inserted into the database and then be indexed. Refer to section 4.2 for the exact steps involved in this process. The table below shows the time required to insert documents of different sizes, independent of the size of an existing index. A total of four XPath queries were run on each document for indexing.

Number of Documents	*Total Size of Document(s)*	*Time taken to insert document(s)*
1	990 bytes	0.557 seconds
1	154,752 bytes	2.221 seconds
1	1,055,349 bytes	6.082 seconds
102	102, 413 bytes	10.051 seconds
174	178,176 bytes	16.721 seconds

Table 1: Time taken to insert and index documents

Table 1 shows numbers that are far from impressive. However, note that the inserting process only needs to be run once per document. In order to increase the performance in inserting documents, a solution would be to merge as many related documents into one. Table 1 shows how it actually takes less time to insert and index one large document than many documents that are cumulatively smaller.

5.2 Querying Documents

The performance of querying documents is the most important for it is the process that is most frequently run. Examples similar to those in section 4.3 were used to show the querying performance at different levels of complexity. All queries were run over a *node* table with 891 records and a *doc* table with 178 records. These records represent the data that is currently being used in one of the Metamedia projects.

The first query run is a simple one similar to that in section 4.3.1 with no subqueries. The second query is more complex (section 4.3.2 – context query) with three levels of subqueries. The last query is the most complex (section 4.3.3 – complex query), which involves three separate processes: inserting, querying, and deleting. The insertion has one subquery level and the actual query also has one subquery level.

Type of Query	*Time Taken*
Simple	0.016 seconds
Context	0.15 seconds
Complex	0.374 seconds

Table 2: Time taken for different queries

All of the numbers shown in table 2 are completely within acceptable levels. Even with the most complex query having three separate processes, it only took 0.374 seconds.

5.3 Reindexing Documents

Reindexing is a process that should not need to be run for as long as the developers are

careful about compiling the initial set of XPath's. Nevertheless, it is a process that must be supported and analyzed for it will most likely be used at some point. The following table shows the time it takes to reindex the same 178 documents in section 5.2.

Type of Reindex	*Time Taken*
Using every XPath in database (5)	10.623 seconds
One extra XPath	2.825 seconds

Table 3: Time taken to reindex documents

Much like inserting, reindexing documents is not a fast process. However, the same argument applies, for reindexing is process even less common than inserting.

5.4 Evaluation

Although it would have been ideal to have better performance figures for inserting and reindexing documents, the figures illustrate how the solution's query performance is well within practical boundaries. The purpose of indexing the documents with preset XPath's was to increase querying performance. Thus, it would be fair to say that it was expected to find poor numbers when inserting and reindexing documents for the idea was to take an initial performance hit to speed up querying.

6. Conclusion

Given the project requirements, budget constraints, and limitations in XML technology, the solution described in this thesis would be considered a success. It is currently being used by the Metamedia group as its XML storage and retrieval archive system. This section will review the pros and cons of the program and then explore its future.

6.1 Pros

The first and foremost advantage of this solution is that it provides critical features that were unavailable in the other programs used. The most important feature being the ability to perform joins in queries. Although it is a simple feature, joining allows for the creation of complex queries. As shown in section 4.3.3, it would be impossible to issue the same query without joins.

The second most beneficial advantage is the high querying speed that it provides over the conventional XML querying method of DOM parsing and XPath querying. The performance increase can be greatly attributed to the fact that the XML documents no longer need to be parsed into a DOM when queried. Instead, the DOM parsing takes place when the documents are inserted into the database.

Yet another advantage is the flexibility of the indexing scheme. By breaking down XPath queries to simpler ones and caching the results of those, it is possible to construct a much wider range of queries.

Lastly, there is the advantage of not modifying the XML documents when they are stored in the database. They are stored in their original form as text in the database. The advantage in this is that should the Metamedia group decide to upgrade to a different

database, the XML documents can be easily migrated over.

6.2 Cons

The majority of the shortcomings in this solution comes from the actual indexing process. All documents that are added to the database need to be parsed into a DOM so that the XPath queries stored in the database can be run on them. When a large number of documents are inserted, this indexing process can take a while, as shown in Section 5. This fault, however, is not as bad as one may initially think for the indexing process is run infrequently. Due to this shortcoming, it is essential that those who are responsible for populating the XPath table be especially wary when compiling the list of preset XPath queries. Otherwise, it may be necessary to reindex the entire database should an unforeseen XPath query be added to the XPath table. Such a situation is undesirable as there may be a large number of documents in the database that need to be reindexed.

Section 4.3 had a number of examples that showed how to convert an XQuery query to a SQL query. This conversion process is quite complicated and it must be done for every individual query. Although this weakness does not affect the performance of the program, it slows down the process of adding new queries. Similarly to the previous shortcoming, the addition of new queries is done infrequently, and thus the severity is not very high.

In addition, it is not possible to create queries that determine the level of ancestry between nodes as the *contains* function only determines whether or not a given node is a parent node of another. For example, the following two XPath's would return different nodes:

```
//a[./b]
//a[.//b]
```

The first XPath would return the nodes labeled "a" that have an immediate child labeled "b." The second, however, will return all the "a" nodes that have a descendant node labeled "b" at any level. The *contains* function would not be able to differentiate the two cases. For this reason, it is only possible to represent a subset of the XQuery syntax with our indexing scheme.

6.3 Future Considerations

Based on the performance analysis, the most obvious shortcomings would be the slow processes of inserting and reindexing. However, it is difficult to gauge the impact those performance figures have on a given project for it is uncertain as to how often the documents will be inserted or reindexed. Nonetheless, it would be beneficial to figure out a way to speed up these processes.

A shortcoming not as obvious as the slow inserting and reindexing is the process of converting XQuery's to SQL queries. Currently, this process is not automated as it is done by the developers. Depending on the complexity of the XQuery, the conversion process may take a long time. Furthermore, there is no guarantee that the translated SQL query returns the exact same results as the XQuery simply because the conversion is being done by humans. The developers of this indexing scheme is currently implementing a feature that would allow other developers to generate the complex SQL queries from a pseudo language that directly maps into XQuery's syntax.

In conclusion, the solution on which this thesis is based was largely developed for the purpose of providing the features of the unreleased XQuery language. Once XQuery is released and is integrated with other databases, this solution may be discarded unless additional features are added.

Appendix A – Namespaces

An XML namespace is a collection of names, identified by a URI (Universal resource identifier) reference, which are used in XML documents to specify the type of an element or attribute. The purpose of a namespace is to prevent "collisions" between elements or attributes that have the same local name, but are from different markup vocabularies. A namespace allows the document constructs to have universal names, whose scope extends beyond the document in which they are contained.

Elements or attributes that utilize a namespace appear as qualified names, which are composed of a namespace prefix, a local name, and a colon in between that separates them. The prefix selects a namespace as it is mapped to a URI reference. The combination of a namespace prefix with a local name provides an identifier for an element or attribute that is universally unique.

Consider the following XML document:

```
<root xmlns:a=http://www.a.com xmlns:b=http://www.b.com>
  <a:person>John</a:person>
  <b:person>Jane</b:person>
</root>
```

Two namespace prefixes, "a" and "b", are used. There are two "person" elements, but they are not the same as different namespace prefixes are used. In the situation that a software module looks through XML documents and needs to extract the "person" elements that refer to the URI, http://www.a.com, the namespace prefix "a" must be used. Otherwise, the software module would not be able to differentiate the different "Person" elements and would extract both John and Jane.

Appendix B – SQL Queries

The examples in section 4.3 used pseudo–SQL queries, which are simpler than the true expanded SQL queries, to help the reader understand the concept of the querying technique. This appendix will provide the equivalent expanded SQL queries to provide a better understanding of the implementation details.

Simple Query

Pseudo–SQL

```
SELECT get(node) FROM nodeTable WHERE path(node) = '//Person'
```

Expanded SQL

```
SELECT get(docid, position, length) FROM node WHERE path(docid, position, length) =
'//Person'
```

The two versions differ in only two ways. Firstly, every reference to a *node* is replaced with a *docid, position,* and *length.* The reason for this is because a node is represented by those three variables (see section 4.3.2). Secondly, the reference to *nodeTable* is replaced with *node* because *node* is the true name of the table in the relational database. The purpose of referring to *nodeTable* was to eliminate any confusion between the actual table and the nodes.

In addition, the *get* and *path* functions take in 3 arguments, instead of just one. The *get* function takes in three integers as arguments: *docid*, *position*, and *length.* It simply returns the substring for a given document, identified by the *docid,* by using the position argument as the offset and the length argument to specify the length of the substring.

The *path* function also takes in the same three arguments as *get*. The *path* function returns the XPath query string that was used to return the substring that is identified by the three arguments. This is possible for a given substring can only be obtained by one XPath query, or an identical one.

Context Query

Pseudo–SQL

```
SELECT get(nodeTable.node) FROM nodeTable,
        (SELECT node FROM nodeTable WHERE path(node) = '//Name' AND get(node) =
'Corenelius') AS Name
WHERE path(nodeTable.node) = '//Person' AND contains(nodeTable.node, Name.node)
```

Expanded SQL

```
SELECT get(node.docid, node.position, node.length) FROM node,
        (SELECT docid, position, length FROM node WHERE path(docid, position, length) =
'//Name' AND get(docid, position, length) = 'Corenelius') AS Name
WHERE path(node.docid, node.position, node.length) = '//Person' AND contains(node.docid,
node.position, node.length, Name.docid, Name.position, Name.length)
```

The same changes from the simple query example are here as well, except for the difference in the *contains* function. The *contains* function takes in six arguments (instead of two): a pair of *docid's*, *positions*, and *lengths*. The function checks to make sure that the *docid's* are the same. It then checks to see if the first substring contains the second substring with their respective *position* and *length* arguments.

Complex Query

Pseudo–SQL

```
SELECT get(Name.node), get(Age.node), get(Occupation.node) FROM
        (SELECT nodeTable.node FROM nodeTable,
                (SELECT node FROM nodeTable WHERE path(node) = '//City' AND
get(node) = 'Boston')
                AS City
        WHERE path(node) = '//Person' AND contains(nodeTable.node, City.node))
        AS Person
        JOIN

        (SELECT node FROM nodeTable WHERE path(node) = '//Name')
        AS Name ON contains(Person.node, Name.node)
        JOIN

        (SELECT node FROM nodeTable WHERE path(node) = '//Age')
        AS Age ON contains(Person.node, Age.node)
        JOIN

        (SELECT node FROM nodeTable WHERE path(node) = '//Occupation')
        AS Occupation ON contains(Person.node, Occupation.node)
```

Expanded SQL

```
SELECT get(Name.docid, Name.position, Name.length), get(Age.docid, Age.position,
Age.length), get(Occupation.docid, Occupation.position, Occupation.length) FROM
        (SELECT node.docid, node.position, node.length FROM node,
                (SELECT docid, position, length FROM node WHERE path(docid, position,
length) = '//City' AND get(docid, position, length) = 'Boston')
                AS City
        WHERE path(node.docid, node.position, node.length) = '//Person' AND
contains(node.docid, node.position, node.length, City.docid, City.position, City.length))
        AS Person
        JOIN

        (SELECT docid, position, length FROM node WHERE path(docid, position, length) =
'//Name')
        AS Name ON contains(Person.docid, Person.position, Person.length, Name.docid,
Name.position, Name.length)
        JOIN

        (SELECT docid, position, length FROM node WHERE path(docid, position, length) =
'//Age')
        AS Age ON contains(Person.docid, Person.position, Person.length, Age.docid,
Age.position, Age.length)
        JOIN

        (SELECT docid, position, length FROM node WHERE path(docid, position, length) =
'//Occupation')
        AS Occupation ON contains(Person.docid, Person.position, Person.length,
```

Occupation.docid, Occupation.position, Occupation.length)

References

[1] Bosak, Jon, XML, Java, and the Future of the Web. Sun Microsystems, 1997

[2] Flynn, Peter, The XML FAQ. http://www.ucc.ie/xml/, 2002

[3] Salminen, Airi, Requirements for XML Document Database Systems. Doc Eng, 2001.

[4] Obasanjo, Dare, An Exploration of XML in Database Management Systems. 2001

[5] W3C, XML Path Language (XPath). http://www.w3.org/TR/xpath

[6] W3C, XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, 2002

[7] Naude, Frank, Oracle Internet Filesystem (iFS) FAQ. http://www.orafaq.com/faqifs.htm, 2000

[8] Yoshikawa, Masatoshi and Amagasa, Toshiyuki, XRel: A Path–Based Approach to Storage and Retrieval of XML Documents Using Relational Databases, ACM Transactions on Internet Technology, Vol. 1, No. 1, August 2001, Pages 110–141

[9] Li, Quanzhong and Moon, Bongki, Indexing and Querying XML Data for Regular Path Expressions. The VLDB Journal, 2001