

X2S

A Major Qualifying Project Report:

Submitted to the Faculty

Of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

By

Richard Nordin

Date: March 15th, 2004

Approved:

Professor Murali Mani

Professor Daniel Dougherty

1. Databases
2. Language Translation

Abstract

In order for web applications to view and query a legacy relational database, an XML view of that database is often provided. This paper is the description of an algorithm that will convert XPath queries to SQL queries and translate the results of the query back into XML.

1.	BACKGROUND	5
1.1	Introduction	5
1.2	Introduction to XML	5
1.3	Introduction to Cot.....	8
1.4	Introduction to the XPath.....	12
1.4.1	“/” Operator: Path Axes	12
1.4.2	“//” Operator : Descendent Axes	13
1.4.3	“*” Operator : Wildcard Nodetest	14
1.4.4	“@” Operator : Attribute Axes	14
1.4.5	“[]” Operator : Node Predicate Operator.....	15
1.4.6	“:” and “;” : The IDRef and ID axes	16
1.5	Benefits of Presenting an XML Schema	17
1.6	Historical Solutions to the Problem	18
2.	BASIC ALGORITHM PROPOSAL	18
2.1	Psuedo-code.....	18
2.1.1	Pop the Leftmost XPath Axes	18
2.1.2	Interpret the Current Step	19
2.2	Tagging.....	21
2.3	Cases in Which DTD is not Required	22
2.4	Cases in Which DTD is Required.....	22
2.5	Design Goals.....	22
2.6	The Problem of Joins.....	23
3	METHODOLOGY	24
3.1	Tools and Software.....	24
3.1.1	NetBeans	24
3.1.2	Java 1.3 SDK.....	24
3.1.3	Transaction Processing Council: DBGEN.....	25
3.2	Testing Scenarios.....	25
3.1.1	Simple Test.....	25
3.1.2	TPC tests	25
4	EXTENSIONS	29
4.1	Write Operations.....	29

4.2	Complex Predicates	30
4.3	Multiple ID-Refs.....	30
4.4	Returning Child Nodes.....	31
4.5	Error Checking.....	32
APPENDIX		33
A.1	File Description.....	33
A.2	How to Recreate the TPC Test Scenarios.....	34
A.2.1	Load the TPC tables into Tablespace	34
A.2.2	Load the TPC data into Tables	34
A.2.3	Convert the Queries	35
A.2.4	Run the Timing Tests.....	35
BIBLIOGRAPHY		36

1. Background

1.1 Introduction

Given an XML view into a legacy relational database, it is necessary to provide support for XML query methods to query the underlying relational database. To that end, this paper is a description of an algorithm to convert XPath queries into good SQL; SQL free from redundant or unnecessary joins and retrieving the required information with the minimal amount of processing. A subset of the entire XPath language will be examined: the /, [], //, and *operators. Special attention will be paid to the node test operator, to cover the rich set of functionality available in XPath as well as the ID-Ref and reverse ID-Ref functions.

In a world where XML becomes increasingly used as the protocol for communication between applications and a method for storing data for ease of access, legacy databases want to share their information with web applications. It is too difficult to maintain separate copies of a database- one in an XML format and one in the legacy format. For this reason legacy databases will often provide an XML view into their data. The data will stay in the original databases and web applications will write XML queries against the view. The algorithm will translate those XML queries into SQL, query the database and then translate the return back into a form that the web application can understand.

1.2 Introduction to XML

XML stands for Extensible Markup Language and is a language used for describing the format and structure of electronic documents. XML is flexible enough so

that any document can be described: mathematical, geographical, or musical for instance. The user defines the grammar; essentially it is a language for describing languages. Originally derived from SGML, XML was designed in 1996 by the World Wide Web Consortium (W3C). Among its design goals were that it were that it was easy to use over the web, humanly legible, but easy for a computer to parse, as well as being easy to design and prepare [1]. It allowed users to break out of using a single inflexible document type, HTML, and allowed users to avoid using SGML, which was hard to program for. [2] The user defines how the document will look, and any parser written to handle XML can process the document, although there will not be any context for the data.

It's interesting to see the relationship that XML has to databases. As a "database format", XML has advantages and disadvantages. On the plus side, XML gains portability in the fact that it is Unicode. Its structure is rigidly defined by the use of Document Type Definitions (DTD), a sort of XML schema. On the minus side, it is plain text which is a very inefficient way of storing data. And it makes absolutely no attempts at being terse. One of the original design goals of the W3C [1] was to specifically ignore conciseness as a virtue.

XML really shines in being a protocol to communicate with databases. Web applications can easily understand it, and through the clever use of DTDs, it can be used to represent very complex data structures. The user will store their data in a very quick, streamlined database such as oracle, design a way for oracle to speak in XML, such as the CoT [3] algorithm which will be covered later, and design a way for a user to ask questions in XML that the database will understand. It's the last part of that equation that this algorithm is concerned with. It will provide a way for users to look at a database

structure in XML and query the actual data that sits somewhere in a database.

An XML document stores its data as a set of nested entities. Entities must have an opening tag, a closing tag, and this nested structure is rigidly enforced. Character data- the actual values of the entities, and other entities can be contained inside the tags. Entities can also have attributes, and all of these things, the structure of the entities, what children they have if any, what attributes they can have if any, are all defined by the creator of the XML document. Consider the following:

```
<Bank Name="Chase">
  <BankAccount Active="Yes" Name="Rich">
    <Balance>-5</Balance></BankAccount>
  <BankAccount Active="Yes" Name="Sally">
    <Balance>4563</Balance></BankAccount>
  <BankAccount Active="Yes" Name="John">
    <Balance>21232</Balance></BankAccount>
</Bank>
```

Fig. 1

Note that this is not the most ideal way of storing bank account information, but simply an illustration of how XML works. First note that the entire document is contained in once single outer node. XML files require that their contents strictly follow a tree format. There must only be one root node and all other nodes must be contained inside it. Also note how each node has an opening and closing tag and the opening and closing tags are nested correctly. That is, the open and the close are on the same depth. A DTD or “document type declaration”, describes the grammar of the document. A close relative to relational database schemas, DTDs describe “how many of what”, what order, and how they relate to one another. Like XML itself, DTDs have a specific language that must be followed. Consider below:

```
<!ELEMENT BANK (BANKACCOUNT)*>
<!ATTLIST BANK Name CDATA #REQUIRED>

<!ELEMENT BANKACCOUNT (BALANCE)>
<!ATTLIST BANKACCOUNT ACTIVE CDATA #IMPLIED NAME CDATA #REQUIRED>

<!ELEMENT BALANCE (#PCDATA)>
```

Fig. 2

The first line of the above specifies that there is some root element, BANK. This bank element has zero or more BANKACCOUNT elements in it and a Name attribute which is required. Furthermore, any BANKACCOUNT elements will not have exactly one balance child node, have a non-required attribute Active and a required attribute Name. The balance node will be filled with parsed character data. By luck or design, the XML introduced in Fig. 1 conforms to the DTD. The only thing that it is missing is the initial DOCTYPE declaration:

```
<!DOCTYPE Bank SYSTEM "bank.dtd" >
```

Fig. 3

This informs the consumer of the XML document that the document conforms to that particular DTD.

1.3 Introduction to Cot

CoT is an algorithm for creating an XML schema based on a relational database using foreign key constraints[3]. It's important to note that CoT does not manipulate data. It observes relationships between the tables in a relational database and tries to best fit that to a tree structured XML DTD.

CoT establishes the relationships between tables by observing which tables have foreign keys into one another and the allowable values of those foreign keys.

```
Student(SName)
Professor(PName, SNameRef#)
```

Fig. 4

If SNameRef is a foreign key into Student's primary key SName, CoT can make some assumptions about how the two tables relate. There are three main cases that CoT is concerned about. The first is if the foreign key is non-nullable and unique. In our above relational database schema if SName is a key of Student (and thus unique by definition), if SNameRef is not allowed to be null, and each Professor node must have a unique SNameRef, then by definition there has to be a one to one relationship between Student and Professor. This is displayed in an XML tree form by making Student a parent node of Professor Node.

```
<Student SName="">
  <Professor PName=""/>
</Student>
```

Fig.5

For ease of understanding, the above example is XML¹. Notice, also that the SNameRef is missing from the document. This is because it really only existed in the relational schema to establish a relationship between the two tables. In XML, that relationship is well illustrated by having the one node encapsulated within the other. The CoT algorithm in its original implementation strips away these foreign keys, which can cause problems as we'll see later.

The second case that CoT is interested in is when the foreign key is non-null, but does not have to be unique. This means that there can be several professors that reference the same Student. It is not enough to make Professor a child of Student again. For this

¹ The output of CoT is a DTD. In all of the future examples in this section, I will be showing an XML document that conforms to the DTD that CoT would generate, for ease of understanding. For more information on DTDs, please refer to section **1.2 Introduction to**

case, we de-parent the Professor node, but put in an id-ref:

```
<Student SName=""/>
<Professor PName=""/>
Fig.6
```

It would appear from the above that there is no relationship between the two nodes, but the underlying DTD is where that information is preserved:

```
<!ELEMENT Student (EMPTY)>
<!ATTLIST Student SName CDATA #REQUIRED SID CDATA #ID>
<!ELEMENT Professor (EMPTY)>
<!ATTLIST Professor PName CDATA #REQUIRED SID-REF CDATA #IDREF>
Fig.7
```

The third case that CoT is concerned about is when the foreign key is “nullable”.

In our example:

```
Student(SName)
Professor(PName, SNameRef#)
Fig.8
```

This implies that Professor can be a child of Student, but doesn't have to be. XML can not represent this by having one node inside the other, id-refs will have to be used again.

Furthermore, if a table has a foreign key reference to two different tables, and both foreign key references are non-null and unique, CoT will choose one to parent the node to, and create an id-ref to the other.

Pulling all these things together, consider the example below:

XML.

```

Cat(CName, ONameRef (non-null,unique), CCatNipRef(non-null, unique))
Dog(DName, ONameRef (non-null, unique))
Goldfish(GName, ONameRef(non-null))
Owner(OName, OStreetAddr, OStateAddr)
CatNip(CCatNipValue)

```

Fig. 9

The CoT solution will be to do this:

```

<!ELEMENT Cat (Empty)>
<!ATTLIST Cat CName CDATA #IMPLIED CCatNipRef CDATA #IDREF>

<!ELEMENT Dog (EMPTY)>
<!ATTLIST Dog CName CDATA #IMPLIED>

<!ELEMENT Goldfish(EMPTY)>
<!ATTLIST Goldfish GName CDATA #IMPLIED ONameRef CDATA #IDREF>

<!ELEMENT Owner (Cat, Dog)>
<!ATTLIST Owner OName CDATA #IMPLIED OStreetAddr CDATA #IMPLIED OStateAddr CDATA
#IMPLIED

<!ELEMENT CatNip (EMPTY)>
<!ATTLIST CatNip CCatNipValue CDATA #ID>

```

Fig.10

Notice that Owner “owns” cat and dog, and owns goldfish by idref. Cat happened to get assigned to Owner this time, but subsequent runnings of the CoT algorithm are not guaranteed to generate the same outcome. This can cause trouble as we’ll see later. Since our XML schemas and documents are coming from scanning relational databases with the CoT algorithm, assumptions that can be made about the document that are not implied by XML in general.

As shown above, no child node will have two parents. A side effect of CoT is also that it removes recursion. This guarantees that our XML DTDs create a tree in the strictest sense of the term. There is a finite, fixed length path from root node to any other given node.

1.4 Introduction to the XPath

1.4.1 “/” Operator: Path Axes

This operator is similar to the absolute path operator in file systems: “/usr/bin”. The first instance of the backslash specifies the root of the document, and subsequent instance can traverse down the document tree to select entire groups of children. For instance if we have the document:

```
<!ELEMENT Student(Professor)>
<!ATTLIST Student DormRoom CDATA #IMPLIED SName CDATA #IMPLIED >
<!ELEMENT Professor(EMPTY)>
<!ATTLIST Professor PName CDATA #IMPLIED>

<Student DormRoom="" SName="">
  <Professor PName=""/>
</Student>
```

Fig.11

Then the XPath query “/” would select the entire document. The XPath query “/Student/Professor” brings up an interesting point. Although our XML schema exists in a very structured form of parents and children, it's important to remember that the underlying relational database do not. In this case, the user wants the Professor node and every single node and attribute it contains. My algorithm takes a “lazy” view of this. If the Professor Node had additional children, retrieving the children would require multiple requests of the underlying relational database. One of my design goals is to minimize these requests so for now we will just return the Node itself.

In the above example, where Professor contains no children nodes, the SQL command would be “SELECT * FROM NodeName;”. It's important to note that no matter where the final node is, no matter how deep it may be, the ‘/’ operator requests nothing about the parents. What that means in terms of our optimizations is that our SQL

translation doesn't need to do any joins of the relations to the left of the requested one.

We can just throw them out.

1.4.2 “//” Operator : Descendent Axes

This operator specifies that the following path can occur anywhere in the document. Using our example above:

```
<!ELEMENT Student(Professor)>
<!ATTLIST Student DormRoom CDATA #IMPLIED SName CDATA #IMPLIED >
<!ELEMENT Professor(EMPTY)>
<!ATTLIST Professor PName CDATA #IMPLIED>

<Student DormRoom="" SName="">
  <Professor PName=""/>
</Student>
```

Fig.12

The command //Professor would return the Professor node. Likewise, the //Student/Professor or /Student//Professor would return the Professor node.

We've come to an interesting point again. Because of the way CoT translates relations, we are assured there is a finite, fixed length path from root node to any other given node.

In terms of the ‘//’ operator looking at a generic XML document, this is not the case.

Consider the generic non-CoT conforming XML document below:

```
<A>
  <B/>
  <C/>
  <A/>
  <D/>
  <C/>
<A>
```

Fig.13

“/A/C” and “/D/C” are two very different requests. The first one will chose any C node that has an A node somewhere above it. The second one will choose any C node

that has a D as a ancestor. This is a complicated pathing algorithm that we can ignore. CoT guarantees us that if there is a C anywhere in the document, it can only have one parent. If C has foreign-keys into other tables, it will be represented by an id-ref. Thus /n//C can only mean one thing, no matter what n is: “select * from C”. In order to determine and properly error-check a ‘//’ operator in the XPath to be translated, some pre-processing will occur. “/x/y//z” will be expanded to an expression such as ‘/x/y/a/b/z’ depending on the true path to z. This will help us when it comes to “sub-setting” later.

1.4.3 “*” Operator : Wildcard Nodetest

This operator does not select the node, it selects only the children of that node. For instance:

```

<!ELEMENT Student(Professor, Address)>
<!ATTLIST Student DormRoom CDATA #IMPLIED SName CDATA #IMPLIED >
<!ELEMENT Address (EMPTY)>
<!ATTLIST Address Street CDATA #IMPLIED State #CDATA IMPLIED>
<!ELEMENT Professor(EMPTY)>
<!ATTLIST Professor PName CDATA #IMPLIED>

<Student DormRoom="" SName=""
    <Professor PName=""/>
    <Address Street="" State="">
</Student>

```

Fig.14

/Student/* would select the Professor node and Address node, whereas /Student/Professor would select only the Professor node.

1.4.4 “@” Operator : Attribute Axes

This operator will select the attribute specified. For instance:

```

<!ELEMENT Student(Professor)>
<!ATTLIST Student DormRoom CDATA #IMPLIED SName CDATA #IMPLIED >
<!ELEMENT Professor(EMPTY)>
<!ATTLIST Professor PName CDATA #IMPLIED>

<Student DormRoom="" SName="">
  <Professor PName=""/>
</Student>

```

Fig.15

/Student/@SName will return SName of all of the students, and
//Professor/@PName will return the PName attribute of all the Professor Nodes. In terms
of our SQL, the translation would be “SELECT SName FROM Student;” or “SELECT
PName FROM Professor;” respectively.

1.4.5 “[]” Operator : Node Predicate Operator

The Node Predicate Operator corresponds very closely to the where clause in
SQL.

```

<!ELEMENT Student(Professor)>
<!ATTLIST Student DormRoom CDATA #IMPLIED SName CDATA #IMPLIED >
<!ELEMENT Professor(EMPTY)>
<!ATTLIST Professor PName CDATA #IMPLIED>

<Student DormRoom="34" SName="Rich">
  <Professor PName="Mani"/>
</Student>
<Student DormRoom="23" SName="John">
  <Professor PName="Greg"/>
</Student>
<Student DormRoom="32" SName="Sally">
  <Professor PName="Dan"/>
</Student>

```

Fig.16

/Student[@Dormroom='32']/ would only choose the bottom node, the one with
sally in it. Thus the translation to SQL would be “select * from student where dormroom
= 32”. XPath queries can have quite complicated predicates in them:

“/Student[Professor/Office/@Desk=23]/@SName.” These predicates require a separate XPath parsing just to verify that they are even legal, so the algorithm ignores them for now. XPath also implements a rich set of node-test functions such as Name(), StartsWith(), Count(). Many things can be requested in XPath that either make no sense in SQL or SQL cannot express, for instance: /Student/starts-with(name(),'P') is another way of selecting the Professor Node.

The predicate operator is our first case of “sub-setting”. Previously, we could ignore all of the XPath command to the left of the interesting bit. For instance: /a/b/c/d/e/@f really just translates to “select f from e”. CoT tells us that every “a” has a one to one relationship with every “b” and so on. The /a/b/c/d/ portion is really just so much useless pathing information. This changes when you subset.

Consider:

“/a[@d=3]/b/@c”

Now we are no longer interested with EVERY single a, but a smaller subset of them. Since every “a” has a one to one relationship to every “b”, we are interested in a small subset of “b’s” as well. We are going to have to do our first join. We will specify that we are only concerned with those “b’s” that have a reference to “a’s” that follow the predicate. Of that subset of b’s, give us the c attribute. This sets up sort of an idea of state for our future parsing algorithm. We are either in a join condition, or a non-join condition depending on the presence of a predicate in our XPath.

1.4.6 “:” and “;” : The IDRef and ID axes

You will not find a reference to these in any XPath specification. In XPath, there is a function to follow IDREF and ID paths, but it doesn’t really mesh well with the other

axes. The algorithm deals with that by expecting idref axes. This will be identical to calling idref() on the idref attribute of the current Node and will select the node that this node id-refs. Id would be dealt with in the same way. For instance:

```
<!ELEMENT Student(Professor)>
<!ATTLIST Student DormRoom CDATA #IMPLIED SName CDATA #ID>
<!ELEMENT Professor(EMPTY)>
<!ATTLIST Professor PName CDATA #IMPLIED>
<!ELEMENT Car (EMPTY)>
<!ATTLIST Car CName CDATA #IMPLIED SName-Ref #IDREF>
```

Fig.17

This DTD could have easily arose if Car had a foreign key reference to Student, but was nullable (not every car is owned by a student). In order to get Students that own Cars, //Cars:Students/ would be called. To get Cars that are owned by Students /Students;Cars would be called. In both cases are “subsetting” is going on. All XPath to the right of an “;” or an “:” would require a join between every relation in the XPath query.

1.5 Benefits of Presenting an XML Schema

There are various reasons why it makes sense to deal with DTDs and XML views, even though one could “translate” the entire database into XML. XML “views” into relational databases make a lot of sense for a couple of reasons. The first one is, the underlying database might have a lot of information in it. Translating, tagging and transmitting this to a web application is too time consuming, and doesn’t make sense. The user could be only interested in a small portion of that data. A request for a single attribute might require a transfer of a couple of gigs. Secondly relational databases and relational database optimizations have been studied for many years. Methods of quickly storing data and quickly looking up data once stored have been on various database

applications currently available on the market. There's just no reason to "re-invent the wheel". Instead of giving a web application a database, it is given an idea of what the database "looks like". Based on this, the user can make small, specialized requests.

1.6 Historical Solutions to the Problem

A simple way of solving the problem that has been suggested is to join all the relations together on their foreign keys and make the request[6]. This is inefficient and unnecessary: the CoT algorithm through its observations about relationships of parent and child constructs an XML view that is "easy to query". By representing non-nullable unique, non-nullable non-unique and nullable table relationships in separate ways, one can make optimizations based on the type of query the user passes. Previous work has not made the distinction between the three types, indeed, the only reason why my algorithm can succeed is because both directions are well defined.

2. Basic Algorithm Proposal

XPath lends itself to a recursive algorithm very nicely. The algorithm will traverse the query, popping "axes" off, writing nested SQL sub queries if necessary and passing the beheaded XPath query and SQL back into the algorithm.

2.1 Pseudo-code

2.1.1 Pop the Leftmost XPath Axes

First, the algorithm pops the left most step off of the XPath query. It is important to note that the algorithm will need at most two axes in a row. This is because X2S can never be perfectly recursive: the translation from XPath to SQL requires one to be able to traverse back and forth between the steps. When considering the XPath query:

/a/b/@c

The first axes can be thrown away, as it is implicit in the query that there is at least a one to one relationship between b nodes to a nodes. If the second step is thrown away while processing the third, however, it will be impossible to determine the FROM clause to SELECT the C attribute from.

2.1.2 Interpret the Current Step

2.1.2.1 No Join Condition

A join condition is when one of the previous steps in the XPath query causes a subset of the entire database to be considered. Some XPath steps are simply “directional” in nature: “how to get from Node A to Node C“, while others are “subsetting” in nature: predicates and idrefs. Include example about what subsetting means. See **2.5 Design Goals** for more information.

a) Attribute

If the current step is an attribute, the previous step must be retrieved. Attributes are meaningless without the “from” clause implied by the previous step.

b) Predicate

Currently, X2S only evaluates simple predicates. That is, it only evaluates predicates where the user is looking for the value of a certain attribute. In XPath, predicates can get quite complicated:

/a[b/c/@d=3]/b

The above means select * from b where the child b that has a child c that has an attribute d equal to the number 3. In order to handle these types of predicates, the algorithm needs the DTD. The algorithm also does not support predicate functions. XPath

has a rich set of functions to test nodes for certain conditions. Not all of these functions can be accurately translated to SQL, and those that can, can be very complicated. For instance, bring up the position function

c) Simple Step

In the case of a simple step, the algorithm will just throw it away. For instance:

`/a/b/@c`

translates to: find all b's that are children of a, and retrieve their c attributes. Since by the XPath statement itself, all b's are implicitly children of a's, we really mean just take all b's and retrieve their c attributes.

2.1.2.2 Join Condition

If there was a join condition in one of the previous XPath steps, we have to do joins on each of the steps.

a) Attribute

Now, we must only consider the subset of children that are joined to the steps to the left of this step. For instance:

`/a[@b=3]/c/@d`

We are looking at a subset of all the a's, and the cs which are children of that subset. We will have to do a join on the two relations; the SQL will be "select d from (c join {select * from a where b = 3} on FK and K)". Where k is a primary key for the relation a and fk is a foreign key of the relation c that points to a. If we add any intermediate steps, those relations will have to be joined in a similar way. My algorithm parses through the "extended" DTD to find the underlying SQL foreign key and key names that the two relations share.

b) Predicate

Multiple predicates in the same XPath query are allowed, of course. For instance, the following XPath query:

```
/a[@b=3]/c[@d=4]/e
```

is looking to find those e nodes which are children of c nodes with the attribute d equal to four and similarly are children of a nodes with the attribute b equal to 3. As in simple steps above, the two steps are joined on the foreign key and key from the DTD.

2.1.2 Tag Returns

The final step of the algorithm will be to tag the data that is returned from our SQL theory. A best case scenario would be to match this data's schema to the schema we have already established for the relational database. Unfortunately, repeated running of the CoT algorithm is not guaranteed to return the exact same schema. For this reason, I have decided to tag the return values flat.

2.2 Tagging

SQL queries can return either a single attribute or columns from a relation. These columns will need to be converted to XML. Since SQL queries often result in virtual, unnamed tables, all results will be returned with the <ROOT> tag as the base tag.

Attributes will be tagged with their attribute name. For instance, the XPath Query /a/@b will return:

```
<ROOT>  
  <b>Value</b>  
</ROOT>
```

Fig.18

The XPath Query /c/d/e will return something along the lines of :

```
<ROOT>
  <E Attribute1="Value" Attribute2="Value"/>
  <E Attribute1="Value" Attribute2="Value"/>
  <E Attribute1="Value" Attribute2="Value"/>
</ROOT>
```

Fig.19

2.3 Cases in Which DTD is not Required

In the case of very simple XPath queries, it is not necessary to have the DTD in front of us to translate them. A simple XPath query can be thought of as any XPath query that does not contain joins, the // operator or the IDRef operators.

2.4 Cases in Which DTD is Required

While traversing an XPath query, predicates and IDReffed nodes cause us to consider a subset of the child nodes. Its important therefore to only consider those child nodes that are joined to the subset. In essence, every axes after the predicate or RefID requires a join. In order to do the join, we need to know the foreign key and the referencing id from the two tables. This information is encoded in the DTD that will be returned by the CoT algorithm. The // operator also poses a special problem. We need to be able to traverse the XML view of the database to translate the // query into a corresponding / query.

2.5 Design Goals

The primary SQL optimization that my algorithm does is avoiding unnecessary joins. At the risk of stating the obvious, SQL queries return subsets of the entire database. Each step in an XPath query might or might not narrow down the subset. The trick is evaluating each step to decide if it will “cause a subset” or not. For instance, the

XPath query:

/a/b/c

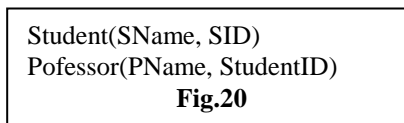
is asking for “all c’s which are children of all b’s which are children of a’s.” Since it is implicit that all c’s are children of b’s and all b’s are children of a’s by the query itself, we know that we do not need to do joins on each of the three relational tables. We can simply consider the set of all c’s. In another case:

/a[@d=3]/b

We are first considering the subset of all a’s who have the attribute d equal to 3. Of that subset, we are considering only the b’s that have one of those a’s as a parent. A join will be necessary. Furthermore, each additional step beyond the predicate step will require a join. It can be said that reading an XPath query from left to right, all steps can be ‘thrown out’ so to speak until reaching a predicate, search operator or idref operator.

2.6 The Problem of Joins

Any XPath operation that returns nodes with children will require a join. CoT in its current implementation hides the foreign keys of relational database tables because they’re implicit in the hierarchical structure of XML. In order to run join operations on relational database, CoT will need some way of communicating those foreign keys to the translation algorithm. The method for this will be Annotated DTDs. Consider:



Where ProfessorID is a foreign key that references PID. This would normally produce a DTD like:

```
<!ELEMENT Student(Professor)>
<!ATTLIST Student SName CDATA #IMPLIED>
<!ELEMENT Professor(EMPTY)>
<!ATTLIST PName>
```

Fig.21

A request for /Student would require the knowledge of the foreign key. We will modify the CoT algorithm to specify this.

```
<!ELEMENT Student(SName, Professor)>
<!ELEMENT Professor(PName) – Student/Professor PK = “SID” FK = “StudentID”-->
```

Fig.22

3 Methodology

3.1 Tools and Software

The algorithm’s proof of concept program “X2S” was written using the Netbeans IDE, JDK 1.3, and a free java package, dtdparser.

3.1.1 NetBeans

NetBeans is a completely free open-source IDE created by Sun Microsystems. It uses the Java 1.3 SDK to compile the code, and offers a rich set of functionality such as JavaDoc tool tips, quick packaging tools, and an easy to use debugger.

3.1.2 Java 1.3 SDK

Java was chosen because of the ability to rapidly develop programs, its portability and its rich built in string manipulation classes. It allowed me to code the proof of concept algorithm on my personal PC, but then display the results to my advisors on their sun stations.

3.1.3 Transaction Processing Council: DBGEN

The Transaction Processing Council, or TPC for short, has created a standardized sample database schema. The tool DBGEN allows a user to create randomized data that is guaranteed to be consistent in terms of foreign key connections. Then using the queries provided by the TPC, a user can see how his database stacks up in terms of the professional ones already on the market. In terms of my algorithm, I used the CoT algorithm to convert the schema into XML. Then I translated the SQL queries by hand into XPath queries. I then used my proof of concept java program, X2S to translate these queries back into SQL. Using the “timing” function provided in SQL, I timed a hundred runs of each query.

3.2 Testing Scenarios

3.1.1 Simple Test

First the algorithm was tested against a simple, hand written schema of two nodes:

```
<!ELEMENT Student(Professor)>  
<!ATTLIST Student SName CDATA #IMPLIED>  
<!ELEMENT Professor(EMPTY)>  
<!ATTLIST Professor PName CDATA #IMPLIED>
```

Fig.23

The following XPath queries were tested:

/Student

/Student/Professor

/Student[SName = 'Rich']/Professor

3.1.2 TPC tests

The TPC has created a group of standardized tests and a random data generator to run them against. The user specifies how much data they want, and TPC’s DGBEN utility will create data for a relational database for a predetermined schema. The data is guaranteed to be consistent in terms of foreign key relationships. To test my algorithm, I

used a megabyte of this generated data, and selected three queries from the set of queries they provide. These queries had to be modified in two ways. First, I was using SQLplus which does not support the full set of SQL queries that DBGEN expects and secondly, not all SQL queries make sense in terms of an XML view of a relational database. That is to say that XPath and SQL intersect, but the subset of XPath that we consider is a much smaller set of functionality than SQL such as aggregation and count (Xquery is a complete query language that supports all of SQL functionality).

3.1.2.1 Test 1

```
select
  l_returnflag
from
  lineitem
where
  l_shipdate <= '1998-12-01';
```

Fig.24

This is a simple SQL query of one table with no joins. Based on the DTD generated by the CoT algorithm, the XPath query was determined to be:

```
/lineitem[@l_shipdate <= '1998-12-01']/@l_returnflag
```

After running my algorithm on it, it created the SQL

```
SELECT L_RETURNFLAG FROM LINEITEM WHERE L_SHIPDATE <=
'1998-12-01';
```

which is identical to the original SQL and no timing tests were required.

3.1.2.2 Test 2

```

select
    PS_AVAILQTY
from
    PART,
    PARTSUPP
where
    P_PARTKEY = PS_PARTKEY and
    P_NAME = 'goldenrod lace spring peru powder';

```

Fig.25

This test was a bit more interesting. It requires the join between the two tables Part and Partsupp. Based on the XML generated by the CoT algorithm, the XPath query was determined to be:

```

/PART[@P_NAME = 'goldenrod lace spring peru
powder']/PARTSUPP/@PS_AVAILQTY

```

After running my algorithm on it, the SQL query was determined to be:

```

SELECT
    PS_AVAILQTY
FROM
    (SELECT
        *
    FROM
        PARTSUPP
    JOIN
        (SELECT
            *
        FROM
            PART
        WHERE
            P_NAME = 'goldenrod lace spring peru powder')
    on
        P_PARTKEY=PS_PARTKEY);

```

Fig.26

Taking into consideration load on the machine that I was running my test on, both queries completed in the same time.

3.1.2.3 Test 3

```
SELECT
  L_COMMENT
FROM
  REGION,
  NATION,
  CUSTOMER,
  ORDERS,
  LINEITEM
WHERE
  R_REGIONKEY = N_REGIONKEY AND
  C_NATIONKEY = N_NATIONKEY AND
  O_CUSTKEY = C_CUSTKEY AND
  L_ORDERKEY = O_ORDERKEY AND
  R_NAME = 'AFRICA';
```

Fig.27

Again, we expand the test to require a 5 table join. Based on the XML schema created by CoT, the XPath query is determined to be:

```
/REGION[@R_NAME='AFRICA']/NATION/CUSTOMER/ORDERS/LINEITEM/@L_COMMENT
```

And my algorithm's conversion back to SQL:

```

SELECT
  L_COMMENT
FROM
  (SELECT
    *
  FROM
    LINEITEM JOIN
    (SELECT
      *
    FROM
      ORDERS JOIN
      (SELECT
        *
      FROM
        CUSTOMER JOIN
        (SELECT
          *
        FROM
          NATION JOIN
          (SELECT
            *
          FROM
            REGION WHERE _NAME='AFRICA')
          on R_REGIONKEY=N_REGIONKEY)
          on N_NATIONKEY=C_NATIONKEY)
          on C_CUSTKEY=O_CUSTKEY)
          on O_ORDERKEY=L_ORDERKEY);

```

Fig.28

Although it looks quite different than the original SQL, the two queries took the same time and are functionally equivalent.

4 Extensions

In this section, we outline possible extensions for future work.

4.1 Write Operations

Although the above algorithm works very well for read operations performed on an XML view, it might be useful to some users to be able to make modifications to the underlying relational database. This is not a trivial problem. First, very strict pre-processor error checking will have to be introduced. XPath commands will not only have

to be validated for syntax, they will also have to be validated logically: Can the user actually make a path to the given node in the way that they specify?

Authentication and privileges pose a problem as well. In its current incarnation, X2S does no authentication of any kind. Obviously, database managers will want to restrict who has access to be able to modify their databases. Would plain text passwords be sufficient, or would there need to be encoded HTTPS?

After the command gets translated to SQL and processed, the return will need to be tagged and transferred back to the user. How return values should appear to the user, and more importantly, how error values should appear is an interesting problem in and of itself.

4.2 Complex Predicates

In its current incarnation, my proof of concept can only handle simple predicates. An example would be:

```
/a/b[@c=3]/@d
```

XPath predicates can become very complicated however: Nested predicates and predicates where the actual attribute value is several axes deep, for example. With minor modifications, the algorithm could work for these. Separate instances of the parser would be spawned off to handle each complex predicate. These would translate into nested SQL queries.

4.3 Multiple ID-Refs

DTDs as they are defined now can have multiple id-refs into different nodes. The drawback is that there is no way to specify which id-ref is referring to which node. For example:

```

<!ELEMENT A (EMPTY)>
<!ATTLIST A AData CDATA #IMPLIED Ref CDATA #IDREF>

<!ELEMENT B (EMPTY)>
<!ATTLIST B BData CDATA #IMPLIED Id CDATA #ID>

<!ELEMENT C (EMPTY)>
<!ATTLIST C CData CDATA #IMPLIED Id CDATA #ID>

```

Fig.29

The AData could have an id-ref into B or C, but there is no way to currently specify which one. The solution is through the usage of DTD comments. These will be ignored by any rendering programs or DTD validators, but will give our parser a chance of establishing a relationship between the two Nodes. The enhanced version of the DTD would look like this:

```

<!ELEMENT A (EMPTY)>
<!ATTLIST A AData CDATA #IMPLIED Ref CDATA #IDREF>
<!-- A.Ref refers to B.Id-->

<!ELEMENT B (EMPTY)>
<!ATTLIST B BData CDATA #IMPLIED Id CDATA #ID>
<!-- B.Id refers to A.Ref -->

<!ELEMENT C (EMPTY)>
<!ATTLIST C CData CDATA #IMPLIED Id CDATA #ID>

```

Fig.30

4.4 Returning Child Nodes

In XPath, when a user specifies that they would like /A/B, they are implicitly asking for the B node, all of B's attributes and any children B might have. In the XML world, this type of query makes sense: If B had a child C, asking for B and getting C back as well would not require any extra work. Since our XML represents underlying relational databases, it is important to note that retrieving the C node and any of its children would

require many more calls to SQL. Since one of the design goals of this algorithm was to create fast SQL, child nodes have been ignored. A good enhancement to the algorithm would be to provide a method for the user to specify whether they wanted all of the child nodes or not. Consider the following:

```
<!ELEMENT Student(Professor)>
<!ATTLIST Student DormRoom CDATA #IMPLIED SName
CDATA #IMPLIED >

<!ELEMENT Professor(EMPTY)>
<!ATTLIST Professor PName CDATA #IMPLIED>
```

Fig.31

If the user asks for /Student, the algorithm currently interprets that as “select * from Student”. To better match the true intent of the XPath query, we should really return select * from {Student join Professor on FK and K}. This could be accomplished with the usage of a command line switch to specify the intent of the users query.

4.5 Error Checking

Currently, the algorithm does not check to see if the XPath query was worded correctly, beyond the presence of the proper foreign keys. Many XPath parsers, both commercial and open source could be adapted to preprocess the query for correctness. Then the query would have to be checked to see if it falls within the subset of allowable XPath commands. Finally having passed all of these, the error conditions of the underlying SQL database would have to be converted back into good XML and returned to the user.

APPENDIX

A.1 File Description

Included in the back of this paper is a CD with the implantation of the algorithm. Below are a description of the files on that CD.

1.xpath 2.xpath 3.xpath	Hand written xpath translations of the TPC sql queries.
1.sql 2.sql 3.sql	The original TPC sql queries, modified to run with sqlplus. These queries are copied a hundred times to make the timing tests a bit easier to run.
1.x2s 2.x2s 3.x2s	X2S' translation of the xpath queries, copied a hundred times to help with timing tests.
cotscript	A shell script running the CoT algorithm on the TPC tables
dataloader	A shell script to load the TPC data after the TPC tables have been established
dbgen	The random data generator, made by TPC.
dtddparser\	Free java DTD parser package, created by University of Darmstadt
customerloader.ctl lineitemloader.ctl nationloader.ctl ordersloader.ctl partloader.ctl partsuploader.ctl regionloader.ctl supplierloader.ctl	These are the SQL loading scripts that will load the data from dbgen into sqlplus after it has been generated
customer.tbl lineitem.tbl nation.tbl orders.tbl partsupp.tbl part.tbl region.tbl	These are the output of dbgen after running my tests. They contain random data that is guaranteed to be relationally accurate. Foreign keys will match up to keys in other tables.

supplier.tbl	
Schema.gmr	This file is the output of running the CoT algorithm on the relational databases created by TPC. It is not yet in DTD format, it must be hand translated
TPH.dtd	The hand translated version of the output of CoT.
table.sql	A SQL script to load the table structure into SQL.
X2S\	This is the java package of my proof of concept. It contains the .java and .class files of all the java classes I created.
xerces-2_6_2\	The xerces XML parser. It is used by CoT and dtdparser.
CoT\	Murali Mani et.al's CoT java package

A.2 How to Recreate the TPC Test Scenarios

My proof of concept was designed to give people who wanted to duplicate my work a very modular step by step approach.

A.2.1 Load the TPC tables into Tablespace

Change directory into the directory that my work is in, and log into sqlplus using your username and password. Then execute the following command

```
@table
```

You should get the output that 8 tables were dropped and 8 tables were created. You now have the 8 empty TPC tables loaded. Quit out of sqlplus.

A.2.2 Load the TPC data into Tables

At this point, you can execute dbgen to create enough as much data as you want to load, or you can use the data that I have provided. If you decide to do the former, please look at www.tpc.org for more information. Otherwise, execute the following at the command prompt:

```
./dataloader
```

This will load all of the data in the .ctl files into your tables, making sure that they

make sense in terms of foreign key references.

A.2.3 Convert the Queries

Again, you can write your own xpath queries against the data, or use the ones that I have provided. If you decide to write your own queries, execute the following command:

```
Java X2S (XPath query) TPH.dtd > filename.x2s
```

Remember to surround the XPath query in quotes if it contains a quoted predicate. At this point, I like to go into filename.x2s, strip out the SQL = part and copy the SQL query a hundred times. This assures that a fast lookup will generate meaningful data.

A.2.4 Run the Timing Tests

Log back into sqlplus and execute your original sql statements a hundred times (or use the ones I have provided)

```
@1.sql
```

This will give you a readout as to how much time the query took to run. Then run your x2s version.

```
@1.x2s
```

Compare the two queries.

Bibliography

[1] W3C *XML Specification* <http://www.w3.org/TR/2004/REC-xml-20040204/>

[2] Flynn, P. *XML FAQ*, World Wide Web 2003, <http://www.ucc.ie/xml/>

[3] Bourret, R. *XML and Databases*, World Wide Web 2003,
<http://www.rpbouret.com/xml/XMLAndDatabases.htm>

[4] Mani, M., Dongwon, L. Chiu, F., Chu, W. *NeT and CoT: Translating Relational Schemas to XML*, International Conference on Knowledge and Information Management, 2002

[5] Zvon, O. *XPath Tutorial*, World Wide Web 2004,
<http://www.zvon.org/xxl/XPathTutorial/General/examples.html>

[6] Sicui, D. *On Database Theory and XML*, SIGMOD Record, VOL.30 No.3

[7] Zvon, O. *XPath Tutorial*, World Wide Web 2004,
<http://www.zvon.org/xxl/XPathTutorial/General/examples.html>