XML-QE:
A Query Engine for XML Data Soures

Bruce Jackson, Adiel Yoaz
{brucej, adiel}@cs.wisc.edu

1. Introduction

XML, short for eXtensible Markup Language, may soon be used extensively for exchanging information over the internet. There is currently no standardized query language for expressing queries over XML data sources, but there are proposed languages. Examples include XML-QL(2), Lorel(1), and Microsoft's proposed XQL. It is only a matter of time before one of these languages (or another) emerge as a standard query language for XML. Whatever language evolves, it will express queries that will probably be implemented as a series of operators as a query plan.

In this document, we describe the design an implementation of a query engine capable of producing valid XML documents as the result of executing a query plan over an XML data source (a set of XML documents). The query engine takes a text query plan as input. This plan is expressed in our self defined query plan language. We have designed this language and the underlying query engine to be flexible enough so that the plan could be produced from whatever XML query language evolves.

The remainder of this document is layed out as follows. Section 2 discusses the general system design and architechture. Section 3 describes our query plan language syntax. In Section 4 we describe the parser for this query plan language. Section 5 describes the query execution engine. Section 6 discusses the implementation of operators. Predicate evaluation is discussed in Section 7. Issues relating to DTD Construction is discussed in Section 8. Section 9 breifly describes the graphical user interface (GUI). And finally we conclude in Section 10 final with comments about the project general performance of various parts of the system

2. System Architecture

The query engine has three phases - plan parsing, data loading phase and the execution phase. The plan parsing phase reads the text file containing the plan, parses it, and creates a hash table of the

operators in the plan. Each operator is associated (by hashing) with it's output buffer ID. The load phase recursively instantiates the operator tree, and passes to each operator references to its source operators. During that phase each operator records the parameters it needs during run-time, and also performs any required initializations. During this phase the scan operator calls on the Data Manager to open the input XML files, and also parse and validate them using the IBM XML4J validating parser. This parser creates an in-memory representation of the XML documents, which can be manipulated using a DOM (Document Object Model) compatible interface. The execution phase starts all

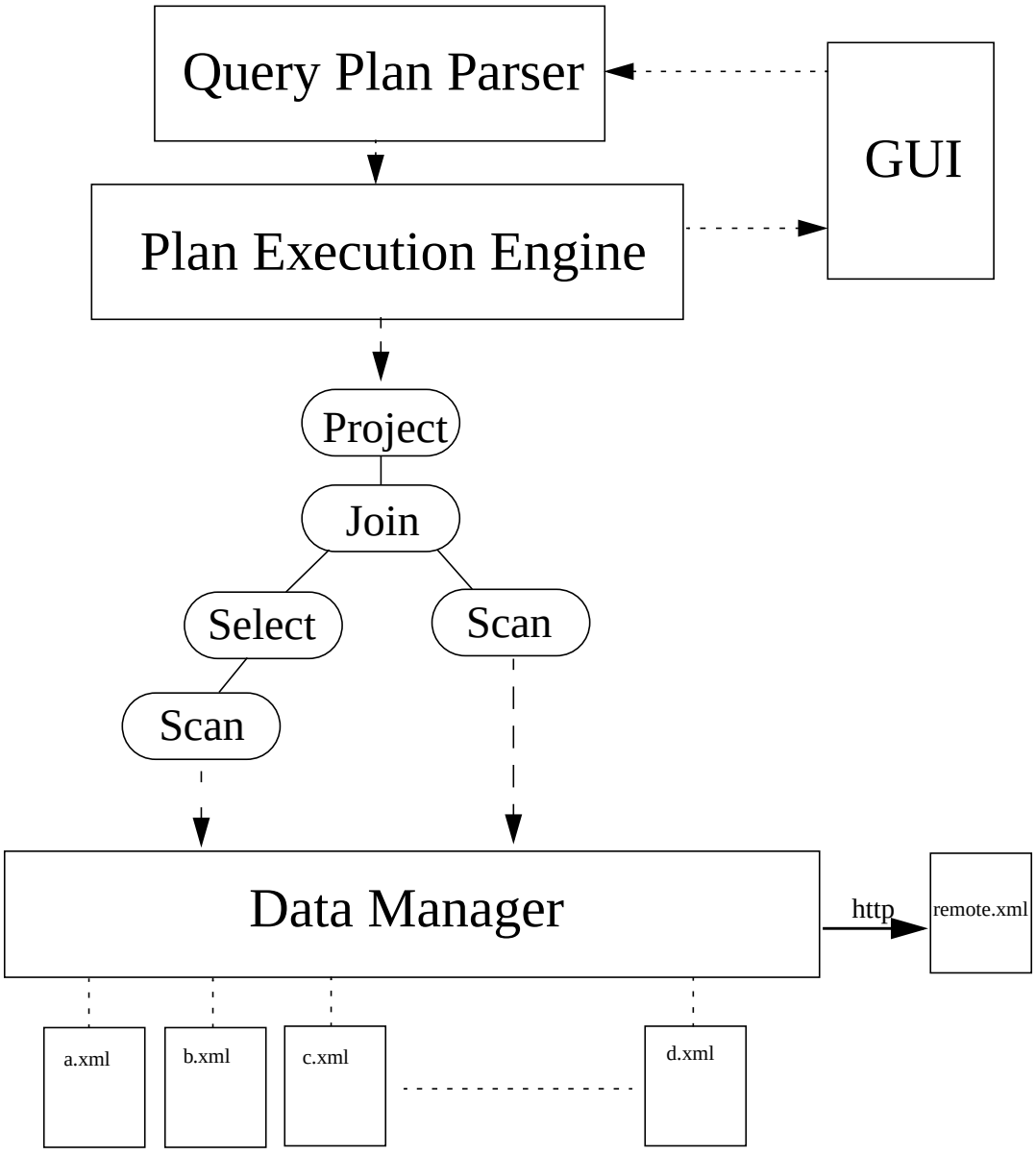

Figure 1. *System architecture*

the threads by calling the start method in each run-time operator object (a run-time operator object contains a thread object, as well as an output buffer). Each operator is responsible for starting its source operators, so at the top level we only need to start the root Construct operator.

Figure 1 Shows our overall system design. The graphical user interface allows the user to specify either the file containing the query plan, or a URL of an XML document to browse. If the GUI is being used for execution of a query plan, the plan file is parsed by the query plan language parser and a list of operator nodes is generated. Some rudimentary typechecking is done in our parser and plan execution engine, but it is assumed that bulk of typechecking would be enforced by some query language parser and plan generator that would fit above our system. The resulting XML file and query statistics are passed back to the GUI for display when a query is executed. These statistics include the plan parse time, the data load time, and the query execution time.

3. Query Plan Syntax:

The syntax recognized by the query plan parser is described below. Most operators have alternative forms, usually allowing an optional tag name to surround the operators output with. A more precise definition of the predicate and path expression syntax is given in later sections devoted to those topics.

```
Scan from “filename.xml” to A ( pathexpr )
Select from A to B condition { predicate }
Join from ( A, B ) to C condition { predicate } [ TAG <atagname> ]
Union from ( A, B ) to C
Construct from A to B [ TAG < atagname > ]
Construct from A to “filename.xml” [ TAG < atagname > ]
Project from A to B [ path.one, path.two,.., path.n ] [ TAG <tag> ]
Rename from A to B ( Tag <oldtag> TO Tag <newtag> )
```

Single capital letters in the above syntax represent buffer identifiers. Identifiers are shown here as single capital letters for clarity, but have the same syntax as an identifier in a language such as C++. The

buffer identifiers are used to match the source and destination buffers for of two linked operators.

4. Query Plan Parser:

We have implemented a parser to process plans written in the language described above. The goal symbol of the parser is an operator list and the output of the parser is a hash table of operator nodes hashed on the destination buffer id of the operator. This hash table is used to instantiate the plan by matching matching the source buffer of a given operator to the destination buffer of some other operator. The parser uses a scanner produced by Jlex, a flex like scanner generator for Java. The parser was created by Java Cup, a yacc like parser generator for Java.

5. Query Execution Engine:

The execution engine takes a query plan input file, and calls the parser to generate the hash table of operators. The top operator will always be a construct operator with an output file as its destination. This output file is the result of the XML query, and will contain an inline DTD and the data satisfying the query. The top operator is used with the recursive method `operator Instatiate( opInfoNode )`. This method called to recursively instantiate the plan. Instantiate takes an `opInfoNode` and looks up the `opInfoNode`s for each of its sources. A recursive call to `Instatiate()` is made on each of these, and the operators returned from each call to `Instatiate()` are given to the constructor of the operator making the recursive call. Since a scan operator receives data from no other operator as input, a scan operator is simply created and returned. The leafs of the operator tree will always be scan operators, and the top node is always a construct operator.

6. Operator Implementation:

All operators are derived from the base class Operator, which provides the following methods.

```
Element getNext()
```

```
void    start()
void    stop()
```

Each operator is implemented as a single thread that calls `getNext()` on its source operators and processes the retrieved Elements as described in the following sub-sections. Each operator object contains a synchronized queue object which serves as its output buffer. We chose to include the buffer in the run-time operator object, because each operator has exactly one output buffer. During the execution phase each thread reads XML nodes from the output buffers of its source operators, and writes XML nodes to its own output buffer. Synchronization between the operator threads is achieved through the use of these buffers. The operator buffers are implemented as synchronized queues, so if a queue becomes full - the producing (writing) operator blocks, and if it becomes empty - the consumer (reading) operator blocks.

6.1 Scan

The scan operator takes a filename and a path expression as input. This filename and path expression is used to initialize a new scan from the Data Manager. The Data Manager returns a list of all nodes in the given document that are reachable by following the given path expression. For example, opening a scan on the document bibliography.xml using the path expression ‘books.author’ would return pointers to all authors of books in the bibliography document. These nodes are placed into the output buffer of the scan operator and the operator above the scan retrieves ‘author’ elements by calling `getNext()` on the scan operator.

6.2 Select

The select operator takes elements from some lower operator, applies a predicate, and places those nodes satisfying the predicate into its output buffer. Details of supported predicates and predi-

cate evaluation are given in section Section 7.

6.3 Join

The join operator takes elements from two lower operators, (a left-outer op, and a right-inner op) and applies a join predicate. Join predicates are limited to equality comparison between two path expressions. The path expression may lead to either simple or complex elements. Equality comparison of simple and complex elements is discussed in Section 7. A new node is constructed for each pair that satisfies the join predicate. This new node is an element with either a user provided, or default tag name as the parent node and the pair of elements satisfying the join as children nodes. These newly constructed nodes are placed into the output buffer of the join operator, and are retrieved from higher operators calling the join operators `getNext()` method.

6.4 Union

The union operator takes two input sources and outputs all the nodes from these inputs into its output buffer. It reads the first source followed by then the second source, so the result is that it appends the nodes from the second input after the nodes from the first input. The source nodes do not have to be the same type (as opposed to SQL) because in XML sibling nodes can be of different types.

6.5 Project

The project operator is given a list of path expressions when it is constructed. It takes input elements from a lower operator, and ‘projects out’ subtrees that are reachable from all paths in the path list. These projected sub trees can be optionally constructed into a new tree with a user provided tagname. If no user provided tagname is given, the projected subtrees are placed directly into the output buffer. Therefore the output from the project operator can be either homogeneous or heteroge-

neous elements.

6.6 Construct

The construct operator takes elements from a lower node and optionally places these nodes under a new user provided tagname. This tagname is provided as an optional tag expression in the query plan. For instance, '`TAG <myresultname>`' would place the input to the construct operator in a new element called 'mytagname'. The following XML-QL query should make clear how this operator is used (in conjunction with a scan operator on bib.books.authors)

```
WHERE <bib>
            <books> $B </>
                  <authors> $A </>
            </books
       </bib>
CONSTRUCT   <mytagname> $A </mytagname>
```

The destination for outputting elements from the construct operator is either a new valid XML file, or an in memory buffer for some higher operator. This is accomplished simply by specifying in the query plan either "filename.xml", or a buffer identifier (H) after the keyword TO in the plan. The following two lines from an ascii plan illustrate this.

```
Construct from A to B TAG < mynewtagname >
Construct from A to "newxmldoc.xml" TAG < mynewtagname >
```

Once again, the tagname is optional. If a filename is given, the XML data is written to the specified filename along with an inline DTD for the new document. The details of DTD creation and unification is discussed in Section 8.

6.7 Rename operator

The rename operator is used to change tag names of nodes. For each node read from the input source, it walks the tree of each node and changes the names of those elements with specified old tag

name to the specified new tag name. To avoid affecting other operators by changing tag names, the Rename operator clones each subtree it reads from the input (note: other operators that change the structure of a subtree also duplicate nodes to avoid altering the original data source. Thus the original data source is in effect read only from the perspective of the query engine).

7. Predicate Evaluation:

Predicate evaluation is accomplished through a separate class. This class provides the following public methods for the evaluation of predicates.

```
boolean eval( Element tree, Predicate pred );
boolean eval( Element tree1, Element tree2, Predicate pred);
```

The first function evaluates predicates over a single tree, and is used by the selection operator. The second function is used by the join operator, and evaluates predicates involving path expressions over two trees.

7.1 Join predicates

Predicates used for join operators are restricted to the comparison of two path expressions with the equality operator, ==, and to boolean literal values (which result in either a cross product, or empty join). Complex element comparison is allowed. The comparison of complex elements is discussed in Section 7.4.

7.2 Selection predicates

Selection predicates may involve the following expression types.

- boolean literals
- EXISTS some.path.expr
- binary comparison operators
- (expr) AND (expr)
- (expr) OR (expr)

- NOT (expr)
- [ANY|ALL] (pathexpr binaryCompOp [path | const])

7.3 Type Coercion

We implement type coercion similar to that used in the Lore system (1). This is accomplished by attempting to convert string values to doubles before the comparison is made. If both operands are able to be converted, then the comparison is made between double value. If the conversion fails for either operand, the comparison is made as a lexical comparison of two strings. In this scheme, the expressions below evaluate as indicated:

Table 1. *Evaluation of expressions with and without type coercion*

Expr	Result With Coercion	Result Without Coercion
1.000 == 1.0	true	false
1.00 < 1.0000	false	true
1.2220 > 1.222	false	true

7.4 Complex element comparison

In both join and selection predicates with the equality operator, XML-QE allows comparison between complex elements. For example, the predicate `(undergrad.name == grad.name)` applied to the following two XML fragments evaluates to `true`.

```
<undergrad>
      <name>
            <first>Fred</first>
            <last>Jones</last>
      <name>
</undergrad>
      :
      :
<grad>
      <name>
            <first>Fred</first>
            <last>Jones</last>
      <name>
```

```
</grad>
```

In the case of complex element comparison, equality means that the elements being compared have the exact same structure, and have the exact same values for all leaf nodes. Type coercion is NOT applied to the leaf nodes, so two leaf nodes of a complex elements with the values of 1.00 and 1.000 respectively will cause the complex element comparison to fail.

8. Output DTD construction

The output of every query is a valid XML document with an inline DTD to which it conforms. Therefore, an output from a query can be used as input for future queries.

The DTD for the output is created at the end of the query process, from the last construct operator, which is the root of the query tree. Since the DTD defines element classes, our algorithm also collects information for each element class (elements which have the same tag name). The algorithm starts by traversing the document tree from the root, and for each element class it collects the set of tag names (classes) of all the immediate children of nodes from that class. For example, suppose we have two nodes with tag name A, the first has children X,Y,Z and the second has children W,Y - then the set of possible child nodes of A is {W,X,Y,Z}. After collecting this information for each element class, we write an element definition line in the DTD for each class. The definitions we create are the least restrictive given the collected information - each definition allows any subset of the children in any order, and also allows for any number of repetitions. We achieve that by using a definition of the form (C1 | C2 | .... | Cn)*, where Ci are the possible children of the element class. For example, the definition we would use for the previous example would be (W | X | Y | Z)*.

This method allows us to create a DTD for any result. It also saves us handling of incompatibilities between the DTDs of the original (input) documents, since we do not directly use the original DTDs. Another approach we attempted was to use the original DTDs and handle any incompatibility

by renaming elements. We succeeded in this approach, but abandoned it because we think this approach has significant disadvantages. First of all, choosing new names is done automatically, so the names lose their meaning to a human. In addition, renaming a node may result in either having to rename the parent (and create a new definition for it), or changing the parent's class definition.

9. GUI - XML browser

We implemented a simple front end for the query engine. Figure 2 shows this front end. This GUI allows the user to enter either the name of the text file that contains the query plan, or a URL for an XML data source to simply browse an XML documents content.

Figure 2. *Snapshot of the query engine front end*

If the file name entered is a query plan, the query is executed and the XML document that is

the query result is displayed. If the filename entered is a URL for an XML document, the document is retrieved, parsed and displayed. XML files are displayed similar to the common interface for browsing directories. Each element can be clicked to open it and show its nested elements.

10. .Conclusions

XML-QE provides the basic functionality to execute queries over XML data sources and produce valid DTD conforming XML documents as query results. Our self defined query plan syntax language is flexible enough that whatever XML query language that evolves with be able to translate queries into query plans.

As with any project, we became aware during the implementation phases of many features that could be added to the query engine. Some of those features are:

- -regular path expression support
- -the plan syntax and predicate evaluator could be extended to allow predicates involving attribute

11. References

1) Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, Janet L. Wiener: The Lorel Query Language for Semistructured Data. Int. J. on Digital Libraries 1(1): 68-88 (1997)

2) XML-QL: A Query Language for XML Submission to the World Wide Web Consortium 19-August-1998
http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/

Appendix A Example Query Plans

```
##############################################################################
#
#  exists.plan
#  ------------
#
#  This plan demonstrates the EXISTS operator by selecting staff who have
#  offices.  From those staff, the firstname and office are projected and
#  re-constructed under the new tag <staffwithoffices>
#
##############################################################################


Scan from “file:/u/a/d/adiel/private/cs764/data/second.xml” to A ( cspeople.staff )
Select from A to B Condition { EXISTS office }
Project from B to C [ name.firstname, office ] TAG <staffwithoffices>
Construct from C to “staffwithoffice.xml”
```

```
##############################################################################
#
#  any-all.plan
#  ------------
#
#  This plan demonstrates the use of the ANY and ALL operators.  It scans the
#  cspeople node (root) from the XML doc, and then selects the cs department
#  iff ANY undergrad has a gpa > 2.5.  For all csdepts matching this criteria,
#  the gradstudent subtrees are projected out.
#
#  Change ANY to ALL and modify the gpa criteria
#
##############################################################################

Scan from “file:/u/a/d/adiel/private/cs764/data/second.xml” to A ( cspeople )
Select from A to B Condition { ALL (undergradstudent.gpa > “2.5”) }
Project from B to C [ gradstudent ]
Construct from C to “allgrads.xml”
```

```
#############################################################################################
#
#  rename.plan
#  -----------
#
#  This query demonstrates the rename operator, and complex predicate evaluation using
#  The NOT, and AND operators.
#
#  Remove the first NOT and get all gradstudents that are not Harit
#
#
#############################################################################################

Scan from “file:/u/a/d/adiel/private/cs764/data/second.xml” to A ( cspeople.gradstudent )
Select from A to B Condition {NOT( NOT( firstname == “Harit” AND lastname == “Modi”)) }
Rename from B to C ( TAG <firstname> to TAG <newfirstname> )
Construct from C to “rename.xml” TAG <harits>
```

```
#############################################################################################
#
#  dualmajors.plan
#  ---------------
#
#  This plan return all gradstudents who are grads in the both the geology and
#  cs departments.  It joins on the condition ( name == name ) where name is a complex
#  element type.  The result is nested in a new tag “<dualmajors>” and then projects
#  the name subnode from the csgrads subelements of the cs gradstudent subnodes
#
#  It demonstrates a join across two XML sources at two different sites.
#  The query is executed by first retrieving the data source from the remote site
#
#  1) Cross site joins
#  2) Complex element equality comparisson
#  3) Projecting subtrees from a newly constructed join element
#  4) When run multiple times, shows the difference in Load phase times associated
#     with fetching the remote document
#
#############################################################################################


Scan from “http://www.geology.wisc.edu/~bjackson/geology.xml” to A  (geopeople.geograds)
Scan from “http://www.cs.wisc.edu/~brucej/xml/cspeople.xml” to B  (cspeople.gradstudent)
Join from (A,B) to C condition { name == name } TAG <dualmajors>
Project from C to D [ dualmajors.gradstudent ]
Construct From D to “/u/a/d/adiel/private/cs764/geo_cs.xml” TAG <geocsgrads>
```

```
##############################################################################################
#
# Restructure the original cs-people file so that each group of people
# are under a new group node. For example, the plan creates a new “grads”
# and puts al the grad students nodes under it.
# The same for staff and undergrads.
#
##############################################################################################

Scan from “data/second.xml” to A1 (cspeople.undergradstudent)
Construct from A1 to A2 Tag <undergrads>
Scan from “data/second.xml” to B1 (cspeople.gradstudent)
Construct from B1 to B2 Tag <grads>
Scan from “data/second.xml” to C1 (cspeople.staff)
Construct from C1 to C2 Tag <staffpeople>
Union from (A2, B2) to D1
Union from (C2, D1) to D2
Construct from D2 to “restructure.xml” Tag <cspeople>
```

```
##############################################################################################
#
#  smartundergrads.plan
#  --------------------
#
#  This query selects all cs undergrads that have a gpa greater than the graduate
#  student with firstname “Adiel”.  It demonstrates a join within a single document
#  with the join condition being “true” (cross product) so that each undergrad
#  is matched with a single grad (Adiel).  The undergrad students that satisfy the
#  condition are projected out and re-constructed under the new tag <smartundergrads>
#
##############################################################################################


Scan from “data/smart.xml” to A ( cspeople.gradstudent )
Select from A to B Condition { name.firstname == “Adiel” }
Scan from “/data/smart.xml” to C (cspeople.undergradstudent )
Join from (B, C) to D Condition { true }
Select from D to E condition { undergradstudent.gpa > gradstudent.gpa }
Project from E to F [ undergradstudent ]
Construct From F to “smartguys.xml” TAG <smartundergrads>
```

```
##############################################################################################
#
#  typecoerce.plan
#  ---------------
#
#  This plan scans all gradstudents from the cspeople database and selects those
#  gradstudents that have a gpa == 3.9.  The purpose of this query is to demonstrate
#  the Lorel style “type coercion” used when predicates are evaluated
#
#
##############################################################################################


Scan from “data/second.xml” to A ( cspeople.gradstudent )
Select from A to B Condition { gpa == “3.9” }
Construct from B to “/u/a/d/adiel/private/cs764/typecoerce.xml” TAG <gpas3point9>
```