DVQ: A DTD-DRIVEN VISUAL QUERY INTERFACE FOR XML DATABASE SYSTEMS

Long Zhang, Shihui Zheng, Aoying Zhou
Computer Science & Engineering Department
Fudan University, Shanghai, China
{lzhang0, shzheng0, ayzhou}@fudan.edu.cn

Hongjun Lu
Computer Science Department
Hong Kong University of Science & Technology, Hong Kong, China
luhj@cs.ust.hk

Abstract XML has been recognized as a promising language for data exchange over the Internet. A number of query languages have been proposed for querying XML data. Most of those languages are path-expression based. One difficulty in forming path-expression based queries is that users have to know the structure of XML data against which the queries are issued. In this paper, we describe a DTD-driven visual query interface for XML database systems. With such interface, a user can easily form path-expression based queries by clicking elements in the DTD tree displayed on the screen and supplying conditions if necessary. The interface and the query generation process are described in detail.

Keywords: XML, DTD, visual query, path expressions

1. INTRODUCTION

The Extensible Markup Language (XML) is an emerging standard for data representation and exchange on the Internet. There is a consensus among the researchers that XML is as an easy-to-write, easy-to-parse language to exchange data in a variety of applications on the Internet. Currently, most business data is stored in relational or object-relational systems. It will continue to be so since the matured relational technology provides excellent queriability, scalability and availability. It becomes a natural choice to store XML documents in relational systems to make use of such advantages and hopefully ease the difficulties in integrating XML documents into other applications. Large amount of research work has been conducted to study issues related to storing XML documents using relational database management systems (Shanmugasundaram et al., 1999; Deutsch et al., 1999; Shanmugasundaram et al., 2000; Manolescu et al., 2001). Storing an XML document into a relational database system requires to map the document into relational tables. In other words, a relational schema must be defined. Yoshikawa and Amagasa categorize such XML-Relation mapping into two categories: *structure-mapping-based* approach and *model-mapping-based* approach (Yoshikawa and Amagasa, 2001). The structure-mapping-based approach generates a relational schema from an XML document based on its logical structure described by its DTD (Document Type Descriptor). The model-mapping-based approach generates a relational schema for an XML document based on its representation model (e.g. trees) without understanding its logical structure.

To query XML data, several XML query languages were proposed including LOREL (Abiteboul et al., 1997), XML-QL (Deutsch et al., 1998), XPath (Clark and DeRose, 1999) and XQuery (Chamberlin et al., 2001). Although those query languages differ from each other in ways of expressing queries, one of the common features of XML queries is that they are mainly path expression based, which is rather different from set-oriented relational query languages, such as SQL.

Compared to querying relational database using SQL, querying XML data is more difficult from the users' viewpoint since users need to know the structure of the data. When XML data is stored in relational systems, this becomes much more difficult, especially when structure-mapping based approach is used for schema mapping as the original structure may not be well reflected by the relational schema. While storage of XML and related query processing techniques received well attention, the efforts in providing users with facilities to ease the difficulty of writing path-expression based XML query are very limited. The DataGuide (Goldman and Widom, 1997; Goldman and Widom, 1998) interface in Lore system can generate simple LOREL (Abiteboul et al., 1997) queries by exploring the DataGuide of semi-structured data. But it can only specify simple path expressions, and set conjunctive conditions on only a unique path. Both the BBQ (Munro et al., 2000) visual interface of MIX and EquiX (Cohen et al., 1999) interface display the DTD structure of XML data to facilitate user to browse and formulate query. But they cannot construct arbitrary regular path expressions and complex constraint conditions too.

In this paper we present DVQ, a DTD-driven visual query interface for an XML-Relational database system VXMLR (Zhou et al., 2001). VXMLR adopts a structure-mapping based approach to map XML data into relational tables managed by a commercial RDBMS. Its query interface, DVQ, displays the structure of the stored XML data on the screen. By simple clicking the nodes in the DTD structure and entering related conditions, a user can easily form path expressions. DVQ then generates the path expression based queries automatically. The query results are represented by XSLT and delivered to users through DVQ. With DVQ, VXMLR users navigate the nested structure of the stored XML data, form query and browse the result documents through DVQ seamlessly. The unique features of DVQ include the following.

- It provides users with a graphical interface so that complex queries can be formed by users' simple GUI actions. Not only XML experts, but also ordinary users can specify query without the knowledge of the XML query languages.
- It is powerful enough to specify regular path expressions with wildcards and any complicate conditions with conjunction, disjunction and negation. At the same time, the WYSWYG (What You See is What You Get) feature of DVQ makes query formulation rather straightforward.
- In addition to enter queries, DVQ also provides facilities for users to navigate through the XML structure and to view the intermediate results of the major steps in query processing.
- It is a separate module running at the client side and driven by DTD of original XML data. That is, it is independent on the underlying XML-relational mapping schema. So, it is a portable module that can be used in any XML database system.

The remainder of this paper is organized as follows. In Section 2, we introduce some background information. In Section 3, we describe the architecture of DVQ. Section 4 describes how DVQ forms path expressions with set constraint conditions using examples. In Section 5, we present DVQ's function of monitoring XML query execution and displaying the query results. Finally, conclusions are presented in Section 6.

2. BACKGROUND INFORMATION

In this section, we introduce some background information about XML data and XML queries.

2.1. A DTD Sample

Unlike HTML documents where their structures are usually unknown, the data type definition (DTD) of an XML document describes the nested relationships between data elements in the document. A DTD example is shown in Figure 1, and we will use it as running example in the following discussion.

The DTD in Figure 1 describes XML documents about the information of laboratories. There are two basic constructs in XML documents, element and attributes, indicated by !ELEMENT and !ATTLIST, respectively. Elements in XML documents can be nested. Each document has a unique root element, which is labinformation in our example. It contains three sub-elements, labname, project, and member. Character * after project indicates that a laboratory element instance can have zero or more projects. + indicates that a laboratory element instance may have one or more members. Each member element has a unique ID attribute, and sub-elements name, email, URL, publication and project. The sign ? after URL indicates that this element is optional, that is, a member instance may not have a URL. The contents of an element is denoted by #PCDATA. The other portion of the DTD can be interpreted in a similar way. Note that member and project are defined recursively. That is, a member element can have project as sub-element, and vice versa.

```
<!ELEMENT labinformation (labname, project*,member+)>
<!ELEMENT member (name,email,URL?, publication*,project*)>
<!ELEMENT project (projtitle,introduction?,member*)>
<!ATTLIST member ID ID #IMPLIED>
<!ELEMENT publication (title,author*,year)>
<!ELEMENT labname (#PCDATA)>
<!ELEMENT projtitle (#PCDATA)>
<!ELEMENT name (lastname?,firstname)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT URL (#PCDATA)>
<!ELEMENT introduction (#PCDATA)>
<!ELEMENT author (name)>
<!ELEMENT year (#PCDATA)>
```

Figure 1. A Sample DTD.

2.2. Path Expressions

We can see from the DTD example that data elements in an XML document are nested to form a tree structure (a graph with IDREF). Figure 2 depicts an example document and its graphical representation, data graph. With such a structure, we can represent an element by the path from the root of the tree to the element. If we adopt the dot notation to denote the parent-child relationship, the path for a members name can be expressed as project.member.name, for example. In general, the path expressions can be more complex. If r, r_1, and r_2 are elements or attributes, a path expression has the form the root of
$r = (r)* \mid (r)+ \mid (r)? \mid r_1.r_2 \mid (r_1|r_2) \mid \# \mid name$

Where *, +, ? mean 0 or more, 1 or more, and 0 or 1, occurrences, respectively. Concatenation $r_1.r_2$ is used to form a path from r_1 to r_2.

Alternation " | " stands for disjunction. Wildcard "#" denotes arbitrary occurrences of any regular expressions. We distinguish two types of path expressions: simple path expression (SPE) and regular path expression (RPE). Simple path expressions are path expressions that consist of only element or attribute names such as `labinformation.project.member.name`. Regular path expressions are path expressions that contain regular operators. For example, `#.(project.member)*.name` is a RPE.

```
<project>
  <projname>XML</projname>
  <member ID="&3">
    <name>Zhou Aoying</name>
    <email>ayzhou@fudan.edu.cn</email>
    <publication author="Zhou">
      <title>A Algorithm ...</title>
      <year>Journal of ...</year>
    </publication>
    ...
  </member>
  <member ID="&24">
    <name>Zengping Tian</name>
    <project>
      <projname>clustering</projname>
    </project>
  </member>
</project>
```

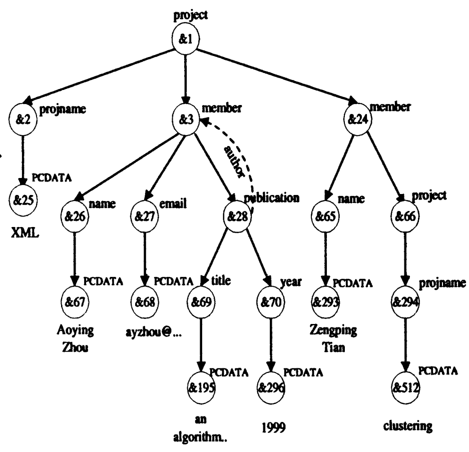


Figure 2. An XML document and its data graph.

2.3. VXMLR: A Visual XML-Relational Database System

The detailed description of VXMLR is out of the scope of this paper. To illustrate the context where DVQ works, we depict the architecture of VXMLR as shown in Figure 3. In VXMLR, XML documents are stored in a relational database system, managed by a relational DBMS. An input XML document is first parsed into a DOM tree. At the same time, the DTD for the document is extracted. The document tree is then mapped to relational tables and stored in the database. User issue queries through DVQ, the visual interface. Those queries are then transformed into SQL statements ready to submit to the underlying relational DBMS. To generate efficient SQL statements from path expression based queries, VXMLR maintains some statistics of data and a path directory. Both the statistics and the path directory are used in the query rewriting process to reduce the number of SQL statements and simplify join conditions. The query results returned from DBMS are constructed and expressed using XSL, which are then delivered to the user through DVQ.

3. DVQ: THE VISUAL QUERY INTERFACE

VXMLR system was implemented on the top of Microsoft SQL Server. The server side program is implemented in C++. The client side program, i.e., the visual query interface, which runs on a browser, is written in Java. Figure 4 shows the reference architecture of DVQ.

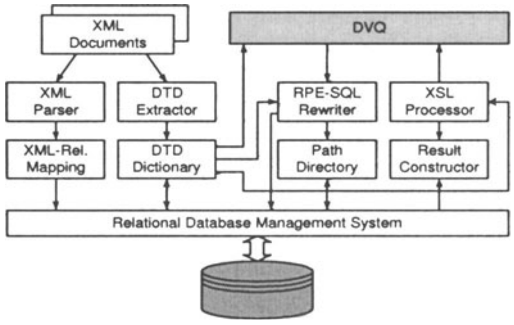


Figure 3. Reference architecture of VXMLR.

The portion upper the dashed line is the client side of VXMLR, while the below portion is the server side. The system works as following: at first, once a DTD descriptor is selected by user, the *DTD Sender* transfers the encoded DTD information to DVQ by way of *CGI*. Next, the *DTD Receiver* at the client side receives the decoded DTD and the *DTD Tree Generator* parses it into hierarchical structure, which then is displayed in *Query Interface*. Once user has constructed the query and clicked the *Submit* button, The *Query Generator* module generates path expression query automatically. The *Query Sender* module then sends the generated query to server. At the server side, the *Query Receiver* decodes the query and submits it to the *Query Engine* of VXMLR. The *Query Engine* translates the query into efficient SQL statements, which is then executed by the underlying RDBMS. VXMLR *Query Engine* constructs the result document using the result relational tables returned by RDBMS. Finally, the result document is represented by the *XSL Processor* and transferred to the browser for displaying.

Figure 5 is the screen dump (main window) of DVQ interface. It consists of two panels and some buttons. The left panel displays the DTD structure of XML data to be queried. What shown is the sample DTD in Figure 1 as a hierarchical directory, where elements and attributes are represented by their names, preceded with a diamond or a dot, respectively. Sub-elements or attributes are

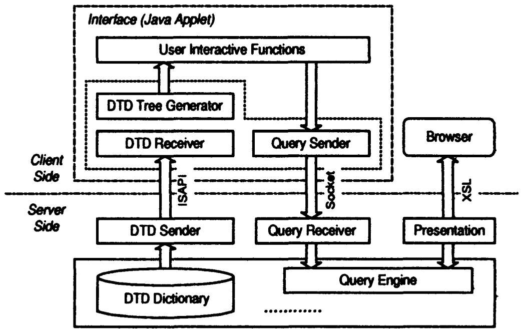
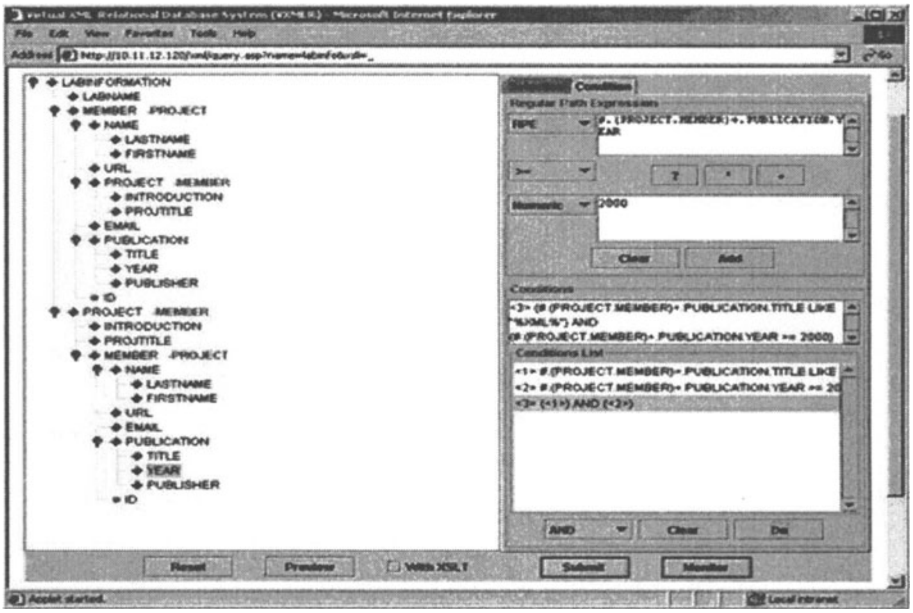


Figure 4. The reference architecture of DVQ.

expressed as the children of their parent node. User can click the icon before an element to explore or collapse its subtree. Recall that `project` and `member` are defined recursively, which is represented by "`project-member`" and "`member-project`" in the DTD panel. The DTD panel provides an easy way for user to navigate the nested structure of the stored XML data conforming to the DTD. To the right of the DTD panel is two tabbed panels, *Selection*



```
#.(PROJECT.MEMBER)+.PUBLICATION.YEAR
>=
2000
<3> (#.(PROJECT.MEMBER)+.PUBLICATION.TITLE LIKE
"%XML%") AND
(#.(PROJECT.MEMBER)+.PUBLICATION.YEAR >= 2000)
<1> #.(PROJECT.MEMBER)+.PUBLICATION.TITLE LIKE
<2> #.(PROJECT.MEMBER)+.PUBLICATION.YEAR >= 20
<3> (<1>) AND (<2>)
```

Figure 5. The DVQ interface.

and *Condition*. The *Selection* panel is used to form the target items (i.e. a list of path expression) to be retrieved. The *Condition* panel is for user entering the conditions that the retrieved items must satisfy. The basic components of both panels are path expression.

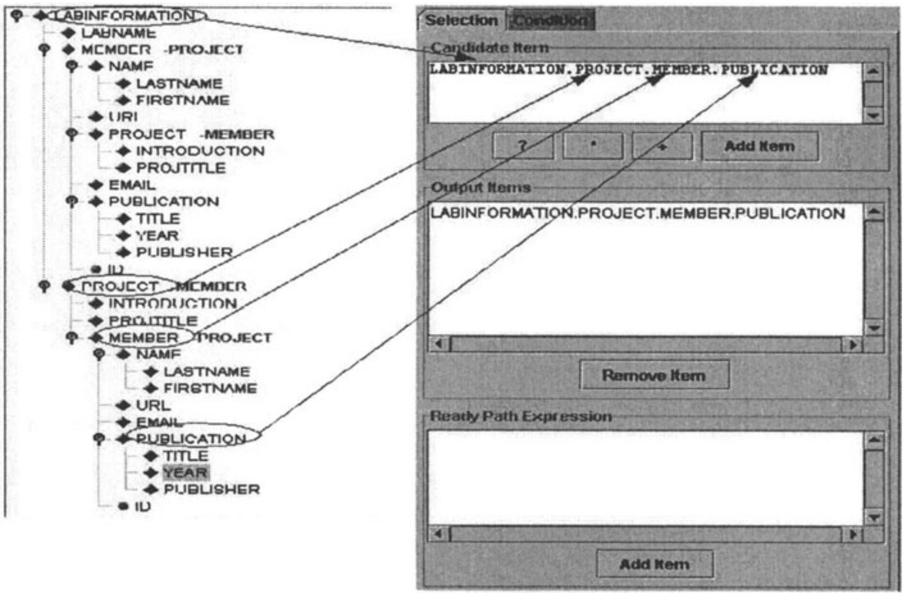
At the bottom of the main window are four buttons and a check box. The leftmost is the *Reset* button, which lets user to clear up the constructed items in the *Selection* and *Condition* panels. The *Preview* button brings up a dialog that displays the generated query for user to preview. To the right of the *Preview* button is a XSLT check box, which indicates the query result will be returned as plain XML file or XML page expressed by XSL. To the right of the XSLT check box is the *Submit* button, which triggers the query to be submitted to the VXMLR server for executing. When the right most *Monitor* button is pushed, an XML page will displayed that shows the elapsed time and query statements in each step of query processing.

4. VISUAL QUERY FORMATION

A user query session in VXMLR consists of four steps: First, browses the DTD structure and forms the output items by clicking nodes in the DTD structure and set constraints through the *Condition* panel. Second, after a user completes the specifications of the output items and constraints, a query in the form of path expressions is generated. Third, the generated query is submitted to the VXMLR server for executing. Finally, the result is returned from VXMLR server to the interface for user to browse.

4.1. Form Path Expressions

We first introduce the *Selection* panel and then show how to formulate path expressions in the select part of a query. Figure 6 shows the DTD panel and

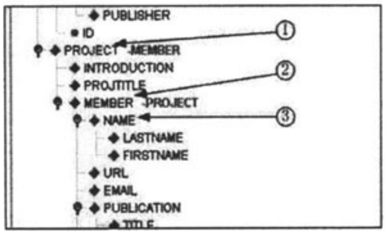


LABINFORMATION.PROJECT.MEMBER.PUBLICATION

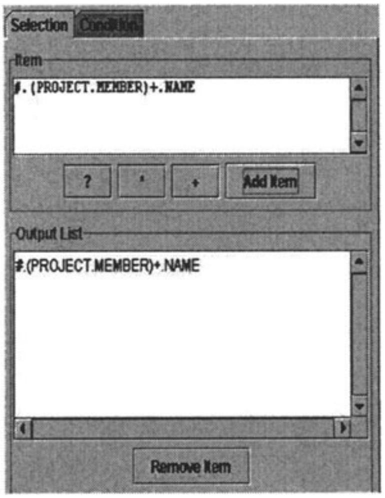
LABINFORMATION.PROJECT.MEMBER.PUBLICATION

Figure 6. Specifying simple path expression by SELECT panel.

Selection panel. The *Selection* panel consists of three text boxes, *Candidate Item*, *Output Items*, and *Ready Path Expression*, and some buttons. The first text box on the top of *Selection* panel, *Candidate Item*, is used for showing the path expression being constructed. There are four buttons under the *Candidate Item* text box. The first three buttons are regular operator buttons labelled with "?", "*", and "+" respectively, which trigger those regular operators been added to the current path expression shown in the select text box. To the right of the regular operator buttons is the *Add Item* button, which brings up the path expression in the *Candidate Item* text box added into the *Output Items* text box under those buttons.



(1)



#.(PROJECT.MEMBER)+.NAME

#.(PROJECT.MEMBER)+.NAME

(2)

Figure 7. Example for specifying regular path expressions by DTD panel (1) and SELECTION panel (2).

Now, we show by example how to form simple path expressions. A simple path expression is constructed in a natural way by clicking the nodes in the DTD panel following top-down links. Clicking a node and its children starting from the root triggers the name of the clicked nodes appended into the *Candidate Item* text box consecutively. Figure 6 shows that user clicks the nodes `labinformation`,`project`, `member` and `publication` consecutively. The generated path expression is SPE,

```
labinformation.project.member.publication
```

which is displayed in the *Condition Item* text box.

If the first clicked element is not the root element, or not the child of the element in the tail of the current path expression, then there is at least one element between the tail of the current path and the clicked element. A "#" operator is inserted before it automatically. For example, as shown in Figure 7(1) step one, user clicks the `project` element first followed by element `member` and `name` element. The first clicked element, `project`, is not the root. Those actions form a regular path expression `#.project.member.name`.

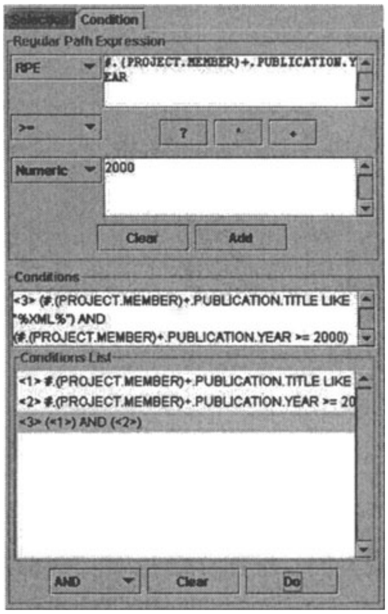
User can also insert the regular operators manually by first highlighting a segment in the path expression followed by clicking the corresponding operator button. The operator is then added on the path segment automatically. In Figure 7(2), highlighting path segment project.member followed by clicking "+" button forms a regular path expression `#.(project.member)+.name`.

Now user clicks the *Add Item* button. It triggers the current path expression in the *Candidate Item* text box appended to the *Output* text box. User can specifies target items and add them into the output items continuously.

Alternatively, user can also use the *Remove Item* button under the *Output* text box to remove one or more path expressions from the *Output Item* text box. At last, the path expressions in the *Output Item* text box will be used to construct the select part of the query.

4.2. Set Constraints on Output Items

Constraints can be placed on the paths in output items by *Condition* panel. Figure 8 shows the *Condition* panel, it consists of three text boxes, three pop-menus and several buttons. Same as the *Selection* panel, the *Path Expression* text box on the top of the *Condition* panel and the three regular operator buttons under it are for user to form path expressions, on which constraints are then replaced. The below part of the *Condition* panel is two text boxes labelled with "Conditions" and "Condition List", which let the user to combine multiple constraints to form complex conditions.



#.(PROJECT.MEMBER)+.PUBLICATION.YEAR

2000

<3> (#.(PROJECT.MEMBER)+.PUBLICATION.TITLE LIKE "%XML%") AND (#.(PROJECT.MEMBER)+.PUBLICATION.YEAR >= 2000)

<1> #.(PROJECT.MEMBER)+.PUBLICATION.TITLE LIKE
<2> #.(PROJECT.MEMBER)+.PUBLICATION.YEAR >= 20
<3> (<1>) AND (<2>)

Figure 8. Forming constraints with CONDITION panel.

To the left of the "?" operator button is the comparator popup menu, which consists of a set of basic comparators ("=", ">", ">=", "<", "<=", "*LIKE*", "*IN*"). Below the comparator popup menu is the *Type* popup menu, which indicates the type of the expression. The type of an expression can be a numeric, a string, a Boolean, or a path expression. In VXMLR, each element is stored as an attribute with string type in relational database. The comparator and type popup menus enable user to flexibly treat an element as other types. For example, user can specify a comparison between target element "`year`" with an integer "2000". In VXMLR, the comparison is implemented by casting the attribute in the relation as a numeric type. To the right of the *Type* popup menu is the *Expression* text box. User can use it to input an expression, which forms a condition comparing the path expression in the *Path Expression* text box with an expression, or input another path expression by clicking the nodes in the DTD panel, which form a condition that joins two paths. In Figure 8 , user first forms this path expression "`#.(project.member)+.publication.year`" in the *Path Expression* text box by clicking the corresponding elements and "+" button. Next, user selects the comparator ">=" and "NUMERIC" type, and inputs an integer values "2000" in the *Expression* text box. Finally, the following constraint condition

```
#.(project.member)+.publication.year>=2000
```

is formed. At this point, user can click the "ADD" button under the *Expression* text box to add the above condition into the *Condition List* text box or withdraw the formed condition by clicking the *Clear* button under the expression text box. In the *Condition List* text box in Figure 8, the condition has been added into the *Condition* text box.

DVQ is powerful enough to construct arbitrary complex conditions with "AND", "OR" and "NOT" predicates. Figure 8 also shows how to form complex conditions using the *Predicate* popup menu at the bottom of the *Condition* panel. User first highlights the two conditions have been added to the *Condition* text box, then selects the "AND" predicate from the *Predicate* popup menu. Those actions bring up the two conditions in the *Condition* text box are connected by "AND" predicate and the following conjunctive condition,

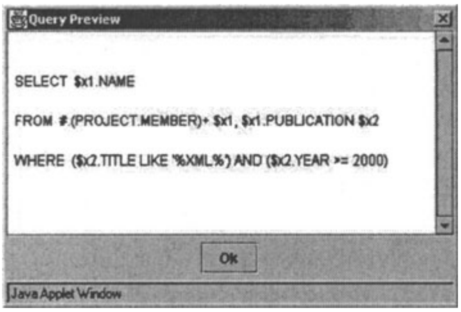
```
#.(project.member)+.publication.title LIKE '%XML%'
AND #.(project.member)+.publication.year>=2000
```

is formed. Continuing to add expressions and predicates, arbitrary nested complex condition composed of conjunction, disjunction and negation can be composed also. The generated conditions are simultaneously displayed in the *Conditions* text box above the *Condition List* text box. To the right of the predicate popup menu of the two buttons are labelled with "Clear" and "Add", which bring up the conditions in the *Condition* text box are cleared or added into the where part of the formed query.

4.3. Executing Query and Browsing Result

After a user specifies the path expression and the related conditions as shown in Figure 7 (2) and Figure 8, DVQ will form the query automatically. User can preview the generated query by clicking the *Preview* button in the right part of the main window.

Figure 9 shows the dialog triggered by the click action, which displays the generated query. Note that variables are added into the queries. DVQ converts the query into the forms of Figure 9 automatically. We just sketch out the main idea for conversion and do not go into the details for query conversion here: the query generator first finds all of the path prefixes



```
SELECT $x1.NAME

FROM #.(PROJECT.MEMBER)+ $x1, $x1.PUBLICATION $x2

WHERE ($x2.TITLE LIKE '%XML%') AND ($x2.YEAR >= 2000)
```

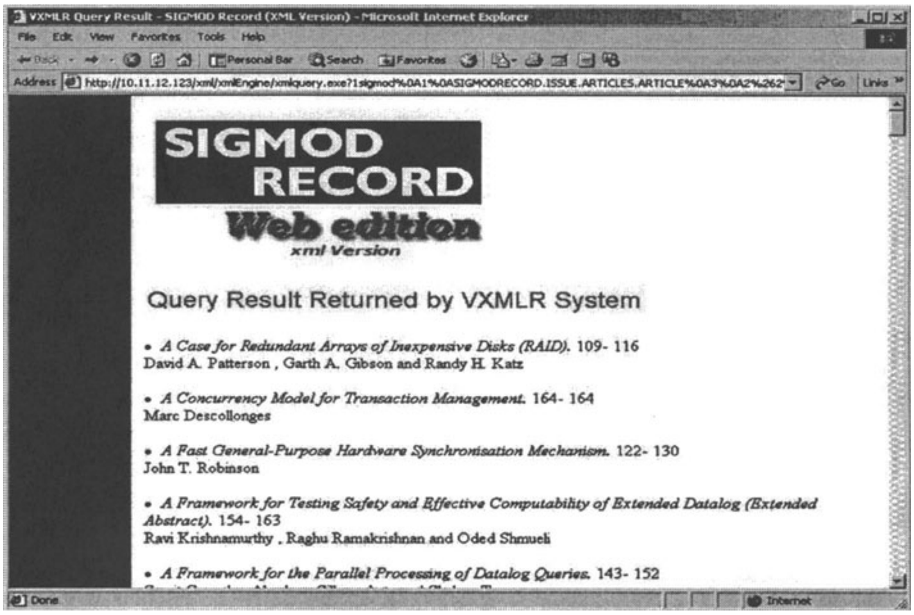
Figure 9. The preview dialog box displaying the generated query.

of those path expression appearing in the output items and constraint conditions, next, assigns each common path prefix a variable and place those common prefixes in the FROM clause. Then replaces all common prefixes with variable in all of those path expressions. Finally, states the constraints in the WHERE clause and puts the retrieved items in the SELECT clause. For our example, `#.(project.member)+` and `publication` are two common prefixes, which are assigned variables $x1 and $x2 respectively. The SELECT clause consists of the target item $x1.member.name and WHERE clause consists of "`$x2.title LIKE ' %XML%'and $x2.year>=2000`".

The generated query can be submitted to the VXMLR sever by click the *Submit* button. Then the query session enters into the third step: VXMLR server translates the query into SQL statements and submits it to RDBMS, then transforms the results into XML documents. If the XSLT check box is selected, then the XSLT sheet of the document is also returned. Finally, the result document is displayed in the browser and delivered to user. For example, the following query,

```
Select $x1
From sigmod.issue.articale.article $x1
Where $x1.initialpage>=100 and $x1.initialpage<=200
```

retrieves the articles with initial page above 100 in SIGMOD RECORD, and the result document represented by XSLT is shown in Figure 10.


VXMLR Query Result - SIGMOD Record (XML Version) - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://10.11.12.123/xml/xmlEngine/xmlquery.exe?1sigmod%0A1%0ASIGMODRECORD.ISSUE.ARTICLES.ARTICLE%0A3%0A2%262
Go Links
SIGMOD RECORD
Web edition
xml Version
Done
Internet

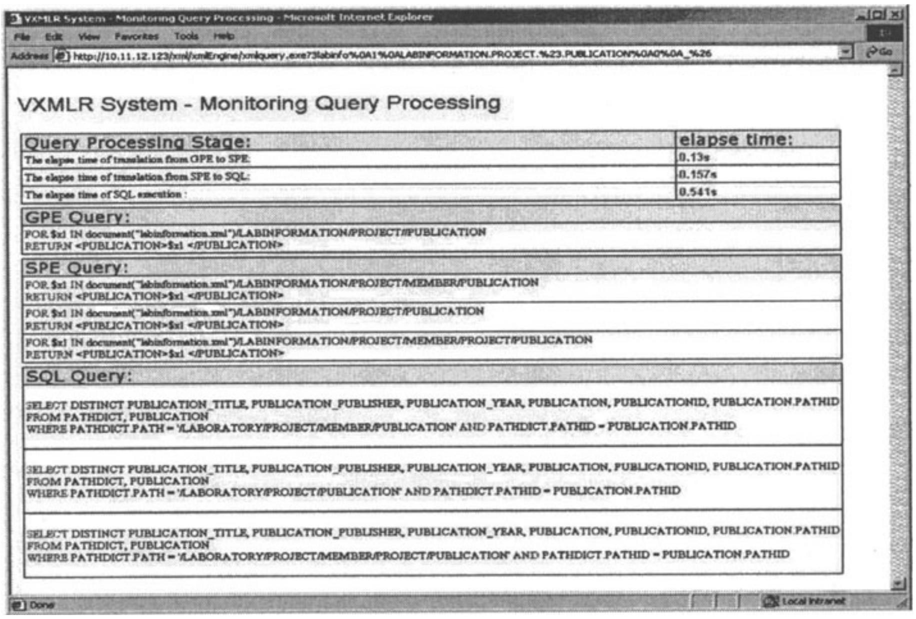
Query Result Returned by VXMLR System

- *A Case for Redundant Arrays of Inexpensive Disks (RAID).* 109- 116
David A. Patterson , Garth A. Gibson and Randy H. Katz
- *A Concurrency Model for Transaction Management.* 164- 164
Marc Descollonges
- *A Fast General-Purpose Hardware Synchronisation Mechanism.* 122- 130
John T. Robinson
- *A Framework for Testing Safety and Effective Computability of Extended Datalog (Extended Abstract).* 154- 163
Ravi Krishnamurthy , Raghu Ramakrishnan and Oded Shmueli
- *A Framework for the Parallel Processing of Datalog Queries.* 143- 152

Figure 10. The result document represented by XSL.

5. MONITORING FACILITY

Querying XML data stored in relational systems is a relatively complex process. The user formed path expression queries have to be translated into SQL statements first. When the user query is a regular expression query, it needs to be transformed to simple path expression queries first. With the monitoring facility of DVQ, user can view the results of those intermediate results, in addition to time used for each step.


VXMLR System - Monitoring Query Processing - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://10.11.12.123/xml/xmlEngine/xmlquery.exe?3labinfo%0A1%0ALABINFORMATION.PROJECT.%23.PUBLICATION%0A0%0A_%26
Go
Done
Local intranet

VXMLR System - Monitoring Query Processing

Query Processing Stage:	elapse time:
The elapse time of translation from GPE to SPE:	0.13s
The elapse time of translation from SPE to SQL:	0.157s
The elapse time of SQL execution :	0.541s

GPE Query:

```
FOR $x1 IN document("labinformation.xml")/LABINFORMATION/PROJECT//PUBLICATION
RETURN <PUBLICATION>$x1 </PUBLICATION>
```

SPE Query:

```
FOR $x1 IN document("labinformation.xml")/LABINFORMATION/PROJECT/MEMBER/PUBLICATION
RETURN <PUBLICATION>$x1 </PUBLICATION>

FOR $x1 IN document("labinformation.xml")/LABINFORMATION/PROJECT/PUBLICATION
RETURN <PUBLICATION>$x1 </PUBLICATION>

FOR $x1 IN document("labinformation.xml")/LABINFORMATION/PROJECT/MEMBER/PROJECT/PUBLICATION
RETURN <PUBLICATION>$x1 </PUBLICATION>
```

SQL Query:

```
SELECT DISTINCT PUBLICATION_TITLE, PUBLICATION_PUBLISHER, PUBLICATION_YEAR, PUBLICATION, PUBLICATIONID, PUBLICATION.PATHID
FROM PATHDICT, PUBLICATION
WHERE PATHDICT.PATH = '/LABORATORY/PROJECT/MEMBER/PUBLICATION' AND PATHDICT.PATHID = PUBLICATION.PATHID

SELECT DISTINCT PUBLICATION_TITLE, PUBLICATION_PUBLISHER, PUBLICATION_YEAR, PUBLICATION, PUBLICATIONID, PUBLICATION.PATHID
FROM PATHDICT, PUBLICATION
WHERE PATHDICT.PATH = '/LABORATORY/PROJECT/PUBLICATION' AND PATHDICT.PATHID = PUBLICATION.PATHID

SELECT DISTINCT PUBLICATION_TITLE, PUBLICATION_PUBLISHER, PUBLICATION_YEAR, PUBLICATION, PUBLICATIONID, PUBLICATION.PATHID
FROM PATHDICT, PUBLICATION
WHERE PATHDICT.PATH = '/LABORATORY/PROJECT/MEMBER/PROJECT/PUBLICATION' AND PATHDICT.PATHID = PUBLICATION.PATHID
```

Figure 11. Browing query translation

Figure 11 shows a screen dump of execution of a query that selects all of the publications reachable from the project node via zero or more edges. That is,

```
select labinformation.project.#.publication
```

It can be seen that DVQ displays the following information: the elapsed time of each main query processing stage and their results, the original regular path expression query, the corresponding simple path expression queries, and the SQL statements executed by the underlying DBMS.

For the example shown in Figure 11, the original RPE query is expanded to three simple path expressions.

```
select labinformation.project.member.publication
union select labinformation.project.publication
union select labinformation.project.member.project.publication.
```

A simple path expression can be straightforwardly translated into joins between parent element and child element in SQL statement. With the path directory that materializes the paths from the root to elements existing in the data, a simple path expression can be translated into a rather simple SQL query. For example, SPE query

```
select labinformation.project.member.publication
```

is translated into the following SQL statement.

```
select labinformation.project.member.publication
from PathDirectory, publication
where PathDirectory.path="abinformation.project.member.publication"
and PathDirectory.pathid=publication.pathid
```

6. CONCLUSIONS

In this paper, we presented DVQ, a DTD-driven visual query interface of a prototype XML databases system, VXMLR. DVQ provides users a graphical interface for navigating XML structure, forming complex path expression queries, submitting queries for execution, browsing query results and monitoring the main steps of a query session.

The current version of DVQ only displays one DTD, and user can only query single document. We are extending this version so that users can query across related documents with different DTDs.

REFERENCES

S. Abiteboul, D. Quass, J. Mchugh, and et al. The Lore Query Language for Semistructured Data, *International Journal on Digital Libraries, 1(1): 68-88, April 1997.*

J. Clark and S. DeRose. XML path language (XPath), *W3C Recommendation 16 Nov. 1999, http://www.w3.org/TR/xpath, 1999.*

D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and et. al. XQuery: A query Language for XML, Technical report, *World Wide Web Consortium, Feb. 2001. Available from http://www.w3.org/TR/xquery.*

Michael J. Carey, Laura M. Haas, Vivekananda Maganty, John H. Williams: PESTO : An Integrated Query/Browser for Object Databases, *Proceedings of the Twenty-Second International Conference on Very Large Data Bases, 1996, 203-214.*

S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, Y. Sagiv, A. Serebrenik. EquiX Easy Querying in XML Databases, *WebDB 1999: 43-48.*

A. Deutsch, M. Fernandez, D. Florescu, and et al. XML-QL: A Query Language for XML, *W3C Note, 1998. Available: http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/*

A. Deutsch, M. Fernandez, D. Suciu. Storing Semistructured Data with STORED, *Proceedings of the 28th SIGMOD International Conference on Management of Data, May 1999.*

Mary F. Fernandez, A. Morishima, D. Suciu. Efficient Evaluation of XML Middle-ware Queries, *Proceedings of the International Conference on Management of Data, Santa Barbara, California. Jun. 2001.*

R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases, *Proceedings of the 23rd International Conference on Very Large Data Bases, Athens, Greece, August 1997, 436-445.*

R. Goldman and J. Widom. Interactive Query and Search in Semistructured Databases, *International Workshop on the Web and Databases, Valencia, Spain, Mar. 1998.*

I. Manolescu, D. Florescu, D. Kossmann. Answering XML queries on heterogeneous data sources, *Proceedings of the International Conference on Very Large Data Bases, Roma, Italy, Sept. 2001.*

A. Marian, S. Abiteboul, G. Cobena, L. Mignet. Change-Centric Management of Versions in an XML Warehouse, *Proceedings of the 23rd International Conference on Very Large Data Bases, Roma, Italy, Sept. 2001.*

J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data, *SIGMOD Record, 26(3): 54-66, Sept. 1997.*

A. Mendelzon, G. Mihaila, T. Milo. Querying the World Wide Web, *Int. J. on Digital Libraries, 1(1): 54-67. 1997.*

K. Munroe, Y. Papakonstantinou. BBQ: A Visual Interface for Browsing and Querying XML, *Proceedings of the Fifth IFIP Working Conference on Visual Database Systems, 2000.*

B. Nguyen, S. Abiteboul, G. Cobena, M. Preda. Monitoring XML Data on the Web, *Proceedings of the International Conference on Management of Data, Santa Barbara, California. June 2001.*

J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL), Available from *http://www.w3.org/TandS/QL/QL98/pp/xql.html, Dec. 1998.*

J. Shanmugasundaram, J. Kiernan, E. J. Shekita, and et.al. Querying XML views of relational data, *Proceedings of the International Conference on Very Large Data Bases, Roma, Italy, Sept. 2001.*

J. Shanmugasundaram, E. J. Shekita, R. Barr, and et.al. Efficiently Publishing Relational Data as XML Documents, *Proceedings of the International Conference on Very Large Data Bases, Cairo, Egypt, September 2000, 65-76.*

J. Shanmugasundaram, K. Tufte, C. Zhang, and et. al. Relational Databases for Querying XML Documents: Limitations and Opportunities, *Proceedings of the International Conference on Very Large Data Bases Edinburgh, Scotland, 1999.*

M. YoshiKawa and T. Amagasa. XRel: A path-based approach to storage and retrieval of XML documents using relational databases, *ACM Transactions on Internet Technology, Volume 1, Number 1, 2001.*

P. Veltri, S. Cluet, D. Vodislav. Views in a large scale XML repository, *Proceedings of the International Conference on Very Large Data Bases, Roma, Italy, Sept. 2001.*

A. Zhou, H. Lu, S. Zheng. and et. al. VXMLR: A Visual XML-Relational Database System, *Proceedings of the International Conference on Very Large Data Bases, (Demonstration), Roma, Italy, Sept. 2001.*