Query Processing using Dynamic Relational Structure for Semistructured Data

B.M. Monjurul Alom, Frans Henskens and Michael Hannaford

School of Electrical Engineering. & Computer Science, University of Newcastle, AUSTRALIA

Summary

The most promising and dominant data format for data processing and representation on the Internet is the semistructured data form termed XML. XML data has no fixed schema; it evolved and is self describing which results in management difficulties compared to, for example, relational data. It is therefore a major challenge for the database community to design query processing techniques and storage methods that can retrieve semistructured data efficiently. In this paper, we present a querying scheme for semistructured data views of relational form. The proposed technique stores element-paths, attributes, contents of the element paths and attributes, and XML processing instructions in a dynamic relational structure termed as Multi-XML-Data-Structure (MXDS).

The technique supports different kinds of queries (such as structural-join, general path query, and twig query) and the existence of attributes in the XML documents. We use a secondary index on MXDS to reduce the search time. The indexing is based on the (assigned) elementary-code value (not encoded value). The proposed technique also supports dynamic data manipulation. Experimental results show the twig query execution time of our proposed technique outperform than that of some other XML query processing techniques (such as TwigStack, TwigStacklist). We performed experiments for relational query using MySQL and ORACLE. To compare with the XML query processing time we also measured query execution time using the XQuery language. We then we analysed the query performance between XML and relational queries.

Key words:

TwigStack, Structural Join, XPath, XQuery, MySQL, Oracle.

1. Introduction

Query processing is an essential part of any type of databases as well as Semistructured (XML) databases. The growth of XML repositories on the Web has led to much research on storing and indexing for efficient querying of XML data. XML query processing is much more complicated than traditional query processing methods because of the structure of XML [1]. A path expression specifies patterns of selection predicates on multiple elements related by a tree structure named Query Tree Pattern (QTP). Consequently, In order to process an XML query, all occurrences of its related QTP should be distinguished in the XML document. This is an expensive task when huge XML documents are attended.

By semistructured (XML), we mean that although the data may have some structure, this structure is not as rigid, regular, or complete as the structure required by traditional database management systems [2]. XML data is becoming more and more prevalent for use in performing simple integration of data from multiple sources.

XML is the dominant data exchange format for Internet-based business applications. It is also used as the data format for automated tasks such as information extraction, natural language processing, and data mining [3]. When in XML form, data is neither table-oriented as in a relational database, nor is it strictly typed as in an object database. Rather, XML data comprises hierarchies that have no fixed schema. While XML form supports Internet transport and certain data processing tasks, it causes issues for other common activities such as querying and updating.

One option for managing semistructured as well as XML data is to build a specialized data manager that contains an XML data repository at its core [4]. It is difficult to achieve high query performance using XML data repositories, since queries are answered by traversing many individual element to element paths requiring multiple index lookups [5]. In the case of XML data, queries are even more complex because they may contain regular path expressions [6]. Additional flexibility is needed in order to traverse data whose structure is irregular or partially unknown to the user.

Another option for managing semistructured data is to store and query it with a relational database [4]. In the database community many researchers argue that the relational (and object-relational) model still is the best option due to its maturity and widespread usage [7].

The well known query processing method termed as Structural Join is described in [2]. In Structural Join, query is decomposed into some binary

Manuscript received April 5, 2010
Manuscript revised April 20, 2010

join operations. Thus, a huge volume of intermediate results are produced in this method. The Holistic twig join approach [8] do not decompose the query into its binary Parent-Child (P-C) or Ancestor-Descendant (A-D) relationships but they need to process all of the query nodes in the document. The query processing method termed TJFast [9] which only process elements which belong to the leaves of QTP instead of processing all the nodes in the XML document. But this method use a structure named Finite State Transducer (FST) for decoding the code of nodes into the same name of the path traversed from the root for each node, so FST waste a lot of time.

In this paper the presented querying scheme for XML (semistructured) data views of relational form stores element-paths, attributes, contents of the element paths and attributes, and XML processing instructions in a dynamic relational structure termed as Multi-XML-Data-Structure (MXDS). The query processing technique termed DRXQP which supports different types of query (such as twig query, structural join query and general path query) while maintaining the semantic intent of XML data. The proposed technique is capable of handling large XML data for representation in dynamic relational structure.

Experimental results show the twig query execution time of our proposed technique outperform than that of some other query processing techniques (such as TwigStack, TwigStacklist). We have done a large number of experiments on the existing relational and XML query processing techniques such as MySQL, Oracle and XQuery language. Experimental results also show the query execution time of our proposed technique outperform than that of XQuery and Oracle; although slightly slower than highly regarded MySQL.

The remainder of this paper is organized as follows: related work is described in section 2. Framework of the proposed technique is described in section 3. The proposed query processing technique and experimental results are presented in section 4 and 5. The paper concludes with a discussion and final remarks in section 6.

2. Related Work

Query processing techniques such as Holistic Twig Join methods have been proposed in [10-13] to process a twig query efficiently; however, they still suffer from large number of redundant function calls. A new approach termed TwigStack+ is proposed in [14] to solve this problem, which based on holistic twig join algorithm that greatly improve query processing performance. The TwigStack+ technique is used to reduce the query processing cost, simply because it can check whether other elements can be processed together with current one. The proposed technique also used to check the usefulness of an element from both forward and backward directions.

TSGeneric[10] made improvements on TwigStack by using XR-Tree to skip some useless elements which have Solution Extensions but cannot participate in any path solution. TwigStackList [11] handles the sub-optimal problem by attaching an element list to each query node to cache some elements, TJFast [9] improved the query processing performance by scanning elements of leaf nodes in the query to reduce the I/O cost. Although the existing methods [Haifeng:2003] can guarantee the optimality of CPU time and I/O when only AD edges involved in the twig pattern, they all suffer from large number of redundant function (getNext(root) calls.

A query processing and update processing method termed EXEL (Efficient XML Encoding and labeling) is presented in [15]. EXEL enables complete avoidance of re-labeling for updates while providing fairly reasonable query processing performance. The labeling scheme is simple but effective to compute the structural relationship. In this approach, a novel binary encoding method is used to generate ordinal bit strings.

SIGOPT (schema information graph) to optimize XML query processing is described in [16]. The presented technique explores the opportunities for schema information to affect the query evaluation performance. The main goal of the method is to develop a practical solution that can perform well within the constraints of an optimizer. For this purpose a simple structure, called Schema Information Graph (SIG) is used to store metadata knowledge. Multi-level operator combination in XML query processing is described in [17] which elaborates the importance to consider the operations at each level. Specifically, the technique considers the influence of projections and set operations on pattern-based selections and containment joins.

There are some query language for semistructured data such as Lorel [18], XML-QL [19], XQL, XML-GL, XSL[20], XPath[21], XQuery [22], UnQL[23], Quilt [7]; however these query languages are complicated to use and have some limitations. XQuery is the most standard, powerful query language also easy and flexible to use. XQuery has the ability to work with data without a predefined schema [7]. It is also used to query several documents simultaneously. But XQuery sometimes leads to unexpected query results and prevents index exploitation [24]. XPath is mainly understood as a language for selecting a subset of the nodes of an XML document tree [25]. All major XPath engines take exponential time on the size of the input queries.

Query rewrite and optimization is more complex for XML queries than for relational queries [24]. XML-QL, XQL do not support update languages [20]. XML-QL, XQL, XQuery, XPath, XSL do not support reduction operations. XSL and XQL do not support any join operations. XSL and XQL, Lorel, XML-GL do not support schema order, but XML-QL supports this [20].

3. Framework of the Proposed Technique

The technique uses a dynamic structure termed "Multi XML Data Structure" (MXDS) that complies with relational structure, supporting the use of relational tools such as query languages for dynamic data manipulation. A Multi XML Stack is the basis for the dynamic MXDS to store the parsed XML data. We use an elementary code table to assign the code value for each type such as element, attributes, data values and processing instructions. MXDS stores the encoded value for each type (root-element, others-element, attributes, data-values, and processing instructions) with their assigned preliminary code value. The encoded values are calculated on the basis of their parent-child relationship using a hash constant. The parent-child relationship is maintained on the multi stack structure.

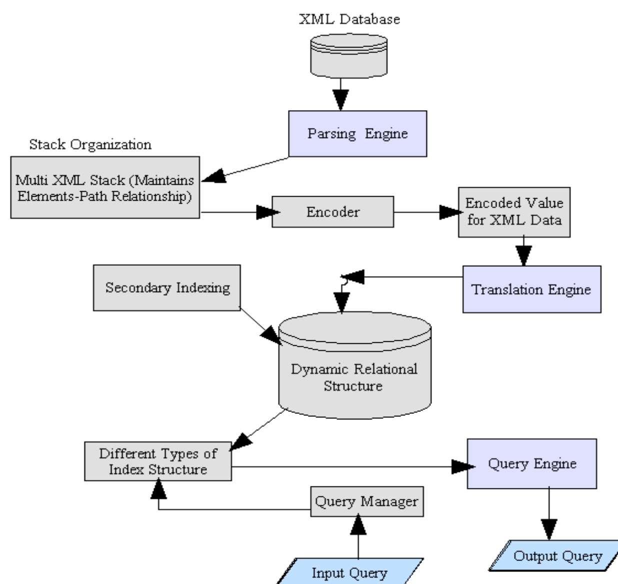

Fig. 1 Architecture of the Query Processing Methodology.

The technique supports the existence of attributes in the XML documents. We use a secondary index on MXDS to reduce the search time. The indexing is based on the (assigned) elementary-code value (not encoded value). The overall structure of the proposed technique is presented in Fig. 1. The XML document is parsed through the parsing engine to store into a multi "XML-stack-structure" which is the basis for encoding the XML data. MXDS also maintains the elementary code for each of the data types by assigned a prime code value. The elementary code value is presented in Table 1. Multi stack structure can be thought of as a tree in the sense that it maintains relationships such as parent-child, ancestor-descendant etc. Each root-path of the document and all of the child elements and attributes of the root-path are stored in the initial (first) stack of the multi-stack-structure. Similarly for all the descendants of a root-element path, the (child) stacks are created based on their parent relationship. This stack structure formation continues until storing of the data values (content of element tags, and attributes) is completed. A multi stack structure for XML data is presented in Fig. 3 for the XML data in Fig. 2. An encoder is used to produce value for the content of each stack entry and all its descendants. We assume the encoded value for root element is zero (0).

```
<Recipe name="bread" prep_time="5 mins">
    <title>Basic bread</title>
    <Ingredient_info>
        <Ingredient unit="dL">
            <Name>Flour </Name>
            <amount>8</amount>
        </Ingredient>
        <Ingredient unit="dL">
            <Name>Water</Name>
            <amount>4</amount>
        </Ingredient>
    <Ingredient_info>
    <Instructions>
        <step>Mix all ingredients together.</step>
        <step>knead thoroughly.</step>
    </Instructions>
</Recipe>
```

Fig. 2 XML Document for Recipe.

Table 1. Elementary Code Table.

Type Name	Value
Root Element	1
Other Element	3
Attribute	5
DATA Value	7
Processing Instruction	11

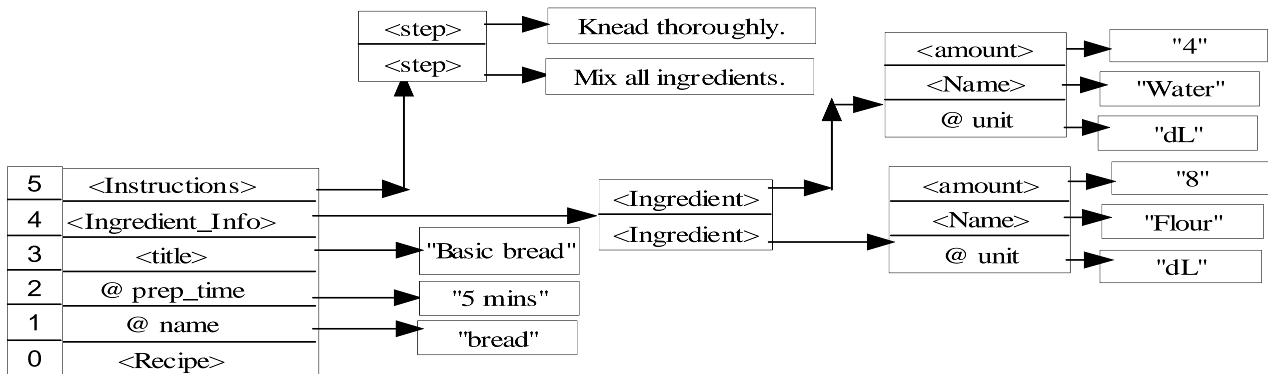

Fig. 3 Multi Stack structure for XML Data.

Table 2. MXDS.

Type-Name	Encoded-Value	Type-Value
<Recipe>	0	1
@name	1	5
"bread"	11	7
@prep_time	2	5
"5 mins"	21	7
<title>	3	3
"Basic bread"	31	7
<Ingredient_info>	4	3
<Ingredient>	41	3
@unit	411	5
"dL"	4111	7
<Name>	412	3
Flour	4121	7
<amount>	413	3
"8"	4131	7
<Ingredient>	42	3
@unit	421	5
"dL"	4211	7
<Name>	422	3
Water	4221	7
<amount>	423	3
"4"	4231	7
<Instructions>	5	3
<step>	51	3
Mix all ingre.....	511	11
<step>	52	3
knead throu......	521	11

Table 3.Indexing on Element Path.

Type-Name	Encoded Value	Type Value
<title>	3	3
<Ingredient_info>	4	3
<Ingredient>	41	3
<Name>	412	3
<amount>	413	3
<Ingredient>	42	3
<Name>	422	3
<amount>	423	3
<Instructions>	5	3
<step>	51	3
<step>	52	3

To find the encoded value of any data type it is highly recommended to search based on the secondary indexing. Table 3, represents a secondary indexing which is based on element value type. Similarly we use secondary indexing on the other data types (such as attributes, data values). To maintain the (semantic) exact information of XML data, we repeat the same element name or attribute name in MXDS. In XML data, it is a common issue that the same path name may represent different information. The encoded values for all other children or descendants are calculated based on the child serial number and hash constant (i.e. encoded value of parent tag * hash constant + child serial number). Each element or attribute or processing instruction or data-value (content of element or attribute) with their encoded value and type-value (prime code value) are recorded sequentially in the dynamic relational structure (MXDS).

A dynamic relational structure is presented in Table 2 for the XML data given in Fig. 2. To reduce the query execution time we use a secondary index on MXDS based on the type-value (each type has different type value in Elementary code table). For example, a title could be the "Book title" or "Person Type either Mr or Mrs or Dr or Professor". To maintain this semantic meaning we repeat the same path or attribute name in MXDS with their different encoded values.

We consider hash constant 10 for the given XML data in Fig. 2. For the large XML document we could use the hash constant 10,000 or 100,000 or 1 million. The hash constant is used to encode the value for each element or attribute or data-value. The idea behind these numbers is to support the maximum number of children within an element path. If we analyze the realistic XML datasets (Bib.xml, Yahoo.xml, Protein_Sequence.xml, Dblp.xml) in [26], we see very few of these have 1 million or 100,000 children within an element.

4. Query Organization

The proposed algorithm is presented in section 4.1, the explanation of different types of query is described in section 4.2, and section 4.3 presents the search time analysis.

4.1 Algorithm for the Proposed Technique

```
Algorithm DRXQP ()
Begin
    Parsing();
    Encoding();
    Searching();
End;
```

```
Parsing()
Begin
Separate each child-element (either
attribute or element) of Root-Tag from
XML document and store onto the Stack;
  For (each child-element in the Stack)
  do

Begin
  If (child-element has nested element
  or children) then
  Begin
     Create stack for the nested
     element;
     If (nested-element has attributes
     & DATA-Value) then
     Begin
          Create stack for DATA-
          Values and attributes;
          Sequentially store DATA-
          Values, attributes onto the
          stack;
  End; //If End;
  If (child-element has nested element
  and nested-element contains DATA-
  Value) then
  Begin
     Create the stacks for
     corresponding nested element and
     DATA-Value;
  End; //If
  If (child-element has no nested
  element except DATA-Value) then
  Begin
     Create the only stack for
     corresponding DATA-Value;
  End; //If End
  End; // For End
End; //Parsing End
Encoding ()
Begin
  Create Elementary_Code_Table;
  Assign the prime codes (1, 3, 5, 7,
  and 11) for each data type (such as
  root-element, others-element,
  attributes, Data-Values, and
  Processing Instructions);
  The encoded value for each child-
  element or attribute or Data-Value is
  calculated as follows:
  Encoded_Value (child-element) =
  Encoded_Value_of_Parent_Tag *
  hash_constant + child_serial_number
  of the Parent_Tag;
  The encoded value for Parent of each
child is calculated as follows:
```

```
   Encoded_Value        (Parent-element)
   =Encoded_Value_      Child_Tag       /
   hash_constant;
   The serial _number of each child is
calculated as follows:
Child_Serial_Number
=Encoded_Value_Child_Tag%hash_constant;
End;
Searching ()
Begin
   Input search_key;
Search the Encoded values of the
search_key based on their type-values
(using secondary index);

Calculate the Parent_Encoded_Value or
child_encoded_value for the search_key;
Parent_Encoded_Value=Encoded Value of
search_key/ hash_constant;
Searched for the "desired output" in
the (E-Index or Attribute-Index or
Value-Index) structure according to the
obtained parent_encoded_value or
child_encoded_value;
End;
```

Fig. 4 Algorithm for the Proposed Technique.

4.2 Explanation of Different Types of Query

The functionality of the searching scheme is demonstrated in the following examples:

Query #1:
/Recipe/Ingredient_info/Ingredient/[amount="4"] / Name
Find (the content of) Ingredient Name that matches with the amount=4. (This type of query is known as twig query)
Answer:
The Data-value of element amount is 4. The assigned elementary code for data value is 7 (from elementary code table). On the basis of the assigned elementary code value of data, it is searched (using Data-Index structure) for the encoded value of "4". We see encoded value of "4" is 4231. The parent of "4" is element amount (from multi-stack-structure). So the system calculates the encoded value of amount, which is 4231/10 (hash constant) =423.
The elements "amount and Name" are both siblings and "Ingredient" is their parent. To find the encoded value of Name, it is required to find the encoded value of its parent "Ingredient". The encoded value of Ingredient=423/10=42. The encoded value of Name (which is a child of Ingredient) is calculated as 42*10+2 (Child serial number of Name, from multi stack structure) =422. The encoded value of the desired output (Name which has the only child) = 422*10+1=4221.

The techniques then find (using Data-Index structure) the Type-name according to the encoded value 4221 and we see it is "***Water".***

Query # 2:
/Recipe//step
Find the "step" element (does not matter where step occurs?) (This type of query referred to structural join).
Answer:

The name of the element is step. The assigned code for the element is 3. On the basis of the assigned elementary code value of element, it is searched (using E-Index structure) for the encoded value of step without having any specific condition.

We see two encoded values for step are 51 and 52. As there is no condition associated with the step element, the technique directly calculates the encoded values of the child of step elements:
$EV_{step\text{-}child}$=51*10+1=511
$EV_{step\text{-}child}$=52*10+1=521.

The technique then searches (using Data-Index structure) for the corresponding Name of these encoded values. We see these are "***Mix all Ingredients"*** and "***Knead thoroughly".***

Query # 3:
/Recipe/ title
Find the title of the Recipe.
Answer:
On the basis of the assigned code value of element, it is searched (using E-Index structure) for the encoded value of "title". The encoded value of "title" is 3. The child-encoded value (title which has the only child) of title is calculated as 3*10+1=31.
It is then searched (using Data-Index structure) for the title name of the corresponding encoded value 31. We see it is "***Basic bread***".

Query # 4:
/Recipe/Ingredient_info/Ingredient [@unit= "dL"]/ Name
Find the list of Ingredients whose unit is "dL". This type of query is based on attribute searching.
Answer:

The searching technique is similar to that described in query #1.

4.3 Search Time Analysis of the Technique

To reduce the search time, we apply secondary indexing on MXDS based on the value types (such as elements, attributes, Data-values, and processing instructions). The indexing avoids searching the whole MXDS structure each time. On the basis of the query input (data), the searching is applied to the corresponding

indexed structure to get the encoded value. After then, it is calculated the parent's encoded value or child's encoded value (using formula based on hash constant). According to these encoded value, the MXDS is searched to find the element name or attribute or entity or processing instruction.

Let N_E be the total number of element paths, N_{Att} be the total number of attributes, and N_{proc} be the total number of processing instructions and N_{Data} be the total number of data-values (content of the Element-paths and attributes).

To find the encoded value (using E-Index structure) based on the element type value, total search time:

$$T_{Encod-Ele} = O(N_E) \quad (1)$$

To find the element name or attribute or entity or processing instruction (using Data-Index structure), total search time:

$$T_{EAP} = O(N_E) + O(N_{Att}) + O(N_{proc}) + O(N_{Data}) \quad (2)$$

Total search time to find element path=

$$T_{Encod-Ele} + T_{EAP}$$

$$= 2 * O(N_E) + O(N_{Att}) + O(N_{proc}) + O(N_{Data}) \quad (3)$$

To find the encoded value from indexed structure based on the Attribute type value, total search time:

$$T_{Encod-Att} = O(N_{Att}) \quad (4)$$

Total search time to find Attribute= $T_{Encod-Att} + T_{EAP}$

$$= O(N_E) + 2 * O(N_{Att}) + O(N_{proc}) + O(N_{Data}) \quad (5)$$

To find the encoded value from indexed structure based on the Data type value, total search time:

$$T_{Encod-Data} = O(N_{Data}) \quad (6)$$

Total search time to find Data value= $T_{Encod-Data} + T_{EAP}$

$$= O(N_E) + O(N_{Att}) + O(N_{proc}) + 2 * O(N_{Data}) \quad (7)$$

Considering equation (3), (5) and (7), we see the search time to find element-paths or attributes or data values depend on their total number of existence in the document.

In general, if we analyze the XML document the probability of the existence of Element-paths and their contents are more than any other types (such as Attributes or Processing instructions or Entities). Hence using the proposed technique, the search time (of Attributes or Processing instructions or Entities)

$T_{Attribute}$ or T_{Entity} or $T_{Process-Ins} < T_{Element-Path}$.

5 Experimental Results

We used Oracle 9i (Enterprise Edition Release 9.2.0.8.0) and Stylus Studio 2009 XML Enterprise Suite Release 2, to evaluate the different query results in the case of a centralized system. We used an Intel Processor with 2.13 GHz, 1.99 GB of RAM under the Windows XP professional operating system. To support the Oracle 9i database we used the Linux operating system. We used the XML datasets (Bib.xml, Yahoo.xml, Protein_Sequence.xml, XMark.xml, Dblp.xml) in [26] to run the comparisons between the XML Query processing and DRXQP technique.

Table 4: Queries used in our experiment.

Query	Data set	XPath Expression
Q-1	DBLP	/dblp/msthesis/title
Q-2	Yahoo	yahoo/listing/seller_info/seller_name
Q-3	Nasa	nasa/datasets/dataset/author/firstname
Q-4	Yahoo	yahoo//memory
Q-5	XMark	//listitem[//bold]/test//emph

File Size (MB)	ORACLE	XQuery	BIQS	MySQL	DRXQP
1.06	1.04	.421	.12	.107	.157
2.59	2.56	1.03	.304	.262	.766
8.68	8.6	3.45	1.02	.879	1.25
15.1	11	4.1	1.85	1.53	2.5
22.4	16.31	6.08	2.74	2.28	4
34.2	24.9	9.28	4.18	3.48	5.5
42	30.5	11.4	5.13	4.27	7
53.3	38.7	14.46	6.51	5.42	9
64.6	46.9	17.5	7.9	6.57	11

Table 5: Execution Time (in Sec).

Table 6: Query Time using Different Techniques.

F-size/ Q-Time (Sec)	12 (MB)	57 (MB)	113 (MB)	174 (MB)	232 (MB)
TSGeneric +	.5	4	8	12	17
DRXQP	.65	4.2	9	12.5	17.5
TwigStack	1	4.4	9.5	13.25	18
TwigStacklist	1.5	5	10.5	14.5	19
TwigStack+	.4	2.5	6	7.5	11
TwigStack+B	.3	2	4	6.5	8

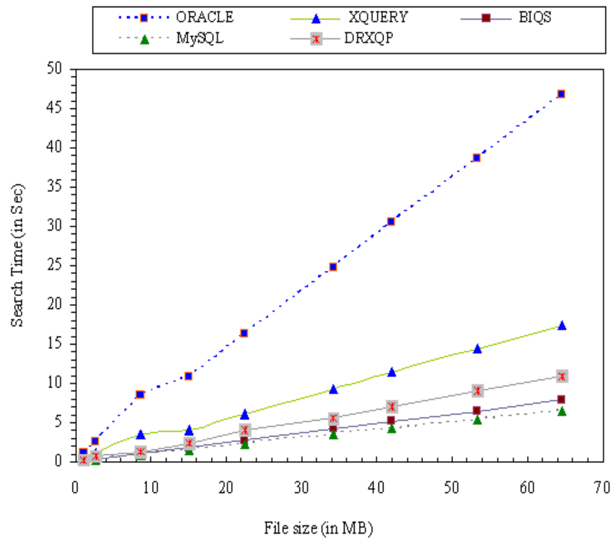

Fig. 5 Time Comparison using Various Method.

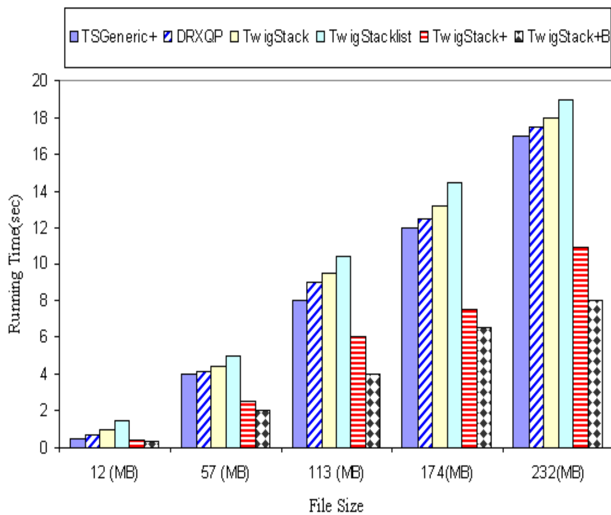

Fig. 6 Comparison of Query Execution Time using Different Techniques using Q-5.

To compare with other relational databases like MySQL, we used our own (custom) generated database named Personal-info comprising different file sizes (5.78 MB, 11 MB, 34.14, 53.03 MB, 104.46 MB, 130 MB, and 683 MB) and consisting of millions of tuples in the database relations. We measured the time (in sec) with respect to each query operation using different numbers of predicates by using Java Eclipse. Java Eclipse is connected with the MySQL database for the execution of different (MySQL DB) query operations.

The comparison analysis for DRXQP, XQuery execution time, execution time in Oracle, MySQL are presented in Fig. 5 and also described in Table 5. To measure the execution time of DRXQP, We took the average query execution time through the queries (Q-1 to Q-4) given in Table 4. Q-1 to Q-3 are the simple path queries and Q-4 is a structural join query. To measure the Twig query execution time, we used the query Q-5 given in Table 6. To compare with other TwigStack queries, we used the different file sizes (12 MB, 57 MB, 113, 174 MB, and 232 MB) for the given query Q-5 in Table 6. The tabular representation of Twig query execution time is presented in Table 6. The corresponding graph is presented in Fig. 6. It can be seen from Fig. 5 and Fig. 6, twig query execution time is more than general path query although for same file size.

It can be seen that the execution time of MySQL is superior to XQuery and highly regarded Oracle execution times across the range of predicates tested. The query execution time of DRXQP outperform than that of ORACLE, XQuery but not better than MySQL and BIQS (our proposed technique described [27]). Fig. 6 clearly shows that, the Twig query execution time of DRXQP is better than some other query processing techniques. Comparison analysis also shows relational query processing time is more efficient than XML query processing time.

6 Conclusions and Future Work

We propose a dynamic relational XML query processing technique which supports different types of query while maintaining the semantic intent of XML data. The proposed technique, termed DRXQP is capable of handling large XML data for representation in dynamic relational structure. DRXQP supports twig query, structural join query and general path query. DRXQP also supports dynamic data manipulation. Experimental results show the twig query execution time of our proposed technique outperform than that of some other query processing techniques (such as TwigStack, TwigStacklist). We have done a large number of experiments on the existing relational query processing techniques such as MySQL, Oracle and XQuery language. Experimental results also show the query execution time of our proposed technique outperform than that of XQuery and Oracle; although slightly slower than highly regarded MySQL and BIQS.

More complex Twig query, aggregate function, deleting, dynamic data updating will be the future research work.

References

[1] V. Garakani, M. Harizi, and M. Harizi, "Effective Guidence-Based XML Query Processing," in *International Conference on High Performance Computing and Communications*, Dalian, China 2008, pp. 605-612.

[2] Al-Khalifa, J. S, K. H.V, P. N, S. J.M, and W. Y, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," in *International Conference on Data Engineering (ICDE)*, San Jose, CA, 2002, pp. 141-152.

[3] A. David, G. David, N. Ashish, C. Knight, and B. Peter, "Semistructured Data Management in the Enterprise: A Nimble, High-Throughput, and Scalable Approach," in *The 9th International Conference on Database Engineering & Application Symposium (IDEAS)*, 2005.

[4] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon, "A Fast Index for Semistructured Data," in *The 27th International Conference on Very Large Databases (VLDB)* Roma, Italy, 2001.

[5] J. McHugh and J. Widom, "Query Optimization for XML," in *VLDB* Edinburgh, Scotland, 1999.

[6] T. Milo and D. Suciu, "Index Structures for Path Expressions," in *ICDT* Jarujalem, Israel, 1999.

[7] A. A. d. Sousa, J. L. Perira, and J. A. Carvalho, "Querying XML Databases," in *The 12th International Conference of the Chilean Computer Science Society (SCCC)* IEEE, 2002.

[8] N. Bruno, N. Koudas, and D. Srivasta, "Holistic Twig Joins:Optimal XML Pattern Matching," in *International Conference on Management of Data (SIGMOD)*, Madison, Wisconsin, 2002, pp. 310-321.

[9] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen, "From Region Encoding to extend dewey: On efficient processing of XML twig pattern matching," in *International Conference on Very Large Databases*, Trondheim, Norway, 2005, pp. 193-204.

[10] J. Haifeng, W. Wei, and L. Hongjun, "Holistic Twig Joins on Indexed XML Documents," in *International Conference on Very Large Databases (VLDB)*, Berlin, Germany, 2003, pp. 273 - 284

[11] L. Jiaheng, C. Ting, and W. L. Tok, "Efficient Processing of XML Twig Patterns with Parent, Child Edges: A Look-ahed Approach," in *International Conference on Information and Knowledge Management*, Washington Dc, 2004, pp. 673-682

[12] B. Nicolas, K. Nick, and S. Divesh, "Holistic Twig Joins: Optimal XML Pattern Matching," in *International Conference on Management of Data (ACM SIGMOD)*, Wisconsin, USA, 2002, pp. 310-321.

[13] C. Ting, L. Jiaheng, and W. L. Tok, "On Boosting Holism in XML Twig Pattern Matching Structural Indexing Techniques," in *International Conference on Management of Data (ACM SIGMOD)*, Maryland, USA 2005, pp. 455-466

[14] Y. Zhou, B. C. Ooi, K.-L. Tan, and W. H. Tok, "An adaptable distributed query processing architecture," *Data & Knowledge Engineering* vol. 53:3 pp. 283 - 309 2005

[15] M. Jun-Ki, L. Jihyun, and C. Chin-Wan, "An Efficient XML Encoding and Labeling method for Query Processing and Updating on Dynamic XML Data," *The Journal of Systems and Software,* vol. 82:2009, pp. 503-515, 2008.

[16] P. Stelios, P. Jignesh, and J. H.V, "SIGOPT:Using Schema to Optimize XML Query Processing," in *International Conference on Data Engineering (ICDE)*, Istanbul, Turkey, 2007, pp. 1456-1460.

[17] A.-K. Shurg and J. H.V, "Multi-level Operator Combination in XML Query Processing," in *CIKM*, Virginia, USA 2002, pp. 134-141.

[18] S. Abiteboul, "Querying Semistructured Data," in *The International Conference on Database Theory (ICDT)* Delphi, Greece., 1997.

[19] A. Deutsch, M. F. Fernandez, and D. Suciu, "Storing Semistructured Data in Relations," in *ICDT*, 1999.

[20] A. Bonifati and S. Ceri, "Comparative Analysis of Five XML Query Language," *SIGMOD,* vol. 29:1, pp. 68-79, 2000.

[21] M. Benedikt, W. Fan, and F. Geerts, "XPath Satisfiability in the Presence of DTDs," in *PODS* Baltimore, Maryland, 2005.

[22] S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, W. Yu, D. Tomic, A. Baras, B. Berg, D. Churin, and E. Kogan, "XQuery Implementation in Relational Database System," in *The 31st International Conference on Very Large Databases* Trondheim, Norway, 2005.

[23] P. Buneman, M. Fernandez, and D. Suciu, "UnQL: a query language and algebra for semistructured data based on structural recursion," *The VLDB Journal,* vol. 9, pp. 76-110, 2000.

[24] A. Balmin, K. S. Beyer, and F. Ozcan, "On the Path to Efficient XML Queries," in *The 32nd International Conference on Very Large Databases(VLDB)* Seoul, Korea, 2006.

[25] G. Gottlob, C. Koch, and R. Pichler, "Efficient Algorithms for Processing XPath Queries," in *The 28th International Conference on Very Large Databases (VLDB)* Hong Kong, China, 2002.

[26] http://www.cs.washington.edu/research/xmldatasets/.

[27] B. M. Alom, F. A. Henskens, and M. R. Hannaford, "Querying Semistructured Data with Compression in Distributed Environments," in *International Conference on Information Technology: New Generations (ITNG)* Las Vegas, Nevada, USA: IEEE Computer Society 2009.

Authors Biography

B.M. Monjurul ALom who born in Bagherpara, Jessore, Bangladesh, is a research (PhD) student in the School of Electrical Engineering and Computer Science, The University of Newcastle, Australia. Mr Alom has completed his MSc engineering degree from Bangladesh University of Engineering and Technology, Dhaka. His research interest is Distributed (Structured and Semistructured) Database Management. Mr. Alom was an assistant professor in CSE dept from 2004 to 2007 and a lecturer from 2000 to 2004 in Dhaka University of Engineering and Technology, Gazipur, Bangladesh.

Dr. Frans Henskens is an Associate Professor in the School of Electrical Engineering and Computer Science, Newcastle University Australia. He is also Head, Discipline of Computer Science & Software Engineering, Deputy Head, School of Electrical Engineering & Computer Science, and Assistant Dean IT in Faculty of Engineering & Built Environment. His research interests include engineering of flexible software systems, bioinformatics, operating systems and computer forensics, distributed and grid computing, resilience and availability in database systems.

Dr. Michael Hannaford is Assistant Dean (Postgraduate Studies) of FEBE, and a Senior Lecturer in the School of Electrical Engineering and Computer Science at the University of Newcastle. His research interests are in the areas of Distributed Computing, and Programming Language Design and Implementation.