Indonesian Journal of Electrical Engineering and Computer Science
Vol. 4, No. 1, October 2016, pp. 184 ~ 194
DOI: 10.11591/ijeecs.v4.i1.pp184-194

Hybridation of Labeling Schemes for Efficient Dynamic Updates

Su-Cheng Haw*, Samini Subramaniam, Wei-Siang Lim, Fang-Fang Chua
Multimedia University, Persiaran Multimedia, 63100, Cyberjaya, Malaysia
*Corresponding author, e-mail: sucheng@mmu.edu.my

Abstract

With XML as the leading standard for data representation over the Web, it is crucial to store and query XML data. However, relational databases are the dominant database technology in most organizations. Thus, replacing relational database with a pure XML database is not a wise choice. One most prominent solution is to map XML into relational database. This paper introduces a robust labeling scheme which is a hybrid labeling scheme combining the beauty features of extended range and ORDPATH schemes to supports dynamic updates. In addition, we also proposed a mapping scheme based on the hybrid labeling scheme. Our proposed approach is evaluated in terms of (i) loading time, (ii) storage size, (iii) query retrieval time, and (iv) dynamic updates time, as compared to ORDPATH and ME schemes. The experimental evaluation results show that our proposed approach is scalable to support huge datasets and dynamic updates.

Keywords*: Query processing, XML database, labeling scheme, indexing, dynamic updates*

Copyright © 2016 Institute of Advanced Engineering and Science. All rights reserved.

1. Introduction

XML is a markup language for documents containing structured information for data exchange due to its simple and flexibility characteristic in nature [1, 2]. It can be used in many aspects, and is already widely used in most application domains such as e-Commerce, digital libraries, and so on. Nevertheless, storing and retrieving XML data still remains as a challenging problem.

Generally, XML can be stored using traditional databases (relational database, object-oriented database) or building a specialized native storage. Relational Database Management System (RDBMS) has been dominant in the market since several decades ago, and it is believed that it will stay on for the next couples of years. It is hard for enterprise to switch to a XML database purely, as they have already invested trillions of dollars in relational database. Besides, people still chooses RDBMS over other databases due to its stability, portability, scalability, maturity, and rich functionality (including support over XML data) [3]. Thus, it is crucial to have a mapper to store and retrieve XML data via relational database.

As XML data are semi-structured content, its nodes consist of basic relationships such as ancestor-descendant (A-D), parent-child (P-C), and sibling. Labeling scheme plays an important role to provide quick identification of the relationships among nodes. Many existing labeling approaches merely support static query processing, i.e., it is assumed that the structural information will not be changed over time. However, with the rapid growth of technology, application data are subject to frequent changes. In order to make XML into a full-featured format, it is essential to support dynamic updates such as inserting, updating and delete operations, over XML content [4]. By having these updates, it could cause the entire XML tree to be re-labeled, and henceforth, the performance will definitely be affected especially on the huge size of XML database [5]. Thus, a persistent, robust and durable labeling scheme which avoids re-labeling is very much desirable.

In this paper, we propose an approach for storing large XML data into relational database, which also supports dynamic updates with least needs to re-label the nodes. Our proposed approach is evaluated by comparing with other existing schemes, in terms of query response time (insertion and retrieval) and database storage, using various types of datasets.

Received April 1, 2016; Revised July 16, 2016; Accepted August 19, 2016

2. Related Works

Some existing labeling schemes are reviewed in this section using the XML example as shown in Figure 1.

```
<dblp>
  <mastersthesis key="ms/Brown92">
     <author>Kurt P. Brown</author>
     <title>PRPL: A Database Workload  Specification Language</title>
     <year>1992</year>
      <school>Univ. of Wisconsin-Madison</school>
  </mastersthesis>
  <article key="tr/gte/TR-0236-09-93-165">
     <author>FarshadNayeri</author>
     <author>Benjamin Hurwitz</author>
     <title>Experiments with Dispatching in a Distributed Object System.</title>
     <journal>GTE Laboratories Incorporated</journal>
     <year>1993</year>
  </article>
</dblp>
```

Figure 1. Example of XML data

2.1. ORDPATH

ORDPATH represents a compressed binary scheme, which compares byte-by-byte to discover relations between nodes. Besides, it also supports insertion of new nodes in any positions. For any new nodes that are to be added in-between of sibling nodes, ORDPATH extends the parent's ORDPATH label with a component for the child, without re-labeling any existing nodes [6]. The even and negative integers are reserved for later insertions. However, ORDPATH suffers from two major drawbacks. Firstly, when the data is huge, the size of ORDPATH label increases as well. Secondly, there is a need of re-labeling since they only reserved the odd and negative number for any new node inserted. Figure 2 shows the tree representation of the XML data, with the ORDPATH labeling. From Figure 2, it shows that for each child node, it contains the label of the parent node with the odd increment starting from 1. Thus, as the depth of the tree goes, so does the labeling size.

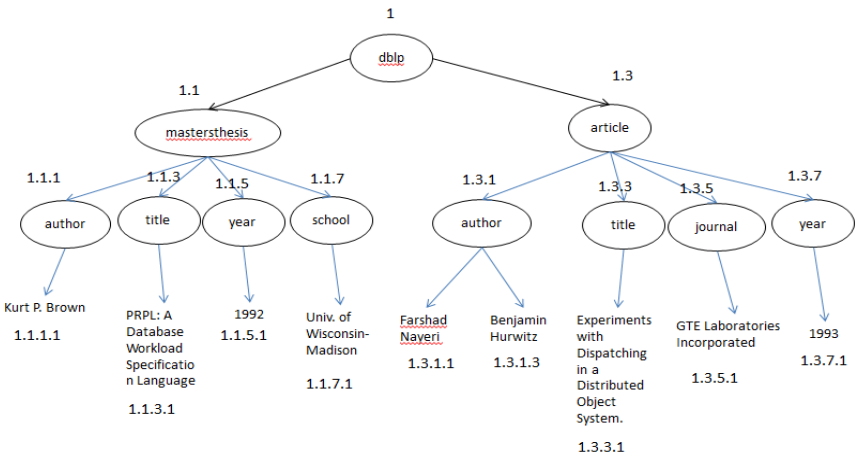

Figure 2. ORDPath labeling scheme

As mentioned earlier, ORDPATH support dynamic insertion by using even-numbered and negative integer components. Insertion between any two siblings (also known as careting in), is done by creating a component with an even ordinal falling between the odd labeling of the two siblings, then following with a new odd component, starting by 1. Figure 3 shows insertion between node 1.1.1 and 1.1.3.

To insert a node "page" between the siblings, a child of node "1.1.2" needs to be inserted between the two siblings. This virtual node acts as a caret for inserting new nodes. The new node "page" will then be inserted as its child node with the label of "1.1.2.1". Both of these nodes represent a complete careting procedure. After inserting node "page", we insert another node, "sub author" between "1.1.2.1" and "1.1.3". Since we already have a virtual node, we can insert the node with label "1.1.2.3" using a simple rightmost insertion [6]. These insertions require no re-labeling of old nodes as the label is always unique and the relations between each nodes is still maintained [7].

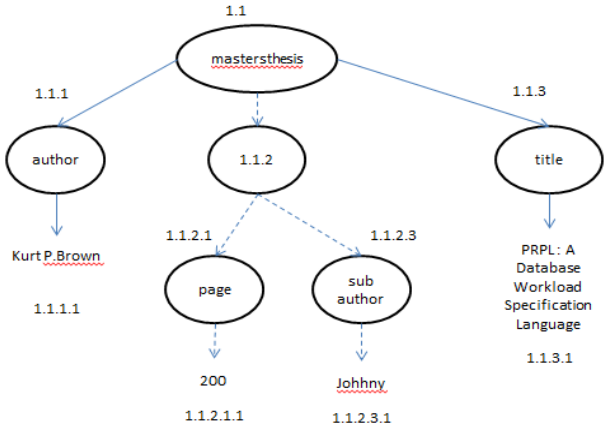

Figure 3. Handling insertion of nodes based on ORDPath labeling scheme

2.2. Multiplicative-Efficient (ME)

Multiplicative-Efficient (ME) labeling scheme uses a combination of odd numbers and multiplication techniques. It is able to maintain and determine the structural relationships. The labeling of ME is defined as (*level*,[*selfLabel*, *ordinal*]), where "*level* defines the level of the node; *selfLabel* is the multiplication of parent label and ordinal; and *ordinal* is the order of the current node using unique odd number starting from 3" [8].

Figure 4 shows the XML tree labeled with ME scheme. The root node of the XML tree will be labeled as 1. Then, the children of the root node will have an odd-numbered ordinal using 2n+1, where n denotes the position of a node in the level. After that, a multiplication of the parent's label with the ordinal will be added to the node's label.

In order to determine the P-C relationship, there are two conditions. First, by using the node's self-label and divide by the ordinal, it should be equals to equals to the parent node label. As for the second requirement, the parent node's level must be one level below the child node. If these two requirements are fulfilled, then it is a P-C relation. For example, from Fig. 4, using the node "mastersthesis" with the label {1,[3,3]} and "title" with label {2,[15,5]}, P-C can be determined by using "title"'s self-label which is 15, divide by its ordinal(5), yields 3, which is also "mastersthesis" self-label. As for A-D relation, there are 4 conditions. First of all, nodeA's self-label must be smaller than nodeD's self-label. Besides, the self-label of nodeD is dividable by the self-label of nodeA and there are no remainders. The third condition is that the self-label of the parent node of nodeD is dividable by the self-label of nodeA, with no remainder. Lastly, the self-label of the sibling node of nodeD is dividable by the self-label of nodeA. For example, node mastersthesis self-label, 3 is lesser than the leaf node "Kurt P.Brown" which is 27. It also can be dividable with a remainder of 0. The self-label of the parent of the leaf node, author, which is 9 can also be dividable by the self-label of mastersthesis. As for the last condition, the sibling nodes self-label, 15, can also be divided by 3 with the remainder 0.

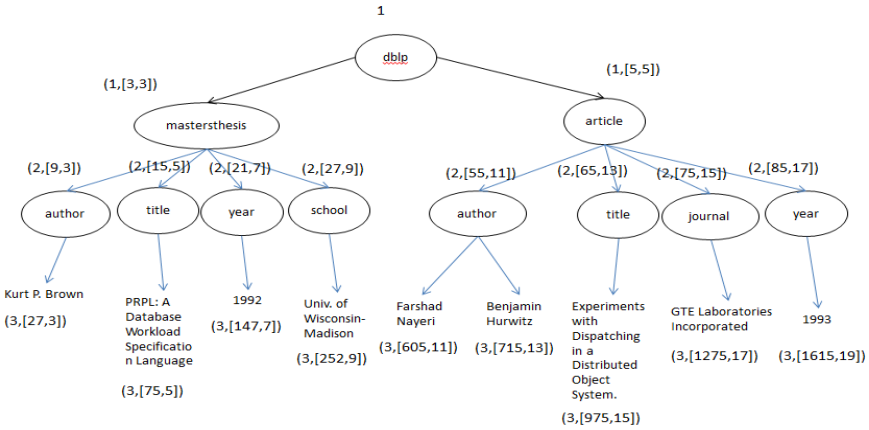

Figure 4. ME Labeling Scheme

ME labeling scheme not only able to maintain the relation among nodes, it also supports dynamic updates. However, this method uses multiplication algorithm, and thus, the size of the ME scheme increases dramatically when the data is huge.

Once all the data are labeled, s-XML [9] is introduced as the mapping technique to store the data. The s-XML is created based on Persistant Labeling Scheme [10]. The s-XML has two tables, namely ParentTable and ChildTable. ParentTable stores the internal nodes whereas ChildTable stores the leaf nodes. The following schemas show the relation representation of both tables: (i) **ParentTable** (*IdNode, pName, cName, Level, LParent, SelfLabel*) where "*IdNode* is the unique Id for the node, *pName* is the Parent's Node Name, *cName* is the Child Name, *Level* is the level of the node, *LParent* is the Parent Label of the node which stores the reference of the parent label, and *SelfLabel* is the self-label or local label of the node which is [n,d] in Persistent Labeling"; (ii) **ChildTable** *(IdNode, pName, Value, Level, LParent, SelfLabel)* with the *IdNode, pName, Level, LParent,* and *SelfLabel* represent the same meaning as the attributes in ParentTable, while *Value* represent the leaf node's value.

2.3. Summary of the Reviewed Labeling Schemes

To summarize the existing labeling schemes that have been reviewed earlier, we analyze them based on four categories, i.e., storage requirement, supported axes, efficiency of extracting relation and dynamic updates efficiency. Storage requirements are the cost of storing the label of each node that was parsed from XML document, processed by each labeling scheme. Support axes defines whether the labeling scheme is able to identify relations between nodes, such as P-C, A-D or siblings. The efficiency of extracting relation are to measure the cost of identifying the relations mentioned, whether they are easily extractable or it comes with a complex process. Dynamic updates efficiency defines whether the labeling scheme supports any update operation such as inserting, deleting and editing, and if re-labeling are needed [11-13]. Table 1 shows the comparison of the reviewed labeling schemes.

Table 1. Labeling schemes and its supported features

Labeling Scheme	Storage Requirement	Supported Axes	Efficiency of Relationship Retrieval	Update Efficiency
ORDPATH	High	A-D, P-C, Siblings	Simply retrieved	Non re-labeling required
ME Labeling	High	A-D, P-C, Siblings	Calculation required	Non re-labeling required

3. Proposed Approach

The proposed approach consists of both labeling and mapping scheme, as they works dependently for inserting XML data to RDBMS. The labeling scheme of our proposed approach is assigned based on depth-first traversal in a form of a *(s-e)l*, where *s* represents the start of the range, *e* represents the end of the range and *l* represents the level of the node.

Nevertheless, *s* and *e* are computed based on the *gap*, which is the $\Sigma(max_{fan\text{-}out}+max_{depth})$. Figure 5 shows the snippet of DBLP dataset annotated with our proposed approach. Firstly, the *gap* must be computed. In this example, the tree has the largest fan-out of 4 and deepest level is 3. As such, the *gap* is 7. Based on the depth-first traversal, the root node will begin with 1 (for the *s*). The *s* for the next node, "mastersthesis", will be assigned with the previous node's *s* added with the *gap* (in this case, it is 1+*7*), followed by author, with 15 as the *s*. Upon returning once a leaf has been reached, the *e* will then be generated by adding the previous running number with the *gap*. For instance if the node is a leaf node with a *s* label of 22, it will have a *e* label of 29, whereas if the node is not a leaf node, it's *e* label will then generated by adding the last child's *e* label with the *gap*.

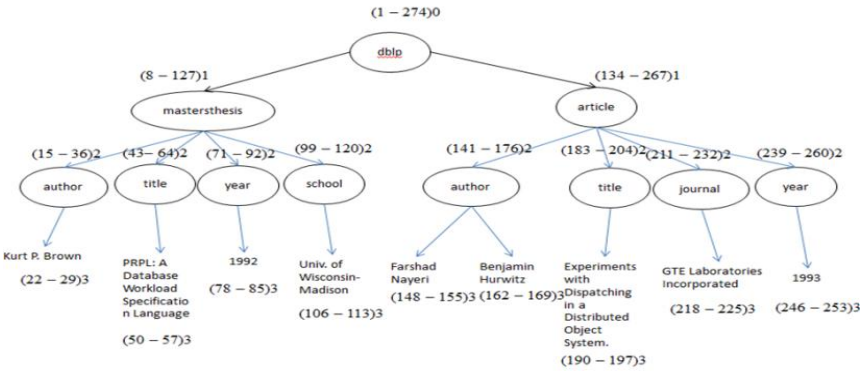

Figure 5. Labeling scheme of our proposed approach

Figure 6 shows the pseudocode for the proposed labeling. Figure 6(a) shows how the helper "gap" is calculated based on the maximum fan out and maximum depth of the tree. The algorithm takes as input a parent node and the next level of the current position. By traversing the parents child node, it will the maximum number of child as the maximum fan out, and as it goes deeper, the maximum level would be assigned as the maximum depth of the tree. As for Figure 6(b), it shows the algorithm for the proposed labeling, where it uses the "gap" calculated from (a) the label. The algorithm runs recursively, by first receive parent node, the current range (the number where the range has reached), and the current level. Then it will assign the Start of the label as the current range, as well as the current level. As stated that the proposed labeling uses depth-first search, it will first traverse the child nodes, and assign each the Start and Level, by using recursive method in Line 10. When the algorithm has reached a leaf node, it will then assign the End as the addition of its Start and Gap, for each of the nodes it passed.

```
   Function GetGap
   input : k - the kth node
   input : nextlevel - next level of the tree
   output: Gap used for labeling
1  MaxFanOut = 0
2  MaxDepth - 0
3  level = nextlevel + 1
4  while( k hasChild, j ) do
5    order = k's number of childs
6    if order > MaxFanOut then
7      MaxFanOut = order
8    GetGap( j, level )
9    if level > MaxDepth then
10     MaxDepth = level
11 return MaxFanOut + MaxDepth
```

(a)

```
   Function AssignLabel
   input : k - the kth node to be labelled.
   input : gap - gap used for identify the range
   input : currentRange - the current range that the algorithm
   has reached
   input : nextlevel - next level of the tree
   output: label - the label of node k.
1  if is root node then
2    start = 1
3    level = 1
4  else
5    startRange = currentRange
6    level = nextlevel
7  currentRange = start + gap
8  level = level + 1
9  While ( k has child nodes, j ) do
10   AssignLabel( j, gap, currentRange, level )
11 end
12 endRange = currentRange + gap
13 return level( startRange - endRange )
```

(b)

Figure 6. Algorithm for (a) Function GetGap (b) Function AssignLabel

As for the mapping scheme, there are two tables, namely iTable (internal table) and tTable (text table). iTable is for storing internal nodes which does not have a text value and tTable is for storing nodes that are leaf nodes. Both tables have the following attributes: (*Start, End, Level, PStart, Value*) where *Start* store s Value of the node, *End* stores e Value of the node, *Level* stores level of the node, *Pstart* stores s Value of the parent node,and *Value* stores element name/text value. Table 2 and Table 3 are the example of sample data on iTable and tTable respectively.

Table 2. iTable (Parent table)

Start	End	Level	Pstart	Value
22	29	3	15	Kurt P. Brown
50	57	3	43	PRPL: A Database Workload Specification Language
78	85	3	71	1992
106	113	3	99	Univ. of Wisconsin-Madison
148	155	3	141	FarshadNayeri
162	169	3	141	Benjamin Hurwitz
190	197	3	183	Experiments with Dispatching in a Distributed Object System
218	225	3	211	GTE Laboratories Incorporated
246	253	3	239	1993

Table 3. tTable (Child table)

Start	End	Level	Pstart	Value
1	274	0	-	Dblp
8	127	1	1	Mastersthesis
15	37	2	8	Author
43	64	2	8	Title
71	92	2	8	Year
99	120	2	8	School
134	267	1	1	Article
141	176	2	134	Author
183	204	2	134	title
211	232	2	134	Journal
239	260	2	134	Year

The proposed approach supports all structural relationships which are P-C relation, A-D relation and sibling. A-D relation is determined with the following formula:

1. if(A(s) < D(s) < A(e)) and (D(level) – A(level) > 1).

Example: Let node1 be journal (211-232)2 and node2 be dblp (1-274)0, (dblp(1) < journal(211) <dblp(274) and journal(2) – dblp(0) > 1). As such, node1 and node2 has A-D relationship.

For P-C relationship, it is determined with the following formula:

2. if (P(s) < C(s) < P(e)) and (C(level) – P(level) = 1)

3. Pstart for C == Start for P (Mapping Scheme)

It is basically similar with the formula for determining A-D, but instead of deducted level is larger than 1, it would be equals 1 since parent would be only 1 level higher than the child. It can also be determined easily from the table by using PStart value.

Example: Let node1 be journal (211-232)2 and node2 be article (134-267)1, (article(134) < journal(211) <article(267) and journal(2) – article(1)=1). As such, node1 and node2 has P-C relationship.

Lastly for Siblings, if the nodes have the same PStart from the table, they are siblings.

Example: Let node1 be author (141-176)2 and node2 be title(183-204)2. From iTable, both have PStart '8'. As such, node1 is a sibling of node2.

3.1. Dynamic Updates

The proposed approach supports dynamic updates such as inserting new nodes or updating values for existing ones, without re-labeling required. For node insertion, we adopted ORDPATH [4] labeling scheme. For insertion between nodes and right most insertion, the e value on the left sibling will be used, but with an addition of byte. As for leftmost insertion, it uses start value instead. Figure 7 illustrates an example of insertion for leftmost, rightmost and

insertion between nodes. The dotted circles and lines represent where the insertion took place. As you can see, a leftmost insertion of “language”, creates a medium node with the leftmost node’s start value with an addition of a byte number “15.1”, then inserts the new node below it with the increase of the byte number along with the level(ignoring the medium node’s level), which the label will be “(15.1)2”. Meanwhile for insertion between nodes, it uses the end value of the left node “36.1”. In a way, the structure of the label remains, and the retrieval of relation between nodes still works with the insertion node.

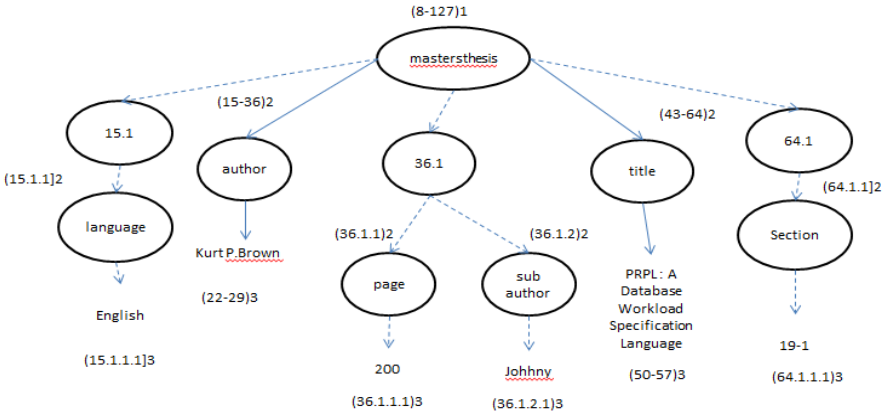

Figure 7. Handling insertion in our proposed labeling scheme

4. Experimental Design

To evaluate the performance, we compare our proposed approach with MELabeling and ORDPATH. The evaluation is divided into three parts: (i) to measure the time to label and store the XML nodes, (ii) to measure the query response time for retrieving data, and (iii) to measure the dynamic updates efficiency response time.

We have selected two datasets with different sizes and characteristics for the evaluation. These datasets are DBLP dataset, and Extended Protein datasets obtained from the University of Washington. The Extended Protein is derived from the original Protein dataset, with duplication on its size, so that we could test on a larger scale. Table 4 shows the characteristics of these datasets.

Table 4. Selected Dataset for Experimental Evaluation

Dataset	File (MB)	Characteristic Description
DBLP	130MB	Max. tree level: 3, structured data
Extended Protein	1.4GB	Max. tree level: 5, unstructured and recursive data

This experiment is performed on a 3.20 GHz AMD Phenom™ II X4 955 Processor, with 12.0 GB Ram on a Windows 8.1 Pro. In order to obtain a better accuracy for the experiment, we run the evaluation three times for each test. The results obtained are the average of these three consecutive runs.

4.1. Labeling and Storing Evaluation

We first evaluate the performance for labeling the XML nodes for the three approaches. Each of the dataset will first parse to nodes using the DocBuilder library, then each of the nodes will be labelled accordingly. After the XML file is being parsed and labelled, we will then store them into RDBMS using the three approaches. As the result, two databases will be created based on the approaches. For comparison, we will measure the (i) time taken for labeling and loading the data, and the (ii) database size. Table 5 shows the evaluation for the approaches in terms of time taken for the loading process, while Table 6 shows the database size for each mapped dataset in RDBMS.

Table 5. Comparison of Various Approaches on Database Loading Time

Dataset	Proposed Approach(min)	MELabeling(min)	OrdPath(min)
DBLP	35	44	34
Extended Protein	728	551	489

Table 6. Comparison of Various Approaches on Mapped Database Size

Dataset	Proposed Approach(MB)	MELabeling(MB)	OrdPath(MB)
DBLP	666.25	673.75	526.493
Extended Protein	7596.20	6711.81	5481.68

From the result, it can be seen that the proposed approach uses lesser time to label the nodes and has smaller size of storage for smaller dataset (DBLP), but uses more time and more storage for large datasets. The is due to the reason that it requires a larger “gap”, which was derived from the maximum of the depth of the tree and the maximum of the fan out of the tree. This requires parsing to be done once before the labeling of each nodes start processing. Thus, it uses more time as compared to the other approaches. Nevertheless, since loading is usually only done one time, this may not affect the operation performance.

4.2. Retrieval Evaluation

For this evaluation, we use two types of queries, which are Path Query and Twig Query. For each type of the query, we have three different SELECT statements. First statement is a P-C query, second statement is a A-D query, and the third is a combination of both. The SELECT statements for Path Query are identified as P1, P2 and P3, whereas Twig Query will be T1, T2 and T3 respectively.

4.2.1. Using DBLP Dataset

Table 7 shows the query description on DBLP dataset. The evaluation results are shown in Figure 8.

Table 7. List of query on DBLP dataset

Query	Query Description
P1	List out all the mastersthesis in year 1991
P2	List out number of the articles that has a sub entity of '1'
P3	List out the title of an article that has a 'i' entity of 'm'
T1	List out all the title of the phdthesis that are in year 1992 and title consist of 'code'
T2	List out all the articles that has a sub entity of "aleph" and sup entity of "2"
T3	List out all the authors in inproceedings that has a sup entity of "n"

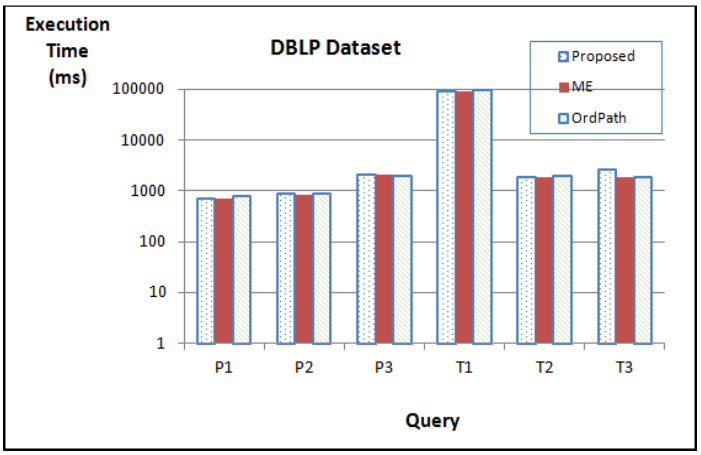

Figure 8. Retrieval evaluation result on DBLP dataset

From the result obtained, we notice the following:

1. For both Path and Twig Queries, the proposed approach uses lesser time compared to the other approaches.
2. For Twig Query 1, it uses more time compared to the other queries. The reason is because the query searches for a specified pattern, which takes more time to process. In this case, it matches values that consist of “code”.
3. Overall, the times taken to retrieve the queries are faster for the proposed approach compared to the others, up to 2.14% faster. This is due to several factors:
   a. The DBLP dataset has only three levels. This causes lesser relations to be created, and thus lesser number of joins.
   b. All approaches has rather similar mapping scheme, i.e., uses two tables for storage, thus for smaller dataset, result shown would not be significant.

4.2.2. Using Extended Protein Dataset

The Protein dataset is 700MB in size, unstructured, contains recursive elements, with five levels of depths. We extended the Protein dataset into 1.2GB and name it as Extended Protein dataset. Table 8 shows the evaluation for the three approaches using Extended Protein dataset. Similar to the test cases in DBLP, six queries (three path query and three twig query) were used to evaluate the performance in using Protein dataset. Figure 9 shows the result of the performance of the three approaches using Extended Protein dataset.

Table 8. List of query on Extended Protein dataset

Query	Query Description
P1	List out all the title of the ref info tat consist of "cytochrome"
P2	List out the number of Protein Entry that consists of note
P3	List out all the Uid from genetics in ProteinEntry.
T1	List out all the length of the summary that has a 'complete' type
T2	List out all the db that are from reference that consist of UID 1748
T3	List out all the description from protein entry that has the keyword that consists of amino

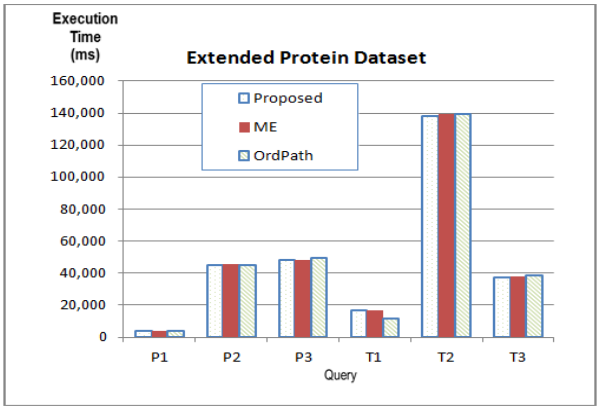

Figure 9. Retrieval evaluation result on Protein Extended dataset

From the result obtained, we notice the following:

1. The proposed approach performs the best for all queries, compared to the ORDPATH and ME.
2. As the dataset size increases, the label of ORDPATH and ME becomes longer and the size of the data increased dramatically. As for the proposed approach, although the gap increases along the size of the XML, retrieval is done by only matching the parent’s node “Start” value with the child node’s “PStart”. Thus this reduces the time of retrieval for the proposed approach.

4.3. Dynamic Updates Evaluation

Dynamic update evaluation is measured by two categories: nodes insertion and node update. This is done by first inserting the existing XML dataset into RDBMS. Then, we will insert a subtree of new nodes into the XML document and parse again. The system will then match the updated XML document with the existing parsed data, to get the new nodes and label them accordingly, then store them. Node update uses the same technique; just that it updates the value of a node instead of inserting new nodes. Table 9 depicts the evaluation results.

Table 9. Dynamic Updates Evaluations on Various Datasets.

Dynamic Update Evaluation	Dataset	Proposed Approach (ms)	MELabeling (ms)	OrdPath(ms)
Insertion	DBLP	184	472	374
	Protein	4,719	8,310	5,155
	Extended Protein	9564	17307	12615
Update	DBLP	2,940	3,144	3,205
	Protein	260,949	264,198	272,231
	Extended Protein	462020	473030	469249

From the result obtained, we observed the following:

1. Our proposed approach perform better (about 51%) for node insertion compared to the other two approaches. ME labeling uses more time as multiplication calculations is needed to generate label for new node.
2. As for ORDPATH insertion, more time is needed, probably due to its large size of label which cause more time for comparison.
3. For node update, since no labeling process is involved, as it only matches the updated nodes label with the existing and overwrites the value.

5. Conclusion

From the evaluations done in this paper, it would seem that the proposed approach has more advantages compared to the other approach. There are several points to support that statement.

First, although the proposed approach has lesser time performance in insertion compared to the others, but it has better retrieval time evaluation As major insertion is usually done only one time, constantly retrieving data have advantage from using the proposed approach. Thus, the proposed approach is a better option for XML data that are constantly being retrieved or updated.

Second, it has better time performance for node insertion compared to the others. Once the original XML document is inserted, users could always insert new nodes efficiently by using the proposed approach. Although other approaches are faster for inserting the huge XML document, it would be inconvenience to insert new nodes afterwards.

Last, but not least, not only it has better retrieval time and node insertion time, the proposed approach doesn't need re-labeling, which could cost a huge impact towards the data. This also means that it supports a larger dataset without the need to relabel them under any circumstances.

Acknowledgements

This work was partially supported by funding from FRGS, Ministry of Education, Malaysia.

References

[1] Haw SC, Lee CS. Node Labeling Schemes in XML Query Optimization: A Survey and Open Discussion. *IETE Technical Review*. 2009; 26(2): 89-101.
[2] Moradi M, Keyvanpour MR. XML and Semantics. *International Journal of Electrical and Computer Engineering.* 2015; 5(5): 1174-1179.

[3] Mayr C, Zdun U, Dustdar S. Reusable Architectural Decision Model for Model and Metadata Repositories Formal Methods for Components and Objects. Lecture Notes in Computer Science. 2008; 5751: 1-20.
[4] Sonawane V, Rao DR. A Comparative Study: Change Detection and Querying Dynamic XML Documents. *International Journal of Electrical and Computer Engineering.* 2015; 5(4): 840-848.
[5] Liang X, Ling TW, Wu H. Labeling Dynamic XML Documents: An Order-Centric Approach. *IEEE Transactions on Knowledge and Data Engineering.* 2012; 24(1): 100-113.
[6] Muller S. Indexing XML Data in a RDBMS using ORDPATH. http://www-db.in.tum.de/~teubnerj/teaching/ss06/xml-proc/ordpath-slides.pdf.
[7] O'Neil P, O'Neil E, Pal S, Cseri I, Schaller G, Westbury N. *ORDPATHS: Insert-Friendly XML Node Labels*. Proceedings of the ACM SIGMOD. 2004: 903-908.
[8] Samini S, Haw SC. *ME Labeling: A Robust Hybrid Scheme for Dynamic Update in XML Databases*. Proceedings of IEEE International Symposium on Telecommunication Technologies. 2014: 126-131.
[9] Samini S, Haw SC, Poo KH. s-XML: An efficient mapping scheme to bridge XML and relational database. *Knowledge-Based Systems.* 2012; 27: 369-380.
[10] Gabillon A, Fansi M. A New Persistent Labelling Scheme for XML. *Journal of Digital Information Management.* 2006; 4(2): 112-116.
[11] Al-Jamimi HA, Barradah A, Mohammed S. *Siblings Labeling Scheme for Updating XML Trees Dynamically*. Proceedings of International Conference on Computer Engineering and Technology. 2012: 21-25.
[12] Al-Shaikh R, Hashim G, BinHuraib A, Mohammed S. *A modulo-based Labeling Scheme for Dynamically Ordered XML tree*. Proceedings of International Conference on Digital Information Management. 2010: 213-221.
[13] Bashir MB, Latiff MS, Ahmed AA, Yousif A, Eltayeeb ME. Content-based Information Retrieval Techniques Based on Grid Computing: A Review. *IETE Technical Review.* 2013; 30(3): 223-232.