CIT. Journal of Computing and Information Technology, Vol. 25, No. 3, September 2017, 211–225
doi: 10.20532/cit.2017.1003557

New Path Based Index Structure for Processing CAS Queries over XML Database

Dhanalekshmi Gopinathan and Krishna Asawa

Department of Computer Science, Jaypee Institute of Information Technology, Noida, Uttar Pradesh, India

Querying nested data has become one of the most challenging issues for retrieving desired information from the Web. Today diverse applications generate a tremendous amount of data in different formats. These data and information exchanged on the Web are commonly expressed as nested representation such as XML, JSON, etc. Unlike the traditional database system, they do not possess a rigid schema. In general, the nested data is managed by storing data and its structures separately which significantly reduces the performance of data retrieving. Ensuring efficiency of processing queries which locates the exact positions of the elements has become a big challenging issue. There are different indexing structures which have been proposed in the literature to improve the performance of the query processing on the nested structure. Most of the past researches on nested structure concentrate on the structure alone. This paper proposes new index structure which combines siblings of the terminal nodes as one path which efficiently processes twig queries with less number of lookups and joins. The proposed approach is compared with some of the existing approaches. The results also show that they are processed with better performance compared to the existing ones.

ACM CCS (2012) Classification: → Information systems → Data management systems → Query language

Information systems → World Wide Web → Web searching and information discovery → Web search engines → Web indexing

Keywords: XML, query processing, CAS query, WWW, index, XPath, database storage

1. Introduction

The extensible Markup Language (XML) is a popular representation structure for expressing nested data. And, it is the accepted standard to represent and transport data on the World Wide Web (WWW). As the number of XML documents on the WWW is growing, there arises a need of an effective way to retrieve data quickly and easily. To retrieve the desired information from the nested structure in an efficient and accurate way is one crucial issue in XML query processing. Database indexing gives a boost for data querying. It helps to find the data in an XML document without traversing the entire document. Hence, it is necessary to develop an effective structure, which indexes the nested structure and content efficiently and supports query processing with high performance.

There are various indexing schemes proposed in the literature. Most of these schemes address different issues of the query processing. In [1], the indexing scheme processes the simple path query without branches. The indexing scheme [2] supports structural queries without any value predicates. The indexing scheme with Root Path and Data Path indexes [3] evaluates XML twig queries with value predicates. The Root Path index stores the prefix, root-to-leaf paths along with the leaf value and reversed schema path. The disadvantage of this scheme is the storage size of the index. There are several other indexes [4] – [6] which have been proposed to answer path queries. But these indexing schemes suffer from some drawbacks such as their increased size, designed for answering only a certain type of queries like only path query without value predicates, and their inefficiency in processing branch query etc.

The objective of this study is to propose an index structure for efficient processing of content and structure queries over nested data stored in XML format. The proposed index structure in this work can handle single path query and twig (branch) queries with or without predicates in a single lookup of the index. The proposed index structure has two indices, namely path index and path combined index. In the first index (path index), all the available root-to-leaf paths are stored as a search key. The second index (path combined index) combines terminal siblings at the same level into one path. This combined path is stored as the search key. The objective of this is to reduce the number of joins when processing the branch query compared to other path based index structures [1] – [3]. A series of experiments are conducted to evaluate the performance of the query evaluation based on the proposed index in terms of two performance metrics – number of elements retrieved and query execution time.

The rest of the paper is organized as follows. Section 2 briefly recalls the preliminary concepts and Section 3 elaborates the proposed system architecture. Section 4 focuses on the experimentation results and their implications are discussed. Section 5 gives the related work and finally, Section 6 concludes the study.

2. Preliminary Study

XML documents are modeled as rooted labeled tree, where trees nodes represent document elements, attributes, and character data. The edges represent the relationship among the nodes of the XML tree. Query processing on XML document is performed by searching the relevant nodes and structural relationships that satisfy the constraints specified by the query. Due to the nested structure of the XML documents, the queries on these documents are expressed as the tree patterns that efficiently capture the structure and content information on the documents. XML query languages like XPath [7], XQuery [8] are expressed as path expressions which can query element nodes and values of the XML documents by specifying structural constraints and value constraints in a predicate form. The structural constraints in the XML query can be specified in the form of a tree, which can be single path query or a twig query (branch) and the value constraints can be specified as value predicates as shown in Figure 1. For example, the XML Query/book [title = "XML"] is a single path Content and Structure Query (CAS) which retrieves all the books with title "XML".

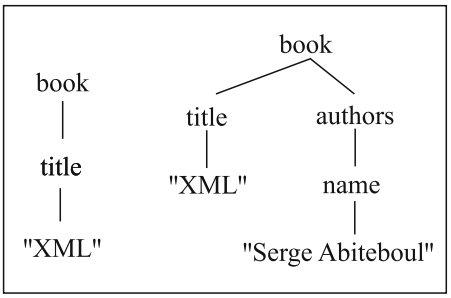

Figure 1. Query tree.

The structure constraint is specified by the path expression – /book/title and the content constraints are specified by the value predicate – title = "XML". The query/book [title = "XML"]/ authors [name = "Serge Abiteboul"] is a branch query. This query selects all authors' elements from the book that has a title "XML" and whose author is named "Serge Abiteboul".

To illustrate the working of the proposed approach, the sample XML document shown below is considered in this paper. Figure 3 shows the labeled tree representation of the sample document. Each node "v" is assigned a unique

```
<book>
<title>Data on the Web</title>
<author>Serge Abiteboul</author>
<section>
<id>"intro" difficulty = "easy" </id>
<title>Introduction</title>
<section>
<title>Web Analysis</title>
</section>
<section>
<title>Web Data and the Two Cultures</title>
<figure>
<title>Traditional client/server architecture</title>
</figure>
</section>
</section>
</book>
```

Figure 2. Sample XML document.

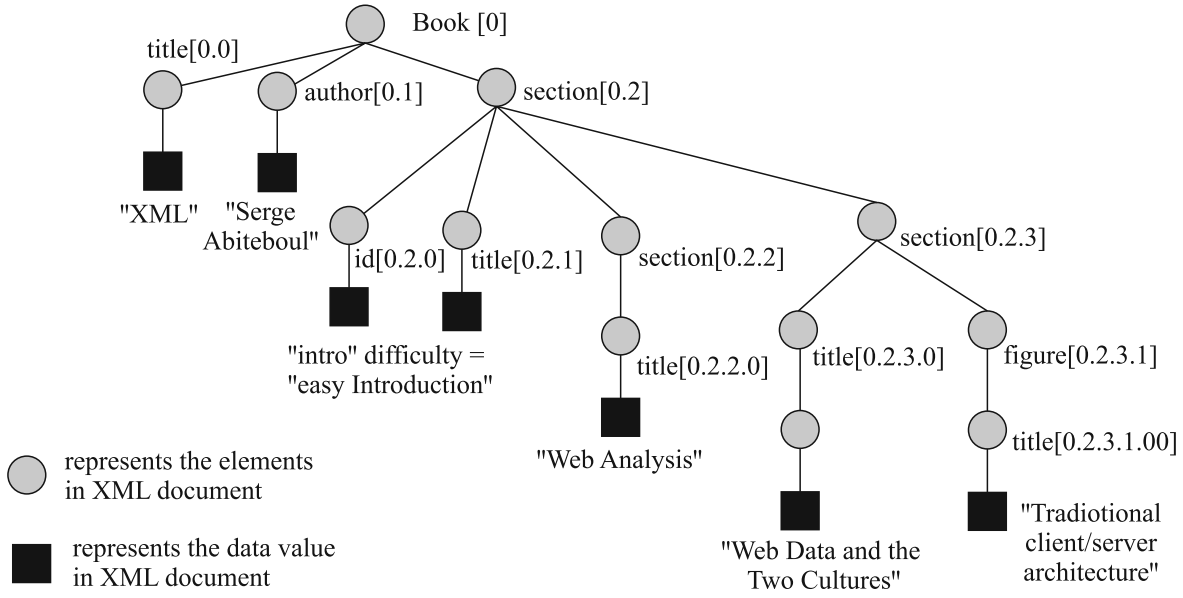

Figure 3. XML tree for the sample document.

identifier known as a label (v). The labels are useful to easily identify the structural relationships among the nodes in the tree. The extended Dewey labeling [9] is used to label each node. Any prefix labeling scheme such as [10] can also be used to label each node in the tree.

2.1. Query Model

This section explains the query model used in the proposed approach. It mainly focuses on three types of queries structures, namely simple path query, recursive query, and branch query.

Each of these queries is processed under two query conditions such as "selecting element by name" and, "selecting element by matching attribute value".

The first one is a structured query where it matches the tag elements in the XML document. And, the second one is referred to as a content and structure query where it matches the structural part as well as the content part of the query.

2.1.1. Simple Path Query

This type of query contains only parent-child relationship. For example, the Q1: /book/section/title retrieves all the titles under the section element whose parent is a book.

2.1.2. Recursive Query

It is a partial match query of the form sep_1 $nodetest_1$ sep_2 $nodetest_2$ ... sep_n $nodetest_n$, where sep_i denotes "/" or "//" and $nodetest_1$ denotes the root of the tree. This type of query will contain at least one A-D relationship (//). For example, the query Q2: book//section/title. The query retrieves all the titles under the section element which has book as an ancestor element. It contains one A-D relationship "book//section".

2.1.3. Branch Query

It is a complex query of the form sep_1 $nodetest_1$ [$pred_1$] sep_2 $nodetest_2$ [$pred_2$] ... sep_n $nodetest_n$ [$pred_n$], where sep_i can be either "/" or "//" and $pred_i$ can be simple path query or recursive query or queries with some predicate values. For example, the query Q3: /book/section [title = "Web"] retrieves all the titles of the section which contains a keyword "Web" in its description.

Table 1 shows the type of sample queries processed by the proposed system. The queries in column 2 represent the content and structure queries which select the elements by matching the attribute values specified as a predicate value. Column 3 represents the structure queries where elements are selected by name.

The next section elaborates the System Architecture of the proposed approach.

Table 1. Types of queries processed by the proposed system.

Query	With value predicate	Without value predicate
Simple Path Query	/book [title = "XML"] /book/section [title = "Introduction"] /book/section/section [title = "Web Analysis"]	/book/title /book/section/id /book/section/title
Recursive Query	//title = "Web Analysis" /book//section/figure [title = "Traditional client/server architecture"]	//section /book //section//title
Branch Query	/book [title = "XML"] /[author = "Serge"] /book [title = "Web"] //author /book//section [title = "Web"]/figure	/book[title]/author /book[title]/author /book/section[id] /title

3. System Architecture

The overall architecture of the proposed system is shown in Figure 4. The proposed system has two main components. The first one is an Index Engine and the second is a Query Evaluation Engine.

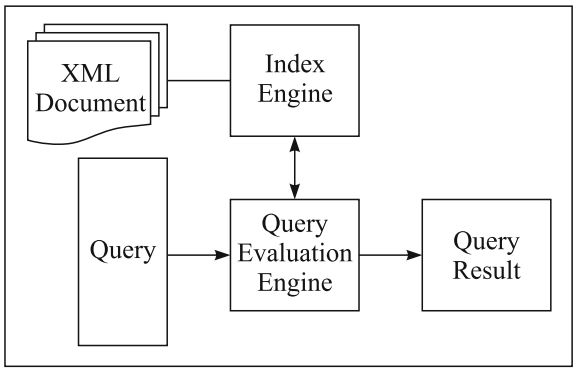

Figure 4. Overall system architecture.

The first component, index engine, parses the XML documents and creates two indices, namely path index (p_index) and path combined index (pc_index). The p_index stores all possible root-to-leaf paths of the XML document. And, the pc_index is constructed by combining terminal sibling nodes of the root-to-leaf paths which have the same ancestor path. It is to be noted that, in general, the branch queries are processed by decomposing them into simple path queries. And, each subquery is processed separately. Finally, the results are merged to obtain the result. By using the path combined index, the proposed approach guarantees that the number of decompositions will be lesser compared to the conventional approach. Both indices have an extended posting list which stores content (or data value) of the document. The construction of the index is explained in subsection 3.1.

Now, the second component of the proposed system – Query Evaluation Engine – takes the query as the input. It first analyses the query structure and constructs a query evaluation plan as per the query criteria present in the input query. It then consults the Index engine to obtain the required result. The query model processed by this engine is explained in subsection 2.1.

3.1. Construction of Index

In general, the indices are defined as the data structures that help to locate specific parts of information form a collection of data. Also it speeds up the query evaluation process. It is noted that an accelerated query evaluation may have to pay additional cost. That is, the index may require additional storage space. And the choice of data structure involved may become significant to create an index structure since it may take less storage space, allow efficient disk access etc.

This section explains the construction of an index of the proposed system. As noted above, it has two indices such as path index (p_index) and path combined index (pc_index). The proposed system uses integration of data structures B+ Tree and HashMap to construct the indices. The Hash Map stores the structural component of the XML document and an offset to the B+ Tree. And the B+ tree stores the content information in the XML document.

The Hash Map of the index engine in p_index stores the root-to-leaf path in the XML documents in each bucket, while the pc_index stores the combined terminal sibling path in each bucket. And the second component – B+ Tree – stores the content information present in the XML document matching the structural part of the query as a posting list. This posting list includes the content information as tuples in the format (NodeName, NodeValue, NodeId).

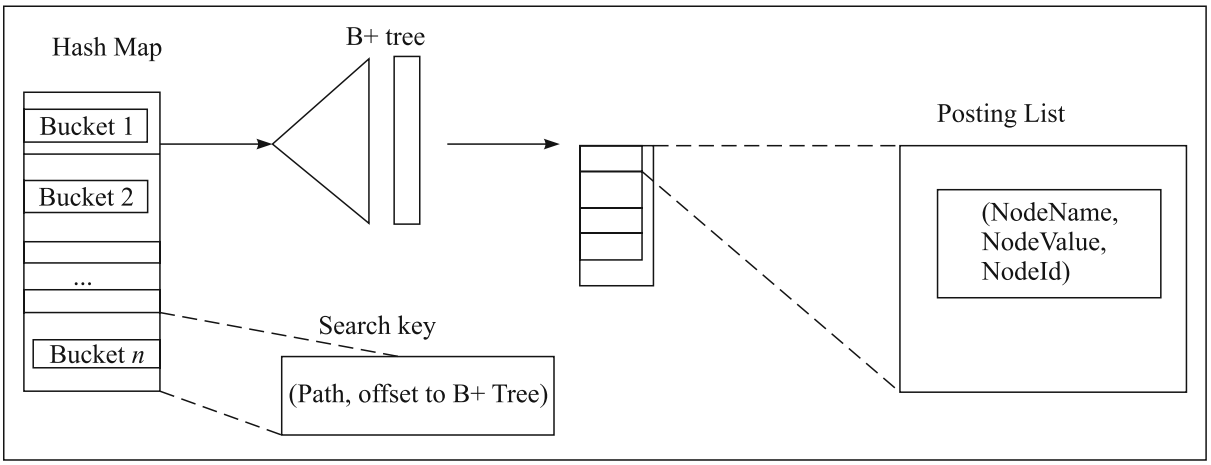

Figure 5. Architecture of the index engine.

The formal algorithm to construct the index engine is given in Algorithm 1.

Algorithm 1. ConstructIndex (I).

Input: XML Document D
Output: Index I (Path index and Path_combined Index)
Begin
For each root-to-leaf path *rp* in D do
If *rp exists in I* **then**
Add the node ids into the posting list;
Else
create a new entry for the path as a search key;
End if
End for
End

Now, to construct the path index, the parser stores all possible root-to-leaf paths as a search key in the Hash Map. And, the extended posting list serves as a value index for the proposed approach. It consists of tuples with the information (Node Value, Node id). Here, the Node value and Node id represent the leaf node in the XML tree. Table 2 shows the path index constructed for the sample document given in this paper.

Next, the second index, path combined index (pc_index) is constructed by combining the terminal siblings with same ancestor paths into a single path. This combined path is the search key. And it has an extended posting list which stores content information of the XML document. Table 3 shows the path combined index (pc_index) for the sample XML document.

Table 2. Path index of the sample document.

Path	With value predicate
\book\title	{("XML",0.0)}
\book\author	{("Serge Abiteboul",0.1)}
\book\section\id	{("intro difficulty = easy", 0.2.0)}
\book\section\title	{("Introduction" ,0.2.1)}
\book\section\section\title	{"Web Analysis", 0.2.2.0), ("Web Data and the Two cultures",0.2.3.0)}
\book\section\section\figure\title	{("Traditional client/server architecture", 0.2.2.1.0)}

Table 3. Path combined index of the sample document.

Path	Posting List
\book\(title |author)	{<title, "XML", 0.0><author, "SergeAbiteboul" ,0.1>}
\book\section\(id|title)	{<id, "intro difficulty = easy", 0.2.0> <title, "Introduction", 0.2.1>}
\book\section\section\title	{<title, "Web Analysis" .2.2.0>, <title, "Web Data and the Two Cultures", 0.2.3.0>}
\book\section\section\figure\title	{<title, "Traditional client/server architecture", 0.2.2.1.0>}

Advantages of storing root-to-leaf path and combined root-to-leaf paths are:

- it reduces the space cost of the index in comparison with other existing approaches [1], [3], [11],
- a single lookup into p_index is sufficient to answer single path query,
- prefix queries (the queries which do not come up to the end of the leaf nodes) of any length is also solved efficiently.

This can be done by using a simple string matching operation with the help of wildcard regular expressions patterns rather than by storing all the prefix paths.

Recursive queries are solved by rewriting them into single path queries and by considering the p_index. Hence, to process all such types of queries it requires only a single look-up of the p_index. The next section elaborates how the path combined index is constructed for the given document.

3.1.1. Path Combining Module

This section discusses how the root-to-leaf paths are pooled to get the combined sibling path. The sample XML Tree shown in Figure 2 is considered for illustrating this concept. All available root-to-leaf paths of this document are listed in column 1 of Table 4. The first two paths in column 1 "/book/title" and "/book/author" differ only in the end nodes and they share a sibling relationship (between them). Hence, these sibling relationships are combined into a single path as "/book/(title|author)". Similarly, the next two paths "/book/section/id" and "/book/section/title" differ in the end node. And it is combined as "/book/section(id|title)". The last two paths are not sharing any terminal sibling relationships. Hence, they are stored in the same way in pc_index.

It is observed that combining the nodes which share the same sibling relationships as above reduces the number of paths to be stored in the index. Thus, instead of storing seven paths, in this case, only four paths are required to be stored in the index. The next section illustrates how the queries are processed over this proposed index structure.

3.2. Query Evaluation Engine

This section elaborates the various phases of the Query Evaluation Engine(QEE) of the proposed system. In this approach, the user query is taken in the form of XPATH query as defined in the query model in Section 2. The QEE has four components (see Figure10): analysis phase, translator phase, plan generator phase and, execution phase.

Each phase takes the input in one representation and produces output in another format.

3.2.1. Analysis Phase

This phase takes the user input in XPATH query form. It analyses the structure of the query by invoking a method called analyseQuery (Q) (explained in the next section). It returns the query structure into a variable called Query Structure Tree (QST). The QST tells whether the user query is a simple path query, recursive query or a branch query.

3.2.2. Translator Module/Rewriting Module

The input to this module is the QST variable, which is generated by the Analysis Phase. The

Table 4. Combined terminal sibling root-to-leaf path.

Root-to-leaf path	Combined Terminal root-to-leaf path
/book/title	/book/(title|author)
/book/author	
/book/section/id	/book/section/(id|title)
/book/section/title	
/book/section/section/title	/book/section/section/(title)
/book/section/section/figure/title	/book/section/section/figure/(title)

QST denotes whether the user query is simple path query, recursive query or a branch query.

Handling Simple Path Query. As a first step, it checks whether it is structural query or a content and structure query. If it is a structural query, then it passes the query to the Plan Generator Module. And, if it is content and structure query(CAS), then the rewriter module rewrites the query by decomposing it into two subqueries. The first is a structural subquery and the second is a content query. This rewritten query is passed to the Plan Generator module to generate the query evaluation plan.

Handling Recursive Query. Rewriter Module handles the recursive queries by invoking a method called rewriteADQuery (Q) (explained in the next section). It then passes the rewritten query into Plan Generator Module.

Handling Branch Query. This module handles the branch query by rewriting it into multiple simple path queries with only parent-child edges. Each subquery is evaluated separately. And, finally, the results of each subquery are merged to obtain the required result.

3.2.3. Plan Generator

This module takes the input from the previous phase and generates the query evaluation plan to execute the query. This module consults the index engine to obtain the required result for the query.

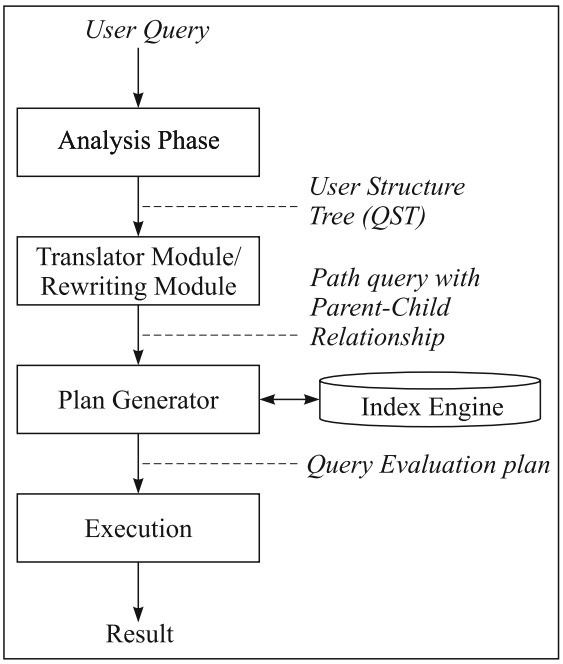

Figure 6. Phases of qurery processing engine of the proposed approach.

3.2.4. Execution

This is the last phase of the query evaluation engine. It executes the query evaluation plan generated by the plan generator. And, the results are returned to the user.

The formal algorithm for the query evaluation is presented in Section 4. The next section elaborates how the index engine is processing the queries specified in the query model of the proposed system.

3.3. Formal Algorithm to Processing the Queries

This section explains the formal algorithm and how different types of queries specified in the query model are processed. The main algorithm is processQuery (Q), it receives the input from the user and gives to the analysis phase. It uses a method called analyseQuery (Q). This method analyses the structure of the query and returns the type of a query structure tree (QST) to a variable with a value in the set{"S","A","B"}, where "S" denotes the simple path query, "A" denotes recursive query and "B" denotes a branch query. If the input query, "Q" contains only Parent-Child (P-C) edges (/), then it returns the type as "S". And, if "Q" contains at least one Ancestor-descendant (A-D) edge (//), the status is returned as "A" to Qtype. Finally, if

Algorithm 2. processQuery (I, Q).

```
Input: Index I, Query Q
Output: All nodes id's that satisfy the query Q.
Begin
Qtype = analyseQuery (Q);
   If Qtype == 'S' then
         processPCQuery (Q);
     Else if Qtype == 'A' then
         processADQuery (Q);
     Else if Qtype == 'B' then
         processBranchQuery (Q);
   End if
End
```

"Q" is a branch query, it returns "B" to Q type. After the analysis phase, the main algorithm returns the QST to the next phase, which in turn calls the required sub algorithm to execute the diverse types of queries.

The processing of each type of queries are explained in the next section.

3.3.1. Processing of Path Query

The proposed system considers two types of queries under the path query. The first is simple path query with only parent-child (P-C) edges. The second one is a recursive query with at least one (A-D) edge. In both cases, it considers the query with or without value predicates. The queries with value predicates are called content and structural (CAS) queries. Simple path query without value predicate is basically a structural query. The proposed approach makes use of p_index to process the path query.

Algorithm to process simple path query.

The method *rewrite(Q)*, rewrites the CAS query by decomposing it into structural part and content part. It stores the result in an array called subQ, where subQ[0] is the structural query and subQ[1] is the content part of the CAS query.

The method *processQ (Q)*, processes *Q* by consulting p_index. If p_index has a matching path stored in it, it returns the corresponding records as the resultSet.

The method *searchResultset(Q, resultSet)* searches the content part from the matched path's posting list which is stored as an extension and implemented as a B+ tree in the p_index.

Algorithm 3. processPCQuery(I, Q).

```
Input: Index I, Query Q
Output: All nodes id's that satisfy the query Q.
Begin
   If (Q is CASQuery) then
        subQ = rewrite(Q);
        resultSet = processQ(subQ[0]);
        return (searchResultset(subQ[1], resultSet));
     Else
        resultSet = processQ(Q);
        return resultSet;
   End if
End
```

Here, subQ is an array to store the subqueries returned by rewrite(Q) is stored.

For example, consider the query "Q1: /book/section/section/title". This is a structural query with structural constraint "/book/section/title". If p_index has a structural match for the given query, then it returns the record values of the element as a query result. In this case, it returns {(title, "Web Analysis", 0.2.2.0), (title, "Web Data and the Two Cultures", 0.2.3.0)} as the query result.

Now, consider the CAS query "Q2: /book [title = "XML"]. It is a CAS query. The translator/Rewriting module rewrites the query Q2 into subqueries Q2a = "/book/title" and Q2b = "XML", where "Q2a" denotes the structural part and "Q2b" denotes the content part.

Algorithm to process recursive query.

Algorithm 4. processADQuery (I,Q).

```
Input: Index I, Query Q
Output: All nodes id's that satisfy the query Q.
Begin
   newQ = rewriteADQuery(Q);
   processPCQuery(newQ);
End
```

The method *rewriteADQuery(Q)*, rewrites the recursive query into simple path query with only P-C edges. For this, as a first step it replaces the A-D edge (//) with P-C edges as (/#). Then, in the next step, it replaces each query node which starts with # into a regular expression wildcard character (.)*t, where t denotes the query node. And, as the last step, it rebuilds the query into a new query with only parent-child edges in it.

In general, if the query is *t1//t2/t3,* then, as a first step, it is rewritten as *t1/#t2/t3*. Next, the query node t2 is replaced as *(.)*t2*. And, as a last step, it rebuilds the new query as *t1/(.)*t2/t3*. Now, this new query is processed in the same way the P-C query is processed.

For example, consider the query "//section" which retrieves all the section elements of the XML documents. To process this query, the query is rewritten as Q_i = */#section*. The result query Q_i is again rewritten as Q_{ii}: *(.)*/section*.

The term *(.)*/section* are searched in the path index. All the entries matching the term are retrieved and merged to obtain the result. Once the search key is matched, the associated posting list is searched for the required result. The next section elaborates the algorithm for processing branch query.

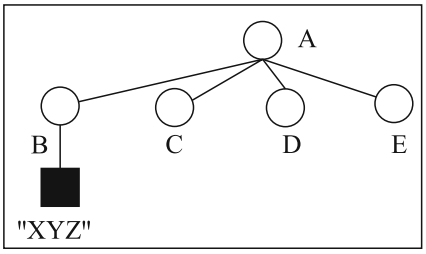

Figure 7. XML data tree.

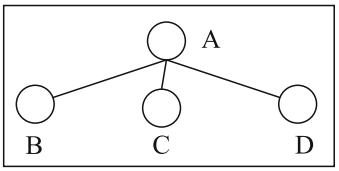

Figure 8. Query tree of /A[B]/[C]/[D].

3.3.2. Processing of Branch Query

The proposed approach processes the branch queries under two scenarios.

Scenario 1: all the queries nodes are at the same level, and

Scenario 2: queried nodes may lie at different levels.

For example, to illustrate this, consider the XML data tree in Figure 7, and the query tree for the branch query "/A[/B][/C][/D]" in Figure 8. It is observed that, here all the queries nodes are terminal sibling nodes and are at the same level.

The first scenario checks whether all the queries nodes are terminal, and they lie at the same level or not. The method *checkSiblinglevel(Q)* returns the value *True*, if it is at the same level. And then it invokes *rewriteCombine(Q)* method. This method rebuilds the query tree by combining all queried terminal sibling nodes in the query into a single path expression. In this case, it rebuilds the query tree as *"/A(B|C|D)"*. In conventional approaches, the branch queries are processed by decomposing them into multiple simple path queries. And each sub queries are evaluated and then merged to obtain the result.

The advantage of the proposed approach is that, if all the queried nodes are terminal and lie at the same level, the query need not be decomposed into multiple queries. The combined path expression can be searched for a structural match in the pc_index. Hence it reduces the number of expensive join operation compared to other existing path index approaches.

To illustrate the second scenario is processed in existing approaches and the proposed approach considers an example query like *"/book[/title]//author[/first name][/last name]"* where the queried nodes *title*, *first name* and *last name* lie at different levels as shown in Figure 9. In conventional approaches, this query is decomposed into multiple simple path queries as:

a) */book/title*,

b) */book//author/first name* and

c) */book//author/last name*.

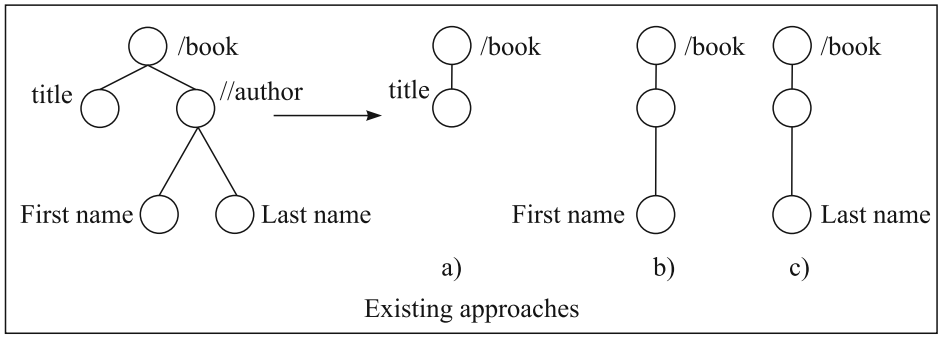

Figure 9. Processing of branch query in scenario2by existing approaches.

These three sub queries are processed separately. And, the results of each sub query are merged together to obtain the result. So, it takes three joins to process this branch query, as shown. Each of these sub-queries can be a path query or a branch query. And, each sub query is evaluated separately to obtain the intermediate results. These intermediate results are merged together to obtain the final result.

Whereas, the proposed approach decomposes the above query as (see Table 1):

a) */book/title* and

b) */book//author[/first name][/last name]*.

It can be observed that the number of decompositions is less in comparison with the existing approaches. This is due to the fact that the proposed approach has the combined path in the pc_index as a search key. Hence, even though, the proposed approach also decomposes the branch query into multiple subqueries, it guarantees that the number of expensive join operations is less in comparison with the existing approaches.

Algorithm 5. processBranchQuery (I, Q).

```
Input: Index I, Query Q
Output: All nodes id's that satisfy the query Q.
Begin
   Flag = checkSiblinglevel(Q);
   If (flag) then
         newQ = rewriteCombine(Q)
         processPCQuery(Q);
   Else
         subQ = rewriteCombine(Q);
         for each q in subQ do processPCQuery(Q);
         End for
   End if
End
```

The method *rewriteCombineQ(Q)* combines all the query nodes which lie at the same level into single path expression. And, the new rewritten path expression is searched in the pc_index to process the branch query.

4. Experiments and Result Analysis

The experiments are conducted to evaluate the performance of the proposed approach on a Core i7–3610QM processor with 3GB of maximum available memory for the Java Runtime Environment. The performance of the proposed algorithms is evaluated considering two metrics. The first metric is the average execution time for the different types of queries. The second metric is the number of scanned elements returned as an answer to the query. This metric measures the efficiency of the index being constructed as part of the proposed scheme. To evaluate the structural queries, the queries with Parent-Child (P-C), Ancestor-Descendant (A-D) queries and the combination of P-C and A-D queries are considered. To evaluate the value part, queries with simple linear path query with value predicate at the end, at any position and A-D queries with value predicate at the end and at any position is considered. The experiments are also conducted by varying the path length of full match query and partial match queries. The query evaluation time with an existing approach [12], [13] is compared with the proposed approach and results are shown in subsection 4.4. Another set of experiments were conducted to measure the time for constructing the index. The data sets chosen for performing the experiments are shown in Table 5.

The proposed system focuses on three types of queries, mainly simple path queries, recursive queries, and branch queries.

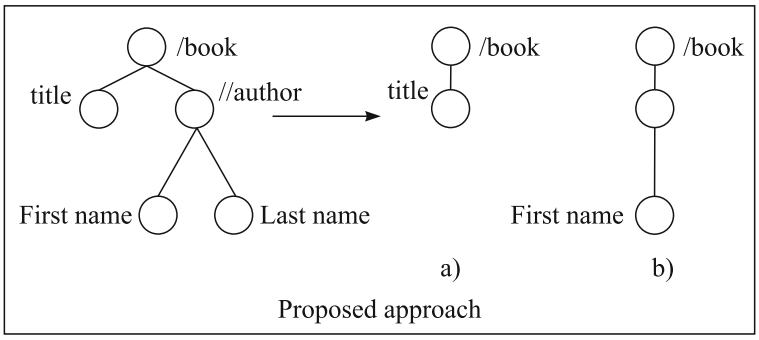

Figure 10. Processing of branch queries in scenario 2 by proposed approach.

Table 5. Characteristics of the dataset.

Parameter	DBLP	Shakespeare	SigmodRecord
#Nodes	3332131	6636	11527
Depth	6	12	6
#Fan-out	22	48	4

Table 6 lists all the queries considered for the experimentation in this paper.

Table 6. Listing of queries.

Listing of queries tested by the proposed approach
Q1. /play/title Q2. /play/act/scene/speech Q3. /play/act **Q4. /play/act/scene**
Q5. //title Q6. //personae Q7. //scene Q8. /play//speech Q9. /play//pgroup Q10. /play//stagedir Q11. //line Q12.//speech//speaker
Q13. //[Speaker = BERNARDO] Q14. //Speech[Speaker = BERNARDO] Q15. //Scene//[Line = Heaven] Q16. //Act//Speech//[Line = Heaven] Q17. /Play//Act//Speech//[Line = Heaven]
QD1. /dblp/article [author = Frank Manola]/title **QD2. /dblp/article [editor = Paul R. McJones]/title** QD3. /dblp/article [/editor = Paul R. McJones][/journal = Digital System Research Center Report][/year]
QD4. /dblp/inproceedings[/author = Tor Helleseth] [/title][/sub] QD5. /dblp/article [/editor=Paul R.McJones][/journal = Digital System Research

The content part is evaluated by considering the queries with value predicate at the end or at any position in both simple path query and recursive queries. The first set of experiments is performed on the metric- number of elements retrieved, for all three types of queries.

4.1. Simple Path Query

The queries Q1 – Q4 listed in Table 6 are simple path queries. These queries are simple structural queries without any value predicate. The structural part is evaluated by varying length of the path. The query Q1 is simple path query which specifies the root-to-leaf path. The last node in the query term is the leaf node. The number of elements retrieved is the number of leaf nodes which satisfies the search criteria. In this case, only one element is retrieved. The Queries Q2 and Q3 are simple path queries, where the last component of the query is not a leaf node. For example, Q3- /play/act, the last component of the query is act, which is not a leaf node. In this case, the query returns all the descendants of the node act as the resultant nodes.

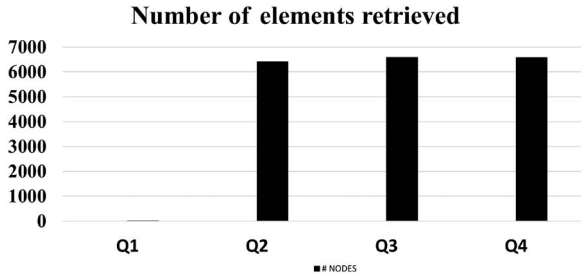

Figure 11. Number of Nodes retrieved for the Simple Path Query Q1 – Q4.

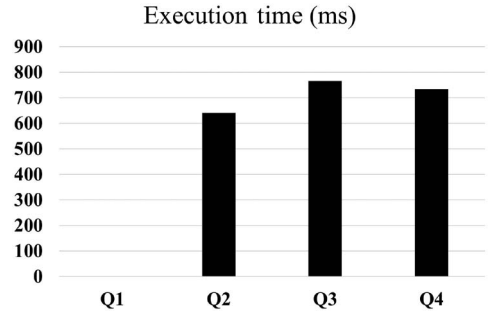

Figure 12. Time taken in millisec to process the simple path query Q1 – Q4.

Figure 12 shows the execution time in millisec for processing the simple path queries Q1 to Q4. It is noted that the query Q1 takes 0 millisec to retrieve the result since it is a root-to-leaf path query. The entire structural path is stored in the index and it takes a single lookup to retrieve the result. And, for the queries Q2 to Q4, the search key is path or substring of the paths, listed in the p_index. If any such match found, the associated nodes from the posting list are retrieved as the resultant nodes.

4.2. Recursive Query

The experimentation recursive query is performed on the Shakespeare's play data set. The queries Q5 – Q12 listed in Table 6 are considered for the experiment. And the graph in Figure 13 and Figure 14 shows the number of elements retrieved and the time taken to process the queries respectively.

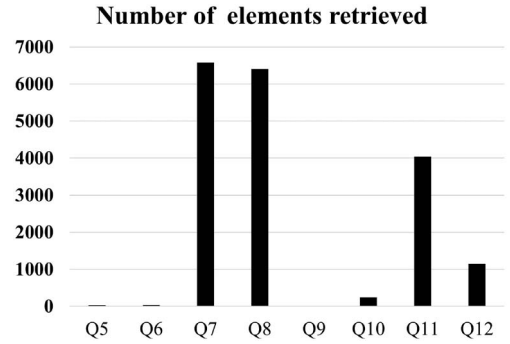

Figure 13. Number of elements retrieved for the recursive queries Q5 – Q12.

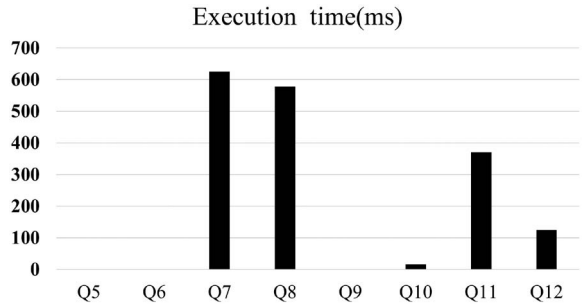

Figure 14. Time taken in millisec to process the recursive queries Q5 – Q12.

It is observed from the graph that the partial match query with // axes nearer to the leaf node or leaf nodes takes less time to retrieve the result. And, if the axis specifier (//) are some internal nodes which are far from the leaf nodes, it takes more time to retrieve the result. For example, queries Q5, Q6, Q9 take less time compared to Q7 and Q8.

4.3. Branch Query

The branch query is performed on the Shakespeare's play data set. The graph in Figure 11(a) and Figure 11(b) shows the number of elements retrieved and the time taken to process the branch query using the proposed index structure.

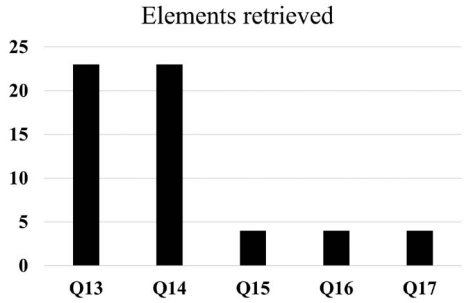

Figure 15. Number of elements retrieved for the branch queries Q13 – Q17.

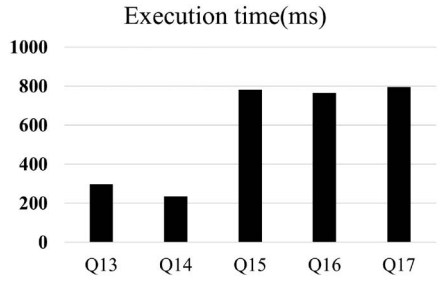

Figure 16. Time taken to process branch queries Q13 – Q17.

4.4. Comparison of Query Performance with the Existing System

The proposed system is compared with some existing systems also. The queries QD1 – QD5 are used to compare the performance of the proposed approach with standard indexing approach and OXDP [12]. The queries are taken from the standard bench mark data set DBLP. Standard indexing approach uses the node elements in the XML documents as the search key.

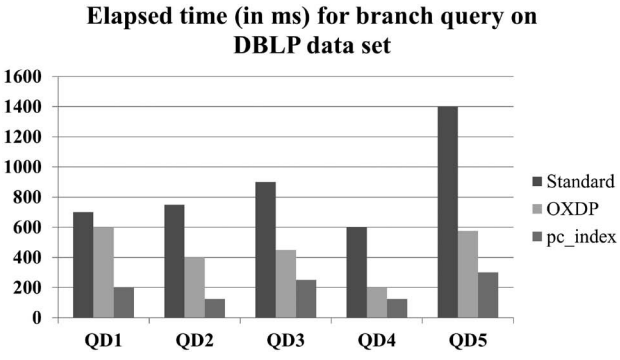

Figure 17. Time taken to process branch queries Q13 – Q17.

The performance of pc_index is compared with the standard value index [10] and the value index proposed in OXDP [12]. For example, the pc_index takes 300ms to retrieve one answer of QD5, the standard index takes almost 1400 ms to retrieve the answer for the same query and OXDP takes almost 575 ms to query the data. This is so because the standard value index is a node based index. And, as the length of the path increases, it needs to scan nodes compared to OXDP or pc_index index. The results in Figure 13 clearly show that pc_index took less time to evaluate the queries.

5. Related Work

There are many indexing approaches of XML documents proposed for supporting the evaluation of XML queries. They are mainly divided into three categories such as node indexing, path indexing and sequence indexing. However, most of the work is focused on processing the structural part in twig queries. Node indexing [11], [14], [15] approach indexes the XML elements and values. For evaluating they must look up all elements and values. This will require a up, since data and elements are indexed separately. The intermediate results are merged to get the final results. For example, for the query //book/title, the indexing approach specified in [11] first looks for all book and title elements in the document and only the related are retrieved. But it can also retrieve the title elements which are not under book, like /movie/title. The problem with this approach is that it provides huge intermediate results which may not be useful for the final result. The indexing approach specified in [16], proposes holistic twig join algorithms to solve the above problem present in [11]. They used a chain of linked stacks to reduce the huge intermediate results. The XISS index in [14] uses B+ tree as an indexing mechanism where they used a node index approach on B+ tree. Here since the basic unit indexed is a node, the query is processed by decomposing it into nodes and these nodes are searched in the B+ tree, for each subpart and finally the intermediate results are joined. The index [17] proposed a tree-structured index XR-Tree with labeled nodes in the XML documents is stored as an extended B+ tree index. These indices are not suitable for content search queries. They mainly focus on the ancestor-descendant or parent-child relationship between nodes. Sequence indexing [18], [19] the XML documents is converted into a sequence representation. The query is also converted into sequence representation. The results are retrieved using subsequence matching. The advantage of using this approach is that twig queries can be answered without merging partial results. However, the disadvantage of subsequence matching is that they require multiple lookups in the index to solve simple path expressions. In [20], authors propose XMIS approach for storing XML documents. They use virtualized inner structure and path index for this purpose. Path indexing approaches [1], [4], [6], [21] indexes XML paths. They index the structural information and value information separately as structure index and value index. This may require expensive join operation or multiple lookups of the indexes for answering path queries. This paper focuses on the construction of an index which consists of two indices, namely, path and path combined index which efficiently answer XML path queries and twig queries with or without value predicates with less expensive joins and in single lookups.

6. Conclusion

A query processing approach using an indexing mechanism to process content and structure queries is proposed in this study. The proposed indexing approach combined the terminal siblings lying at the same level into one combined path. This combined path is stored as a search key in the path combined index. Due to this effort, the branch queries are efficiently evaluated with less number of expensive joins and lookups compared to existing conventional node based or path based approaches. More comparisons taking consideration of different parameters in terms of quantitative terms will be performed as future work.

References

[1] R. Goldman and J. Widom, "Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases", *Proc. Int. Conf. Very Large Data Bases*, 1997, pp. 436–445.

[2] S.-C. Haw and C.-S. Lee, "Extending Path Summary and Region Encoding for Efficient Structural Query Processing in Native XML Databases", *J. Syst. Softw.*, vol. 82, no. 6, pp. 1025–1035, 2009.

[3] Z. Chen *et al.*, "Index Structures for Matching XML Twigs using Relational Query Processors", *Data Knowl. Eng.*, vol. 60, no. 2, pp. 283–302, 2007.

[4] R. Kaushik *et al.*, "Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured Data", *Proc. 2002 Int. Conf. Data Eng.*, pp. 129–140, 2002.

[5] B. F. Cooper *et al.*, "A Fast Index for Semistructured Data", *Proc. Int. Conf. Very Large Data Bases*, vol. 1, pp. 341–350, 2001.

[6] C.-W. Chung, "APEX: An Adaptive Path Index for XML Data", *Proc. ACM SIGMOD Int. Conf. Manag. Data*, pp. 121–132, 2002.

[7] W3c, "XML Path Language (XPath)", *Engineering*, pp. 297–318, 2010.

[8] D. Chamberlin, "XQuery: An XML Query Language", *IBM Syst. J.*, pp. 191, 2002.

[9] J. Lu *et al.*, "From Region Encoding to Extended Dewey: on Efficient Processing of XML Twig Pattern Matching", *31St Int. Conf. ...*, pp. 193–204, 2005.

[10] G. Dhanalekshmi and A. Krishna, "LPLX-Lexicographic-Based Persistent Labelling Scheme of XML Documents for Dynamic Update", *Int. J. Web Sci.*, vol. 2, no. 4, pp. 237–257, 2014.

[11] S. Al-Khalifa *et al.*, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching", *Proc. Int. Conf. Data Eng.*, pp. 141–152, 2002.

[12] N. S. Alghamdi *et al.*, "Semantic-Based Structural and Content Indexing for the Efficient Retrieval of Queries over Large XML Data Repositories", *Futur. Gener. Comput. Syst.*, vol. 37, no. July, pp. 212–231, 2014.

[13] N. S. Alghamdi *et al.*, "Semantic-Based Construction of Content and Structure XML Index", in *Proceedings of the Twenty-Fourth Australasian Database Conference*, vol. 137, 2013, pp. 61–70.

[14] Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions", *Vldb*, pp. 361–370, 2001.

[15] I. Tatarinov *et al.*, "Storing and Querying Ordered XML using a Relational Database System", *ACM SIGMOD Int. Conf. Manag. Data, SIGMOD'02*, 2002, pp. 204–215.

[16] N. Bruno *et al.*, "Holistic Twig Joins: Optimal XML Pattern Matching", *Proc. 2002 ACM SIGMOD Int. Conf. Manag. Data*, 2002, vol. 2, pp. 310–321.

[17] H. Jiang *et al.*, "XR-Tree: Indexing XML Data for efficient Structural Joins", in *Data Engineering, 2003. Proceedings. 19th International Conference on*, 2003, pp. 253–264.

[18] H. Wang and X. Meng, "On the Sequencing of Tree Structures for XML Indexing", in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, 2005, pp. 372–383.

[19] P. Rao and B. Moon, "PRIX: Indexing And Querying XML Using Prüfer Sequences", *ICDE'04 Proc. 20th Int. Conf. Data Eng.*, 2004, pp. 288–299.

[20] C. Mathis *et al.*, "XML Indexing and Storage: Fulfilling the Wish List", *Comput. Sci. – Res. Dev.*, vol. 30, no. 1, pp. 51–68, 2015.

[21] R. Kaushik *et al.*, "Covering Indexes for Branching Path Queries", *Proc. ACM SIGMOD Int. Conf. Manag. data – SIGMOD'02*, 2002, pp. 133.

Received: February 2017
Revised: August 2017
Accepted: September 2017

Contact addresses:
Dhanalekshmi Gopinathan
Department of Computer Science
Jaypee Institute of Information Technology
Noida, India
e-mail: dhanalekshmi.g@jiit.ac.in

Krishna Asawa
Department of Computer Science
Jaypee Institute of Information Technology
Noida, India
e-mail: krishna.asawa@jiit.ac.in

Dhanalekshmi Gopinathan received her M.Tech degree in computer science and engineering from National Institute of Technology, Calicut, India, in 2002. She is currently an Assistant Professor in the Department of Computer Science and Engineering at Jaypee Institute of Information Technology, Noida, India. She is currently pursuing her PhD degree from JIIT, Noida. Her research interests include databases, information retrieval, compiler design, and artificial intelligence.

Krishna Asawa is working with Jaypee Institute of Information Technology, Noida, India as a professor. She was awarded Doctor of Philosophy (CSE) in 2002 from Banasthali Vidyapeeth University, India. Her areas of interest and expertise are Soft Computing and its Applications, Information Security, Knowledge and Data Engineering. Other than JIIT, her employment associations were also with the National Institute of Technology, Jaipur, India and Banasthali Vidyapith, India.