Efficient Processing of XPath Queries Using Indexes

Sanjay Madria[1], Yan Chen
Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65409, USA
madrias@umr.edu

Kalpdrum Passi[2]
Department of Math & CS
Laurentian University
Sudbury, ON, Canada
kpassi@cs.laurentian.ca

Sourav Bhowmick
School of Computer Engineering
Nanyang Technological University
Singapore 639798
assourav@ntu.edu.sg

Abstract: A number of indexing techniques have been proposed in recent times for optimizing the queries on XML and other semistructured data models. Most of the semistructured models use tree-like structures and query languages (XPath, XQuery, etc) which make use of regular path expressions to optimize the query processing. In this paper, we propose two algorithms called Entry-point algorithm and Two-point Entry algorithms that exploit different types of indices to efficiently process XPath queries. We discuss and compare two approaches namely, Root-first and Bottom-first in implementing the Entry-point algorithm. We present the experimental results of the algorithms using XML benchmark queries and data and compare the results with that of traditional methods of query processing with and without the use of indexes, and ToXin indexing approach. Our algorithms show improved performance results than the traditional methods and Toxin indexing approach.

1. Introduction

Traditional databases are highly structured and their implementations are well understood. The benefits of indexing databases for more efficient query processing include faster access to data with a lower overhead cost in processing time. Regular expressions are powerful mechanisms for specifying matching patterns of strings in queries. They work very well when used against the indexes created in traditional databases. However, for querying semistructured (non-traditional) data using some tree-centric model, *path* expressions or *path* regular expressions are needed. Expressions are required to successfully handle the permutations and variability found in the paths from the root to the leaf nodes.

XML models semi-structured data and is becoming standard for interoperability on the Web. XML's schema is self-describing using Document Type Declarations (DTD). XML models heterogeneity more naturally than relational or object-oriented data [23]. But this presents a problem for querying XML data because standard database indexing techniques used in relational and object-oriented databases do not suffice for XML data due to the irregular nature of the data and because of its tree-centric data model with no schema. Hence, XML query languages have heavy reliance on the use of regular path expressions. Path traversals evaluate the tree-shaped document to generate a collection of subtrees, which can be recursively traversed. XML data models use mainly two types of indexes; value (data) and structure (path) indexes for processing queries.

[1] Partially supported by UM Research Board Grant and Intelligent Systems Center
[2] Partially supported by NSERC grant 232038 and an internal LURF grant.

Several query languages have been proposed to query semistructured data, such as XQuery[5], XML-QL[12], XML-GL[9], Lorel[1], and Quilt[10]. XPath [4] is a language that describes the syntax for addressing path expressions over XML data. Most of these query languages are based on XPath and their efficiency depends on XPath processing. To improve the performance of the query on large XML files several indexing techniques have been proposed [11,14,18,19,21,23]. Most of these techniques relied upon recondite processing techniques, which are hard to deploy in real systems. Moreover, the focus is on building new types of indexes, rather than on processing queries efficiently using those indexes.

In this paper, we propose two algorithms called Entry-point algorithm (EPA) and Two-point Entry algorithm (TPA) to efficiently process XPath queries using the simple indexes such as name index, value index and path index and present performance evaluation of the algorithms. Our intent here is not to propose new indexing techniques for XML data, but to design algorithms for efficient processing of XPath queries using simple indexes and compare them with some existing XPath processing techniques using indexes. In the EPA, we find an entry-point node in the XPath expression that can be used to split the XPath expression. The nodes in the XPath expression that have index (Nindex and/or Vindex) defined on them are possible candidates for entry-point. Such nodes along with the descentdant information determine entry-point nodes. Then, we split the XPath expression at the entry-point node and test for the path condition for the first part and eliminate nodes from DOM (Document Object Model proposed by w3c.org which defines the logical structure of the XML document and the way it is accessed and manipulated [13]) tree that do not satisfy the path condition. Next, we test the remaining part of the XPath expression recursively and eliminate nodes that do not satisfy the path condition. The algorithm is implemented using top-down and bottom-up approaches. We explain these two approaches in Section 3. In the TPA, an entry-point and a comparison-point are found instead of a single entry-point as in EPA. We explain the EPA and the TPA techniques using examples before formally giving the algorithms.

We present experiments to show that XPath queries on large XML data execute faster using different types of simple indexes (i.e., node, value and path) with the two algorithms proposed. Compared to *traditional methods* of querying with or without indexing and ToXin XML indexer, our method performs much better in terms of time taken to process XPath queries over different benchmark XML data and queries. In traditional methods without indices, the queries are implemented by traversing the complete XML DOM tree. In traditional methods that use index, the query implementation does not exploit the index information of the ancestor nodes, i.e. it only exploits name or value index but not the path index. ToXin exploits the parent and children information to capture the path index, whereas our method explicitly uses the complete path index from the root to the given node. Also, our method exploits more than one index at multiple levels whereas ToXin exploits path index at the level of parent and child relationship and does not exploit indexes at multiple levels. Note that Toxin index has not been used in our algorithm to show that simple indexes with efficient XPath query processing algorithms can achieve better performance.

The rest of the paper is organized as follows. In subsection 1.1, we review the existing XML indexing work. In Section 2, different types of indexes and examples are given. In Section 3, Entry-point

algorithm and cost analysis is provided. In Section 4 we discuss the Multi-point method and propose Two-point Entry algorithm, which is a special case of Multi-point method. We also discuss the cost analysis of Two-point Entry algorithm. Experiments and observations are given in Section 5. Finally, we conclude the paper in Section 6.

1.1 Related Work

Semistructured data such as XML do not confirm to a rigid, predefined schema and have irregular structure. Indexing techniques in relational or object-oriented databases depend on a fixed schema based on a known, strongly typed class hierarchy. Therefore, such techniques are not directly applicable in XML data. Several indexing schemes have been proposed for semistructured data in [19,20], dataguides [14], 1-indexes, 2-indexes, and T-indexes [21], ToXin[23], XISS[18], index fabric [11], and ViST [25].

In LORE [19] system, four different types of index structures have been proposed, namely, value, text, link, and path indexes. Value index and text index are used to search objects that have specific values; link index and path index provide fast access to parents of an object and all objects reachable via a given labeled path. Lore uses OEM (Object Exchange Model [14]) to store data and OQL (Object Query Language) as its query language.

Dataguides record information on the existing paths in a database. However, they do not provide any information on the parent-child relationships between nodes in the database. As a result, dataguides cannot be used for navigation from an arbitrary node. They can only work on a single regular path expression and require regeneration over the original database. Dataguide can be constructed only from graph-based OEM (Object Exchange Model), which can be of exponential cost.

Another approach of indexing XML data is Index Fabric [11] based on text indexes. Index fabric is based on Patricia tries. Nodes in the Patricia tries are labeled with their *depth*: the character position in the key represented by the node. The keys are formed by encoding data paths using *designators*: special characters or character strings. A unique designator is assigned to each tag that appears in the XML. The designator-encoded XML string is inserted into the layered Patricia trie of the Index Fabric. Raw paths index the hierarchical structure of the XML by encoding root-to-leaf paths as strings. Raw paths do not preserve the sequential ordering of tags in the XML document. Evaluating a query is based on encoding the desired path expression as a key string and matching it with the index fabric. The index fabric is conceptually similar to Dataguide in that it indexes all raw paths starting from the root. Index fabric also defines refined paths, which are specialized paths through the XML that optimize frequently occurring access patterns and support queries that have wild cards, alternates and different constants. A tree-structured query not in the form of refined paths has to use join operations [8]. The paper [25] avoids join operation by proposing a specialized index structure. Most of these methods rely on specialized index or data structures not well supported by DBMS and incur additional cost in terms of complexity and space in maintaining those structures. Some recent methods on query processing that are based on structural summary have been proposed in [15,16,22]. Using those structures, the data graph is summarized with a graph of a smaller size that maintains the structural characteristics of the original data. These proposals are

mainly concerned with creating effective indexes and their maintenance, and are not focused on efficient query processing.

T-indexes [21] are specialized path indexes, which only summarize a limited class of paths. 1-index and 2-index are special cases of T-indexes. A regular expression must be specified by a path template to take advantage of a given T-index. A T-index supports only single regular expressions and updating T-index is an open issue. In our approach we add path information to every node that can trace the parent-child relationship for every node and each XPath sub-query can have its own regular path expression. However, T-index does not support value index.

ToXin [23] has two different types of index structures: the value index and the path index. The path index has two parts: index tree and instance functions, and these functions can be used to trace the parent-child relationship. Their path index contains only parent and children information but in our model, we store the complete path from root to each node. ToXin uses index at a single level while we use multiple indexes at different levels. Also, in our proposal, we consider three different types of indexes to process XPath queries efficiently. Our ideas here are motivated and closely related to ToXin.

Li and Moon [18] proposed the XISS system to index and store XML data. They use two types of indexes related to value and three types of indexes related to the structure. The structure related indexes are element, attribute, and structure indexes. Along with identifier of each node these indexes contain a pair of numbers assigned to the node – extended traversal number and size of each node used by a selection algorithm for computing ancestor-descendant relationship without tree traversal. In XISS, a complex path expression is decomposed into several subsequent simple expressions, which are processed and later joined. This makes indexing scheme much faster. However, the model incurs significant query cost using the numbering scheme proposed since additional reference nodes are being accessed as these nodes are not modeled by the numbering scheme. This scheme also proposes another indexing technique whereas our papers' focus is on efficient processing of XPath queries given different types of indexes.

2. Indexing XML Data

Consider an XML file given in Figure 1 that has information about a bookstore containing say 100,000 books. The DOM tree for the XML fragment is shown in Figure 2. If we need to retrieve all the books, i.e. *topic* nodes with author's last name as "Silberschatz" from the bookstore (a simple query which is often executed in information retrieval system), without using any optimization technique, we need to find all the nodes in the DOM tree with nodes labeled as *topic*. Then for each node *topic*, we need to test *author* *lastname*. In the worst case, after about 100,000 comparisons, we get a couple of books with *author* "Silberschatz" as the output.

By using index on *lastname* node, we do not need to test *author* of each *topic* node. With the index key as "Silberschatz", we can find all the *author* nodes faster (e.g. only three such nodes). The nodes obtained can be checked if they satisfy the query condition. The execution time can be reduced considerably by using the index. This is a "bottom-up" query plan. Such a plan is useful in the case when we have a relatively "small" result set at the bottom, which can be pre-selected. However, consider the

query to find all the books with the title having keyword “system” and the *author lastname* as “Silberschatz” and assume that “Silberschatz” is a famous author’s last name with more than 5,000 books in the store. The query plan in this case could be to first get all the books with the *title* containing the keyword “system” disregarding their authors. If there is a small number of books satisfying the constraint, (e.g., four books with keyword “system” in the *title*), it might be useful to introduce another type of index on the values of some nodes (index on strings). We can now limit our search to nodes returned in the first step (relatively small number of nodes) and apply the second condition of *author lastname* “Silberschatz” on these nodes. The above analogy suggests a strategy for efficient processing of XPath expressions. We find a set of nodes in the XPath tree as the “entry set” such that the remaining search space is minimal. The entry set will depend on the specific query and on the type of XML data.

```
<BOOKSTORE name = “Benny”>
    <BOOK name = “New Books”>
        <CATEGORY type=”Computer Science”>
            <TOPIC title=”Operating System Concepts”>
                <ISBN>1234</ISBN>
                <PUBLISHER>John Wiley</PUBLISHER>
                <AUTHOR>
                    <FIRSTNAME>Abraham</FIRSTNAME>
                    <LASTNAME>Silberschatz</LASTNAME>
                </AUTHOR>
                <AUTHOR>
                    <FIRSTNAME>Abraham</FIRSTNAME>
                    <LASTNAME>Silbershatz</LASTNAME>
                </AUTHOR>
            </TOPIC>
            <TOPIC title=”Database System Concepts”>
                <ISBN>1235</ISBN>
                <PUBLISHER>Prentice Hall</PUBLISHER>
                <AUTHOR>
                    <FIRSTNAME>Abraham</FIRSTNAME>
                    <LASTNAME>Silberschatz</LASTNAME>
                </AUTHOR>
            </TOPIC>
        </CATEGORY>
        <CATEGORY type=”Engineering”>
            <TOPIC title=” Thermodynamic Systems”>
                <ISBN>1236</ISBN>
                <PUBLISHER>Addison Wesley</PUBLISHER>
                <AUTHOR>
                    <FIRSTNAME>Neil</FIRSTNAME>
                    <LASTNAME>Gaiman</LASTNAME>
                </AUTHOR>
            </TOPIC>
            <TOPIC title=”Turbo Mechanics”>
                <ISBN>1237</ISBN>
                <PUBLISHER>Prentice Hall</PUBLISHER>
                <AUTHOR>
                    <FIRSTNAME>Martha</FIRSTNAME>
                    <LASTNAME>Brooks</LASTNAME>
                </AUTHOR>
            </TOPIC>
        </CATEGORY>
    </BOOK>
    <GIFTS>
        <ITEMS>
            <TOPIC title=”Garden Shop”>
                <NAME>Juliska</NAME>
                <PRICE>$59.95</PRICE>
            </TOPIC>
            <TOPIC title=”Bridal”>
                <NAME>Wedding Planner</NAME>
                <PRICE>$74.95</PRICE>
            </TOPIC>
        </ITEMS>
    </GIFTS>
<MUSIC>
    <CATEGORY type=”Pop”>
        <SONG title=”Up!”>
            <ASIN>2345</ASIN>
            <COMPANY>Universal</COMPANY>
            <SINGER>
                <FIRSTNAME>Shania</FIRSTNAME>
                <LASTNAME>Twain</LASTNAME>
            </SINGER>
        </SONG>
        <SONG title=”Come Away with Me”>
            <ASIN>2346</ASIN>
            <COMPANY>Blue Note</COMPANY>
            <SINGER >
                <FIRSTNAME>Norah</FIRSTNAME>
                <LASTNAME>Jones</LASTNAME>
            </SINGER >
        </SONG >
    </CATEGORY>
    <CATEGORY type=”Jazz”>
        <SONG title=”What a Wonderful World”>
            <ASIN >1237</ ASIN >
            <COMPANY >Sony Music Canada Inc.</COMPANY >
            <SINGER >
                <FIRSTNAME>Tony</FIRSTNAME>
                <LASTNAME>Benett</LASTNAME>
            </SINGER >
        </SONG >
    </CATEGORY>
</MUSIC>
<BOOK name = “Used Books”>
    <CATEGORY type=”Computer Science”>
        <TOPIC title=”Applied Operating System Concepts”>
            <ISBN>1238</ISBN>
            <PUBLISHER>McGraw Hill</PUBLISHER>
            <AUTHOR>
                <FIRSTNAME>Abraham</FIRSTNAME>
                <LASTNAME>Silberschatz</LASTNAME>
            </AUTHOR>
        </TOPIC>
        <TOPIC title=”Computer Architecture”>
            <ISBN>1239</ISBN>
            <PUBLISHER>Prentice Hall</PUBLISHER>
            <AUTHOR>
                <FIRSTNAME>Morris</FIRSTNAME>
                <LASTNAME>Mano</LASTNAME>
            </AUTHOR>
        </TOPIC>
    </CATEGORY>
    <CATEGORY type=”fiction”>
        <TOPIC title=” The First Time”>
            <ISBN>1241</ISBN>
            <PUBLISHER>Pocket Books</PUBLISHER>
            <AUTHOR>
                <FIRSTNAME>Jay</FIRSTNAME>
                <LASTNAME>Fielding</LASTNAME>
            </AUTHOR>
        </TOPIC>
    </CATEGORY>
</BOOK>
</BOOKSTORE>
```

Figure 1. An XML Fragment

2.1 Types of Indexes

We now describe three types of indexes that can be built over an XML file. We describe methods and algorithms to process XPath queries that utilize the indexing structures to achieve high performance. The first type of index identifies objects that have specific name values; the next two are used to efficiently retrieve the objects in an XML tree based on values and path nodes. In XML Structures, each node has an

associated ID. We exploit this attribute and establish a relationship between the node ID and storage address to speed our search.

a) Name-index (Nindex)

A name-index represents nodes with their tag names. Using this index, we group nodes that have the same tag name. The Nindex for the incoming tag <BOOK> over the XML fragment in Figure 2 will then be {&2, &5}.

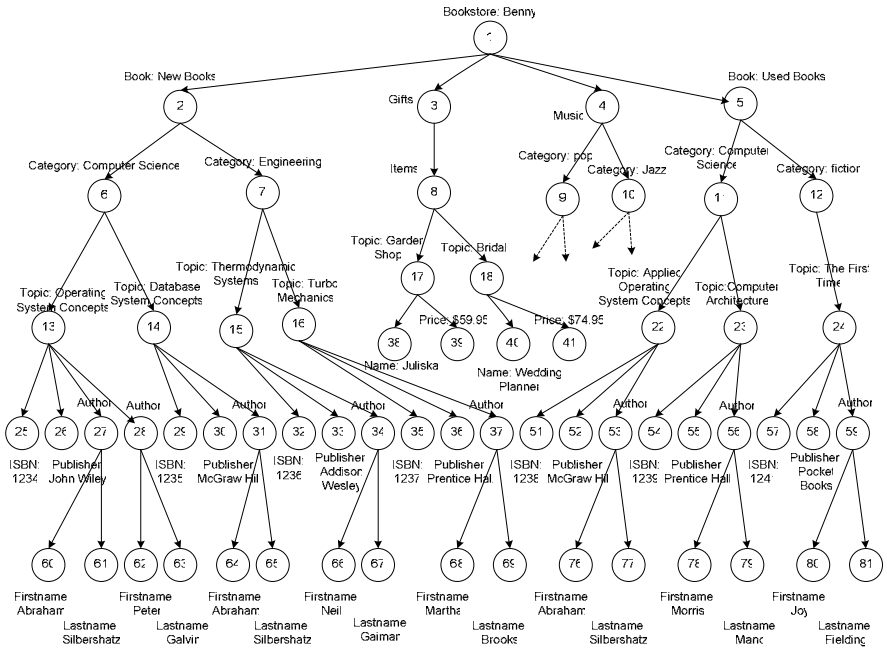

Figure 2. Simple DOM Tree of Benny-bookstore

b) Value-index (Vindex)

A value-index locates nodes with a given value. Since XML has classified different data types, therefore we do not need type coercion [19]. Vindex can be built selectively over basic types, such as numbers, strings or on other types. The value-index for the word “Silberschatz” is {&61, &65, &77} for the word “system” is {&13, &14, &15, &22}. It may not be useful to build an index on every value. For example, the index on the word “shop” seems to be redundant. The administrator or the user can decide which values would be useful as a value-index.

The Nindex and Vindex can also be grouped together, as we need the name or type of nodes to restrict the value-index. We can integrate these two indexes to facilitate and accelerate the query. In the simulation program, we extract the features of Nindex and Vindex to an abstract “type” index, which could be replaced by a specific index in an actual environment.

c) Path-index (Pindex)

A path-index locates nodes with the path from root node. After we get an initial set of nodes while processing an XPath query, we further need to test the nodes if they satisfy the input XPath expression. To make the testing efficient, it is helpful to record the node's path from root. Path-index is the information we attach to each node to record its ancestors starting with the root. This information is also very useful in our algorithm. As in some of the traditional algorithms [14,19], if we do not execute the query from top to bottom, we need to know the ancestors of certain nodes at the middle level (other than the root and leaf nodes). Since we need to build this information on each node, an index will be useful. In Figure 2 the path information of node &14 is {&1, &2, &6}; node &39's path information is {&1, &3, &8, &17}.

Definition 1: Descent Number (DN)

Descent Number is the information attached to every node to record the number of its descents. The descents of a node include all the arcs going out of that node and reaching the leaf nodes. This information is used to select one index node set among several index node sets. The usage of DN will be elaborated in our algorithms. In Figure 2, the DN of node &65 is 0; the DN of node &31 is 2.

3. Entry-point Method

XPath is a language for addressing parts of an XML document. XPath also provides basic constructs for manipulation of strings, numbers and Boolean data. XPath operates on abstract, logical structure of an XML document, rather than on its surface syntax. XPath uses path notations for navigating through the hierarchical structure of an XML document. A query written in any of the query languages such as XQuery[5], XML-QL[12], XML-GL[9], Lorel[1], and Quilt[10] is easily transformed to an XPath expression. If we need to retrieve a relatively small part (data) from the large XML file under certain constraints expressed using XPath, it will be expensive to compare each node with given search conditions.

Assuming that various types of indexes, such as Nindex, Vindex and Pindex defined in Section 2 have been created, we give two techniques to process and optimize the XPath expression generated from a given query. In the first technique, we find an entry-point node in the XPath expression that can be used to split the XPath expression. The nodes in the XPath expression that have index (Nindex and/or Vindex) defined on them are possible candidates for entry-point. A node that minimizes the search space in the XPath tree is selected as the entry-point from the set of indexed nodes. This is determined by the descent of the nodes. The node set in the DOM tree representing the entry-point that satisfies the node test forms the entry set. The entry set is tested for the path condition of the first part of the split XPath expression using Pindex. The entry set that satisfies the path condition represents root nodes of a set of subtrees forming the new search space. The second part of the split XPath expression is now processed recursively on the new search space. The algorithm can be implemented top-down using root-first approach or bottom-up using bottom-first approach. In the root-first approach, the entry set is tested for path condition of the first part of the split XPath expression from root node to the entry-point nodes. The nodes in the entry set that do not satisfy the path condition are eliminated. The algorithm is recursively applied on the new entry set as the

root of their subtrees for the second part of the split XPath expression. In the bottom-first approach, the entry set is tested for path condition of the second part of the split XPath expression from leaves to the entry-point nodes. The nodes in the entry set that do not satisfy the path condition are eliminated. The algorithm is recursively applied on the new entry set as the leaves for the first part of the split XPath expression. Note that XPath returns the nodes in document order. We explain the technique using the example in Figure 1, before formally giving the algorithm. Suppose we are given the following query:

```
Select BOOKSTORE//TOPIC
where TOPIC.title contains "system"
  and //AUTHOR/[LASTNAME = "Silberschatz"]
  and BOOKSTORE.name = "Benny"
```

The above query is transformed to the following XPath expression:

```
BOOKSTORE [@name  = "Benny"]//TOPIC[@title = "%system%"]/AUTHOR/
      LASTNAME[. = "Silberschatz"]
```

Given several types of indexes we can retrieve some specific node sets. For example, we can use Nindex to get all *topic* nodes or *lastname* nodes. There can be two ways to accelerate the query execution. In the first case, we can get all the books that have the keyword "system" and then test the condition on each one of them if the *author lastname* is "Silberschatz". In the second case, we can get all authors with last name as "Silberschatz", and then test the ancestor nodes if book title contains the keyword "system". In the root-first strategy, we evaluate the former part of XPath expression first, that is,

```
BOOKSTORE [@name  = "Benny"]//TOPIC[@title = "%system%"]
```

Then, we test each author's *lastname* node on the *topic* nodes returned from first part of XPath expression. This test forms the latter part of XPath expression:

```
/AUTHOR/ LASTNAME[. = "Silberschatz"].
```

In terms of the XML DOM tree, we find an entry-point node from the indexed nodes in the given XPath expression that has the minimum Decent Number. In the above example, *topic* node is the entry-point, which splits the given XPath expression into two parts as shown above. The *topic* nodes that satisfy the node test, i.e. "system" is a keyword in the *title*, forms the entry set. Nodes in the entry set are then tested for path information of the first part of the split XPath expression using Pindex. Nodes in the entry set that do not satisfy the path condition are eliminated. Next, we consider the remaining entry-point nodes as the root nodes of a set of subtrees and eliminate nodes recursively from the subtrees to get the resulting nodes. The *topic* nodes in the entry set that satisfy the path condition of the second part of the split XPath expression, i.e. *author lastname* nodes with "Silberschatz" are selected. The bottom-first strategy is similar, except that the entry-point nodes, i.e. *topic* nodes, will constitute the result node set. To decide which strategy is more efficient, we need to find out the cost of searching the remaining set of nodes that satisfy the path condition. This will depend on the data in an XML file. Suppose we have n1 number of *topic* nodes having keyword "system" and n2 *lastname* nodes having the value "Silberschatz", then the time for the first strategy (i.e. selecting *topic* nodes first) will be c+O(n1)+cost of searching remaining set of nodes. Similarly, time for the second strategy (i.e. selecting *lastname* nodes first) will be c+O(n2)+ cost of searching remaining set of nodes (c being a constant). The cost of searching remaining set of nodes in the

first strategy could be estimated by counting the nodes in the remaining subtrees after the first step. This is implemented by adding the descent number (DN) to every node of the DOM tree.

3.1 Entry-point Algorithm

We now present two versions of Entry-point Algorithm using the root-first and bottom-first approaches.

INPUT: XPath expression **root/X_1/X_2/…/X_i/…/X_m, OUTPUT: $\Re$** denotes the result node set

STEP 1: IF input XPath consists of only the last level node /X_m
THEN
return all nodes x_m of type X_m in $\Re$

STEP 2: FOR each X_i, i=1,…,m
IF X_i is indexed (Nindex or Vindex)
THEN
Add X_i to set I, the set of indexed nodes;

Let | I | = p, where | I | denotes the number of nodes in I
For i=1,…,p
{
Let x_{i1}, x_{i2}, …, x_{ik} be k node instances of $X_i \in$ I
Let DN_i denotes the total descent of all instances of X_i and d_{ij} be the descent of node x_{ij}

$$DN_i = \sum_{j=1,\ldots,k} d_{ij}$$

}

STEP 3: Let DN_ℓ = Min {DN_1, DN_2, …, DN_p} and X_ℓ denote the node corresponding to DN_ℓ
Then X_ℓ is the entry-point
Add $x_{\ell 1}$, $x_{\ell 2}$, …, $x_{\ell k}$ to set E, where E denotes the entry set
Split the XPath into root/X_1/X_2/…/$X_{\ell-1}$ and $X_{\ell+1}$/…/X_m by the entry-point X_ℓ;

STEP 4: FOR j = 1, …, k
IF the Pindex of node $x_{\ell j}$ does not match the path root/X_1/X_2/…/$X_{\ell-1}$/X_ℓ
THEN
E = E – {$x_{\ell j}$}; (*This is a delete operation*)

Let | E | = q, where q ≤ k
FOR each node $x_{\ell j} \in$ E, j = 1, …, q
Consider all subtrees with $x_{\ell j}$ as the root nodes;

INPUT = X_ℓ/$X_{\ell+1}$/…/X_m
GO TO STEP 1

Algorithm 1.1 Entry-point Root-first

STEP 1: IF input XPath consists of only the root and the node X_ℓ, where X_ℓ is indexed and has the minimum descent
THEN
return all nodes x_m of type X_m in $\Re$

STEP 2: FOR each X_i, i=1,…,m
IF X_i is indexed (Nindex or Vindex)
THEN
Add X_i to set I, the set of indexed nodes;

Let | I | = p, where | I | denotes the number of nodes in I
For i=1,…,p
{
Let $x_{i1}, x_{i2}, \ldots, x_{ik}$ be k node instances of $X_i \in I$
Let DN_i denotes the total descent of all instances of X_i and d_{ij} be the descent of node x_{ij}

$$DN_i = \sum_{j=1,\ldots,k} d_{ij}$$

}

STEP 3: Let DN_ℓ = Min $\{DN_1, DN_2, \ldots, DN_p\}$ and X_ℓ denotes the node corresponding to DN_ℓ
Then X_ℓ is the entry-point
Add $x_{\ell 1}, x_{\ell 2}, \ldots, x_{\ell k}$ to set E, where E denotes the entry-set
Split the XPath into root/$X_1/X_2/\ldots/X_{\ell-1}$ and $X_{\ell+1}/\ldots/X_m$ by the entry-point X_ℓ;

STEP 4: FOR j = 1, …, k
Find $x_{\ell j}$ // x_m , i.e. all descendant nodes x_m of each $x_{\ell j}$ and add x_m to $\mathfrak{R}$
IF the Pindex of node x_m does not match the path $X_\ell/X_{\ell+1}/\ldots/X_m$
THEN
$\mathfrak{R} = \mathfrak{R} - \{ x_m \}$ and E = E – $\{x_{\ell j}\}$; (*This is a delete operation*)

Let | E | = q, where q ≤ k
FOR each node $x_{\ell j} \in$ E, j = 1, …, q
Consider all subtrees with $x_{\ell j}$ as leaf nodes

INPUT = root/$X_1/X_2/\ldots/X_\ell$
GO TO STEP 1

Algorithm 1.2 Entry-point Bottom-first

Note that due to the recursive algorithm, we use multiple indexes that can enhance the performance. This is illustrated in the comparison cases in Section 6. Next, we illustrate the Entry-point Root-first Algorithm with the help of following example.

Example: XPath expression to be evaluated: Bookstore[@name =”Benny”]/Book/category
[@type=”Computer Science”]/topic[@title=”%system%”]//lastname[. =”Silberschatz”]

Assume that indexes have been built on nodes ‘category’ and ‘topic’.

Step 1: Find the sum of all descent numbers for the nodes ‘category’ and ‘topic’. The descent numbers of nodes are shown in Figure 3.
DN for ‘category’ = 63
DN for ‘topic’ = 42

Step 2: Find the entry-point with minimum descent. The node ‘topic’ becomes the entry point as it has the minimum descent. It is important to note that it is possible that the DN of an entry-level node at a higher level might be smaller than the DN of an entry-level node at a lower level in the DOM tree. This is possible for the case when there might be a large number of entry-level nodes at the lower level as compared to the number of entry-level nodes at a higher level of the DOM tree. The tree obtained after deleting all branches that do not have the node ‘topic’ is given in Figure 4.

Split the XPath expression into Bookstore[@name=”Benny”]/Book/category[@type=”Computer Science”]/ and //lastname[. =”Silberschatz”] by the node topic[@title=”%system%”].

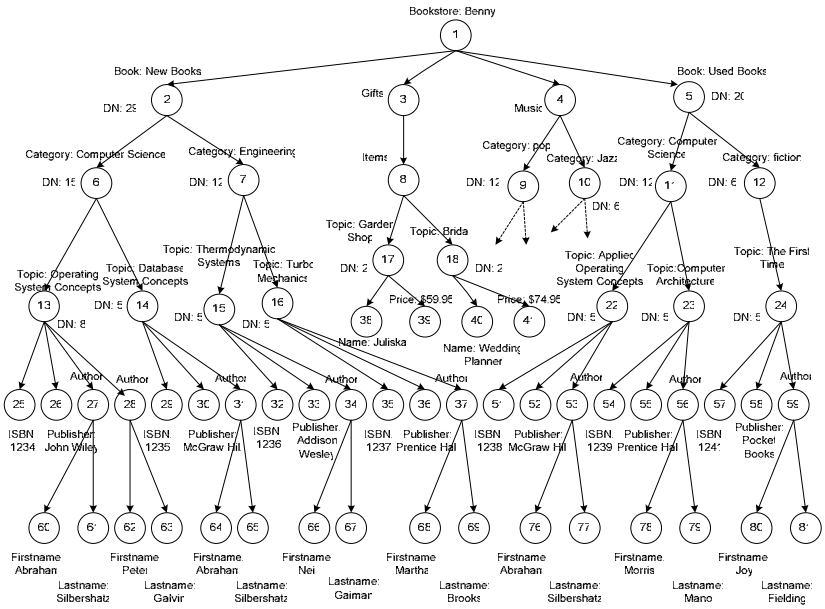

Figure 3: Finding descent numbers of nodes

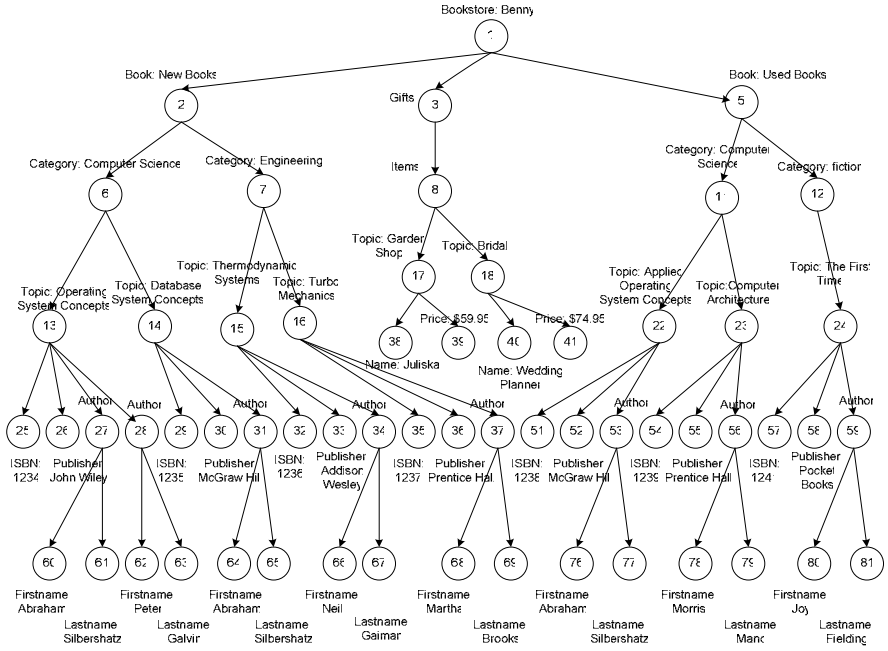

Figure 4: Deleting nodes and branches that do not have ‘topic’ node

Step 3: Test the path Bookstore[@name=”Benny”]/Book/category[@type=”Computer Science”]/topic[@title=”%system%”] on each ‘topic’ node.

We obtain the tree in Figure 5 after deleting the branches that do not satisfy the above path condition.

Step 4: The remaining path is //lastname[. ="Silberschatz"] and cannot be split further. We obtain all the double circled nodes in Figure 6 that satisfy topic[@title="%system%"]//lastname[."Silberschatz"].

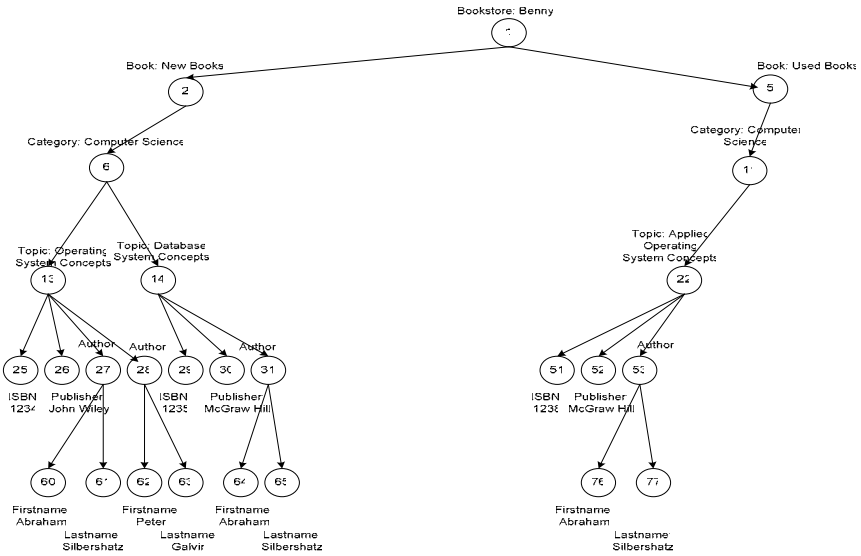

Figure 5: Deleting nodes and branches that do not satisfy the path

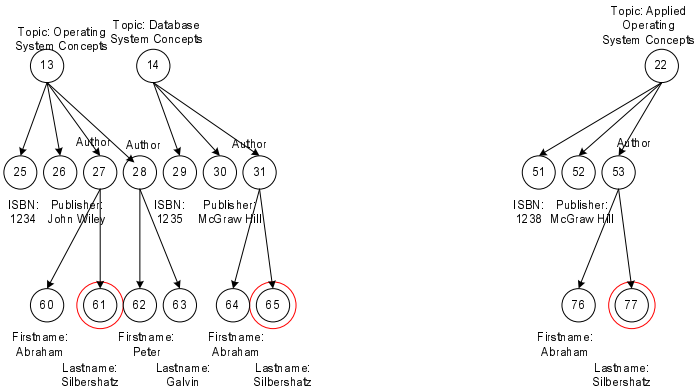

Figure 6: Retrieve nodes satisfying the path condition

It is easy to see that the Entry-point algorithm will work with regular expressions too. We observe that if an XPath expression is of the form $\mathbf{root/X_1/X_2/\ldots/X_i/\ldots/X_m}$ where any expression X_i = *, we can check if the path of the entry-point node x_n matches the given regular path $\mathbf{root/X_1/X_2/*/\ldots/X_{n-1}/X_n}$ in Step 2 of the algorithm by using string matching, structural join or containment techniques [2,8,25]. Using the technique in [25], the query need not be decomposed into sub-queries to improve the performance further

by avoiding join operation, but it needs to maintain a different index structure. The path index technique given in [3] can further improve the performance of containment relationships, as it can efficiently process path expressions with wild cards. Also note that the result returned is in the same order as the XML file. Since the nodes have unique identifiers in the DOM tree construction and the traversal is depth-first, the nodes are returned in XML order.

3.2 Cost Analysis of Entry-point Algorithm

Here, we outline the worst-case cost analysis of Entry-point algorithm 1.1. We assume that the cost of DN is associated with the generation of the DOM tree. Let N be the total number of nodes in the DOM tree.

Step 1: It returns $\mathfrak{R}$, the result node set. There is no computation involved in this step.

Step 2: For each X_i, i=1,…,m

One comparison at each iteration of this loop implies a total of m comparisons

For i=1,…,p

For j = 1,…,k

One addition at each iteration of this loop implies total of $p(k^2+k)/2$ additions

Step 3: Min $\{DN_1, DN_2, \ldots, DN_p\}$

Finding minimum of p numbers will involve p comparisons.

Step 4: For j = 1, …, k

In this step, we match Pindex with XPath string of length ℓ, an O(1) operation as $\ell \ll N$.

That means a total of k matches.

The steps are repeated on q subtrees. Without loss of generality, we can assume that subtrees are of O(N) nodes. The recursion will be at most log m times. Also, note that $p < m$ and $k \leq \log N$.

Thus, the total cost = $\log m\, [m + p(k^2+k)/2 + p + k]$

$\leq \log m\, [m + m(\log^2 N + \log N)/2 + m + \log N]$

$= \log m\, [m/2 \log^2 N + (m+2)/2 \log N + 2m]$

$= O(\log^2 N)$ (since m « N, m can be considered a constant)

For space complexity, notice that a count of up to N nodes in the DOM tree will need log(N) storage space. The index on k instances of p number of x_i nodes will require a space of $pk \leq p.\log(N)$ (since k is bounded by log(N)), where p can be considered a constant as the number of nodes in the XPath expression will be small compared to N. The XPath expression with m nodes will require a constant space as m is small compared to N. Therefore, the total space required will be $m + p.\log(N) + \log(N) = O(\log(N))$ space.

3.3 Correctness of Entry-point Algorithm

We will prove the correctness by induction on the length of XPath expression. When the length of the XPath expression is 1 or 2, i.e. either $/X_1$ or $/X_1/X_2$, respectively, the result nodes are obtained in $\mathfrak{R}$ from Step 1. Consider XPath expression of length 3, i.e. $/X_1/X_2/X_3$. Assume X_2 is indexed. Note that even though X_3 (leaf nodes) may be indexed and will have the minimum descent 0, we do not consider X_3 as the XPath

expression cannot be split using X_3. Since X_2 is the only indexed node in this case, the minimum descent is represented by X_2. X_2 is the entry-point which is used to split the XPath expression into $/X_1/X_2$ and $/X_2/X_3$ in Step 3 of the algorithm. All node instances of X_2 represent the entry-set. The Pindex of each node instance of X_2 is compared with XPath expression $/X_1/X_2$ and the nodes that do not have X_1 as its parent are deleted from the entry-set in Step 4 of the algorithm. Next, consider all subtrees with X_2 nodes in the entry-set as the root nodes and get all the child nodes X_3 in $\mathfrak{R}$. Consider an XPath expression of length m, i.e., $/X_1/X_2/\ldots/X_i/.../X_m$. Assume that Xi is indexed and has the minimum descent among all the indexed nodes in the XPath expression. Then X_i represents the entry-point and the XPath expression is split using X_i into $/X_1/X_2/\ldots/X_{i-1}/X_i$ and $X_i/X_{i+1}/\ldots/X_m$. All the node instances of X_i form the entry-set. The Pindex of all X_i nodes is compared with the XPath expression $/X_1/X_2/\ldots/X_{i-1}/X_i$ and all X_i nodes that do not match the path are deleted from the entry-set. Next all the remaining Xi nodes in the entry-set are represent the root nodes of the subtrees in the new search space. The same steps are applied to the XPath expression $X_i/X_{i+1}/\ldots/X_m$ on the subtrees and the correctness follows from the case where length of XPath expression is less than equal to 4.

4. Multi-point Method

In the Entry-point algorithm, we find an entry-point node in the XPath expression from the nodes that have index defined on them and split the XPath expression at the entry-point. Some of the entry-level nodes are eliminated by comparing their path from the root node to the first part of the split XPath expression. The XPath expression is evaluated recursively using the same technique. The Entry-point algorithm may not perform well under certain conditions. For instance, in the BOOKSTORE database shown in Figure 1, suppose we want to find out books written by authors with last name “Silberschatz” where the title of the book contains the word “system”. The XML file might have hundreds of books having the word “system” in the title and further there might be a large number of books by author “Silberschatz”, but only one of them has the word “system” in its title. The Entry-point algorithm first eliminates all the nodes that do not have the word “system” in its title. Then it eliminates the nodes that do not have “Silberschatz” as the author last name. Similarly in the Entry-point Bottom-first approach, the nodes that do not have “Silberschatz” as the author last name are eliminated first and then the ancestor nodes that do not have the word “system” in its title are eliminated. In this case, due to relatively large number of instances at the two levels, a large number of eliminations are required. We refine the Entry-point algorithm to make it more efficient to handle such cases by selecting two or more entry-level points in the XPath expression and eliminating the nodes based on those entry-level nodes.

In the Multi-point method, we consider nodes at more than one level that have indexes defined on them and eliminate nodes by checking their ancestor information in the path-index. The nodes that do not satisfy the child/ancestor relationship are eliminated in the intermediate step. This step optimizes the Entry-point algorithm by eliminating some more nodes without the comparison of complete path-index. The nodes that satisfy the query condition and that have the minimum number of descendants is available in

step 2 of the Entry-point algorithm. In addition to the nodes having the minimum descent, we consider nodes with second minimum, third minimum, and so on. For example, after step 2 of the Entry-point algorithm, we know the DN (descent number) of nodes at Level A that satisfy say condition A is 2000, DN of nodes at Level B that satisfy condition B is 1000, DN of nodes at Level C that satisfy condition C is 200, DN of nodes at Level D that satisfy condition D is 3000, DN of nodes at Level E that satisfy condition E is 400, assuming that levels are labeled from leaves to the root node. The minimum DN is at Level C and the ordering of the levels in terms of DNs is (C, E, B, A, D). As the ancestor node information is available in path-index, we can filter some nodes at Level C by checking 200 nodes at Level C that have node E as its grandparent. Similarly, we can filter some other nodes at Level C by checking 1000 nodes at Level B that have node C as its parent. The elimination of nodes at Level C will be complete by checking ancestor information of nodes at Level C with nodes at Level D. The elimination of nodes is done by comparing the level having minimum descent with multiple levels. But in practice, comparison of nodes at only two levels may suffice in order to achieve performance gains.

4.1 Two-point Entry Algorithm

Two-point Entry algorithm (TPA) is a special case of Multi-point algorithm (MPA) where we consider only two levels in the XPath expression for comparison and elimination of nodes at an intermediate step. The node that has the minimum descent among the indexed nodes in the XPath expression is labeled as the entry-level node. The node that has the second minimum descent among the indexed nodes in the XPath expression is labeled as the comparison-level node. The comparison-level may be above or below the entry-level. An intermediate step is introduced in the Entry-point algorithm (EPA) to eliminate nodes at the entry-level by comparing nodes at the comparison-level. If the comparison-level is above the entry-level, the nodes at the entry-level are eliminated if the comparison-level nodes are not in the path-index of the entry-level nodes. Similarly, if the comparison-level is below the entry-level, the entry-level nodes are eliminated if they are not in the set of ancestor nodes of the path-index of comparison-level nodes. This extra step in the Two-point Entry algorithm eliminates some entry-level nodes without comparing the complete path-index with its ancestor information. There is a significant performance improvement as will be evident from the experimental results that we discuss in the next section. Next we present the Two-point Entry algorithm.

INPUT: XPath expression **root/X_1/X_2/…/X_i/…/X_m, OUTPUT:** $\mathfrak{R}$ denotes result node set

STEP 1: IF input XPath consists of only the last level node /X_m
THEN
return all nodes x_m of type X_m in $\mathfrak{R}$
STEP 2: FOR each X_i, i=1,…,m
IF X_i is indexed (Nindex or Vindex)
THEN
Add X_i to set I, the set of indexed nodes;

Let | I | = p, where | I | denotes the number of nodes in I

For i=1,…,p
{
Let $x_{i1}, x_{i2}, \ldots, x_{ik}$ be k node instances of $X_i \in I$
Let DN_i denote total descent of all instances of X_i and d_{ij} be the descent of node x_{ij}

$$DN_i = \sum_{j=1,\ldots,k} d_{ij}$$

}

STEP 3: Let DN_ℓ = Min $\{DN_1, DN_2, \ldots, DN_p\}$ and X_ℓ denote the node corresponding to DN_ℓ
and DN_t = Min$\{\{ DN_1, DN_2, \ldots, DN_p\}$ - $DN_\ell \}$ and X_t denote the node corresponding to DN_t
Then X_ℓ is the entry-point and X_t is the comparison-point
Add $x_{\ell 1}, x_{\ell 2}, \ldots, x_{\ell k}$ to set E, where E denotes the entry-set
Add $x_{t1}, x_{t2}, \ldots, x_{tk}$ to set C, where C denotes the comparison-set

STEP 4: FOR each node $x_{\ell j} \in$ E, j=1,…,k
IF (ℓ < t AND Pindex of $x_{\ell j}$ does not contain x_{tj}) OR
(ℓ > t AND Pindex of x_{tj} does not contain $x_{\ell j}$)
THEN
E = E – { $x_{\ell j}$} (*This is a delete operation*)

Let | E | = r, where r ≤ k (as some nodes have been deleted from E)
Split the XPath into root/X_1/X_2/…/$X_{\ell-1}$ and $X_{\ell+1}$/…/X_m by the entry-point X_ℓ;

STEP 5: FOR each node $x_{\ell j} \in$ E, j = 1, …, r
IF the Pindex of node $x_{\ell j}$ does not match the path root/X_1/X_2/…/$X_{\ell-1}$/X_ℓ
THEN
E = E – {$x_{\ell j}$}; (*This is a delete operation*)
Let | E | = q, where q ≤ r
FOR each node $x_{\ell j} \in$ E, j = 1, …, q
Consider all subtrees with $x_{\ell j}$ as the root nodes;

INPUT = X_ℓ/$X_{\ell+1}$/…/X_m
GO TO STEP 1;

Algorithm 2. Two-point Entry

Example: XPath expression to be evaluated: /Bookstore[@name=”Benny”]/Book/category
[@type=”Computer Science”]/topic[@title=”%system%”]//lastname[. =”Silberschatz”]

Assume that the indexes have been built on nodes ‘book’, ‘category’ and ‘topic’.

Step 1: Find the sum of all descent numbers for the nodes ‘book’, ‘category’ and ‘topic’. The descent numbers of nodes are shown in Figure 3.

DN of node book = 49

DN of node category = 63

DN of node topic = 42

Step 2: Find the entry-point with minimum descent. The node ‘topic’ becomes the entry point as it has the minimum descent. The comparison point with second minimum DN is the node ‘book’. The tree obtained after deleting all branches that do not have the node ‘topic’ is given in Figure 4.

Split the XPath expression into Bookstore[@name=”Benny”]/Book/category[@type=”Computer Science”]/ and //lastname[. =”Silberschatz”] by the node topic[@title=”%system%”].

Step 3: Delete 'topic' nodes whose ancestor does not have the nodes 'book'. The tree obtained after this operation is given in Figure 7.

Step 4: Test the path Bookstore[@name="Benny"]/Book/category[@type="Computer Science"]/topic[@title="%system%"] on each 'topic' node.

We obtain the tree in Figure 5 after deleting the branches that do not satisfy the above path condition.

Step 5: The remaining path is //lastname[. ="Silberschatz"] and cannot be split further. We obtain all the nodes in Figure 6 that satisfy topic[@title="%system%"]//lastname[.="Silberschatz"] and are circled in red.

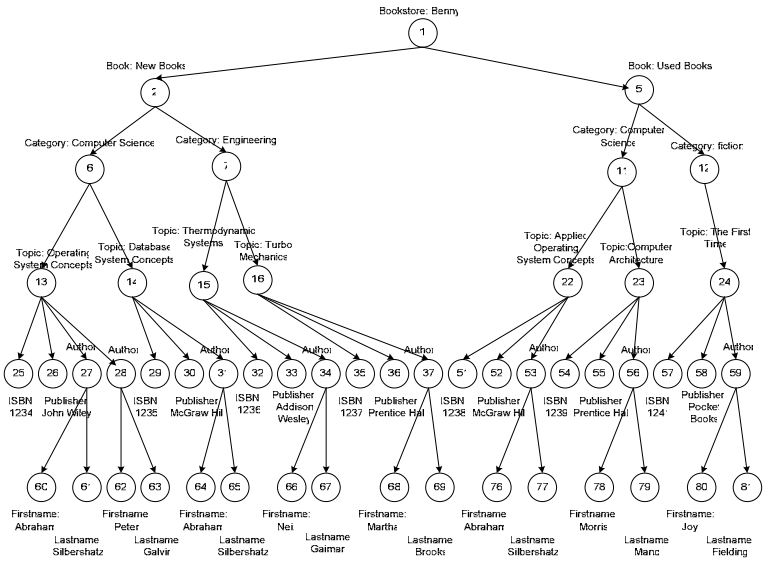

Figure 7: Delete 'topic' nodes whose ancestor does not have the nodes 'book'

4.2 Cost Analysis of Two-point Entry Algorithm

We assume that the cost of DN is calculated during generation of DOM tree. Let N be the total number of nodes in the DOM tree.

Step 1: It returns $\mathfrak{R}$, the result node set. There is no computation involved in this step.

Step 2: For each X_i, i=1,…,m

One comparison at each iteration of the loop implies a total of m comparisons.

For i=1,…,p

For j = 1,…,k

One addition at each iteration of the loop implies $p(k^2+k)/2$ additions

Step 3: Min $\{DN_1, DN_2, \ldots, DN_p\}$ and Min$\{\{ DN_1, DN_2, \ldots, DN_p\} - DN_\ell \}$

The cost to find a minimum will involve p comparisons

Step 4: For j = 1,…,k

One comparison at each iteration of the loop for checking if x_{ij} is either ancestor or descendant of x_{tj}

Thus, a total of k comparisons.

Step 5: For j = 1, …, r

At each iteration match Pindex with XPath string of length ℓ, an O(1) operation as $\ell \ll N$

Thus, a total of r matches.

The steps are repeated on q subtrees. Without the loss of generality, we can assume subtrees with O(N) nodes. The recursion will be at most log m times. Also, note that $p < m$, $k \leq \log N$ and $r \leq \log N$.

Thus, the total cost = $\log m\,[m + p(k^2+k)/2 + p + k + r]$

$\leq \log m\,[m + m(\log^2 N + \log N)/2 + m + \log N + \log N]$

$= \log m\,[m/2 \log^2 N + (m+4)/2 \log N + 2m]$

$= O(\log^2 N)$ (since $m \ll N$, m can be considered a constant)

Although the time complexity of Two-point Entry algorithm (TPA) and Entry-point algorithm (EPA) is the same as we have taken the upper bounds, the actual computation time in TPA is less as compared to EPA. This is because some nodes are eliminated in Step 4 at the intermediate level and a smaller number of subtrees have to be traversed in the next recursion.

The space complexity of Two-point Entry algorithm will remain the same as that of Entry-point algorithm which is O(log(N)) space.

4.3 Correctness of Two-point Entry Algorithm

The correctness of Two-point Entry algorithm can be proved similar to Entry-point algorithm. The only difference between the two algorithms is an intermediate step where the nodes in the entry-set are compared with nodes in the comparison-set that represent the nodes with second minimum descent. The case where length of XPath expression is less than or equal to 3 is trivial and handled as in Entry-point algorithm. Consider an XPath expression of length 4, i.e. $/X_1/X_2/X_3/X_4$. Assume X_2 and X_3 are indexed and that X_3 has the minimum descent and X_2 has the second minimum descent. Then all nodes X_3 represent the entry-set and all nodes X_2 represent the comparison-set. The Pindex of all X_3 nodes is checked in Step 4 to find and delete nodes from the entry-set that do not have X_2 as its parent. The XPath expression is split using X_3 into $/X_1/X_2/X_3$ and X_3/X_4. In Step 5 the Pindex of all remaining X_3 nodes in the entry-set is compared with the path $/X_1/X_2/X_3$ and nodes that do not satisfy the path are deleted. Next, consider all subtrees with X_3 as the root are considered and all result nodes X_4 are returned in $\mathfrak{R}$. It can be argued similarly for the case for XPath expression of length m.

5. Experiments

In this section, we present the performance evaluation of the two proposed algorithms and compare those with some existing algorithms. We have design experiments in the two phases as illustrated in next two sub-sections.

5.1 First Phase Experiments

We have implemented the Entry-point and Two-point Entry algorithms and compared the performance of these algorithms with traditional algorithms with and without indexes. In this section, we present the results of experiments by synthetically generated XML DOM trees. In case of the traditional algorithms with or without indexing, bottom-up strategy is used. For the Entry-point algorithm, we use both Root-first and Bottom-first strategies for comparison purposes.

The experimental environment consists of Sun SPARC Solaris 2.5 and used Java 2 and the IBM parser XML4J to create the DOM tree from the XML document. All the experiments presented herein are using the data sets obtained from the author of Toxin [23] and both Index and DOM were kept in memory in the experiments as in Toxin.

Experiment 1: In this experiment, we compare the processing time of all the methods with increasing number of nodes in the DOM tree. The height of the tree is fixed as 8 and the tree widens as the number of nodes increases. The query used in this experiment is //A0//D2//O4//X7 where D2 and O4 are added to the index list. As bottom-up strategy is used in the traditional method, X7 is added to the index list only for traditional method with index. A0 means that the first level is looking for "A" elements. Table 1 shows the processing times (in seconds) and Figure 8 shows the results graphically.

Number of Nodes	9841	23317	36431	69883	148527
Result Nodes	413	1096	1583	3355	6343
Traditional with No Index	0.12	0.274	0.395	0.967	1.668
Traditional with Index	0.115	0.256	0.39	0.873	1.646
Entry-point Bottom-first	0.053	0.129	0.203	0.407	0.837
Entry-point Root-first	0.017	0.059	0.072	0.25	0.393
Two-point Entry	0.012	0.04	0.07	0.159	0.393

Table 1. Result of experiment 1

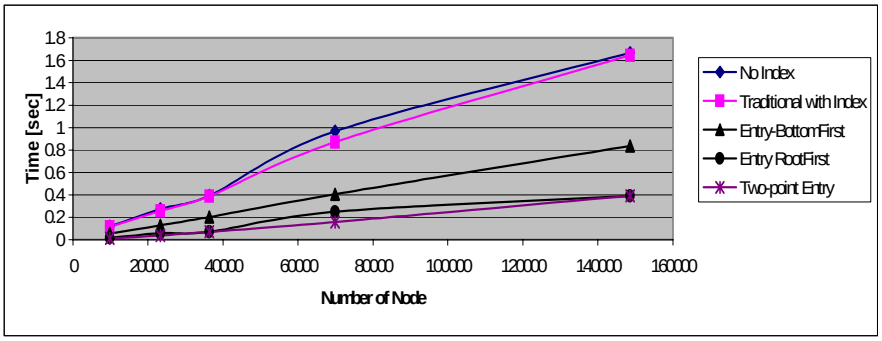

Figure 8. Result of Experiment 1

Although the number of entry nodes increases as the number of nodes increases, the Entry-point with Root-first and Two-point Entry algorithm still show good performance. Note that there is very less difference in the execution times of traditional method without index and with index. We observe that this is due to the fact that X7 was selected for index in a bottom-up strategy used in the traditional method with index. As the number of nodes at the leaf level increase the number of comparisons increase and as such there is very less gain in the execution time as compared to traditional method without index. Also, note that the performance of Entry-point Root-first algorithm is better than Entry-point Bottom-first algorithm due to the fact that D2 and O4 nodes are included in the index, which are middle-level (D2 refers to node D at level 2. The height of the tree is fixed at 8 and levels are numbered from 0 to 7) nodes. As the height is fixed and the number of nodes increases, the Entry-point Root-first algorithm is able to exploit the index better than the Entry-point Bottom-first algorithm. Also, note that in the Root-first approach, nodes in the entry set are eliminated by comparing the path index of the entry-point nodes with the first part of the split XPath expression, where path index gives all nodes from root to that node. However, in the bottom-first approach nodes in the entry set are eliminated by checking if the result nodes have the entry-point nodes as their ancestor. This requires checking the presence of the entry-point node in the path index of the result nodes. If the entry-point node is not in the path index of the result node, we need to delete the result nodes as well as the entry-point nodes. This decreases the time efficiency of the bottom-up approach.

Experiment 2: In this experiment we compare the processing time of all the methods by increasing the height of tree. The number of nodes is fixed as 30,000 and height of the tree increases from 5 to 14. As the number of nodes is fixed, the tree gets narrower and taller. Query used in this experiment is //B1//F3//X13 where B1 and F3 are added to the index list. Also X13 is added to index list only for traditional method with index. Table 2 shows the processing time (in seconds) and Figure 9 shows the graphical results.

Formatted: Centered

Height of Tree	5	6	7	8	9	14
Number of Nodes	29257	28123	27306	31061	29524	32767
Result Nodes	2024	1913	1774	1765	1612	1406
Traditional with No Index	0.415	0.451	0.47	0.551	0.551	0.753
Traditional with Index	0.411	0.445	0.467	0.543	0.546	0.749
Entry-point Bottom-first	0.117	0.112	0.101	0.15	0.137	0.226
Entry-point Root-first	0.106	0.09	0.068	0.128	0.12	0.159
Two-point Entry	0.104	0.06	0.057	0.07	0.073	0.115

Table 2. Result of Experiment 2

The result shows that the execution times for Entry-point Root-first and Two-point Entry algorithms decrease as height increases from 5 to 6. The reason is that the number of entry nodes is decreasing as the DOM tree gets narrower and higher. But if the height is increased further, the execution time increases again due to longer path traversal. The figure also shows that the execution time for the traditional method with index increases as the height increases. As the height of the tree increases, Entry-point Root-first algorithm and Entry-point Bottom-first algorithm perform almost same as both are able to exploit the index at the middle-level equally.

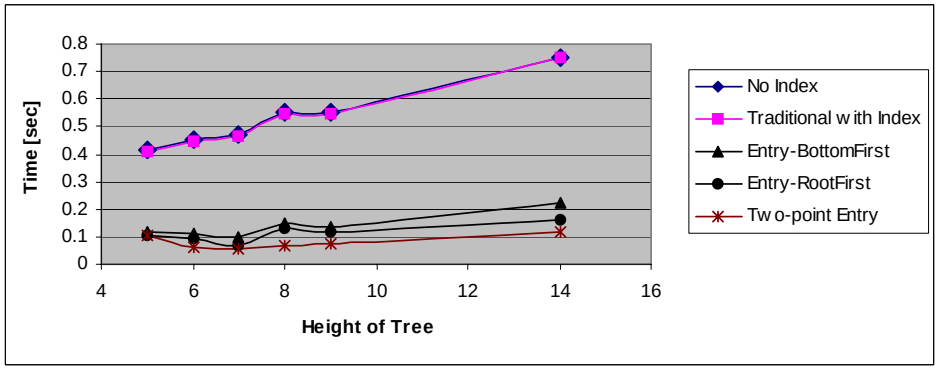

Figure 9. Result of Experiment 2

Experiment 3: Here we compare the processing time of the methods by increasing the number of result nodes. The height of DOM tee and the number of nodes are fixed as 8 and 36431 respectively. Query used in this experiment is /A0//D2//O4//X7 where D2 and O4 are added to the index list. X7 is added to the index list only for traditional method with index. The results show that with increasing number of result nodes Entry-point and Two-point Entry algorithms still perform well as compared to traditional method with or without index. Again Entry-point Root-first algorithm performs better than Entry-point Bottom-first algorithm as Entry-point Root-first algorithm is able to exploit the index at the middle-level nodes. Table 3 shows the processing time (in seconds) and Figure 10 shows the graphical results.

Number of Nodes	36431	36431	36431	36431	36431
Result Nodes	218	531	820	1115	1556
Traditional with No Index	0.254	0.297	0.481	0.406	0.421
Traditional with Index	0.243	0.292	0.457	0.396	0.401
Entry-point Bottom-first	0.082	0.131	0.132	0.186	0.214
Entry-point Root-first	0.023	0.038	0.068	0.09	0.099
Two-point Entry	0.018	0.028	0.039	0.056	0.073

Table 3. Result of Experiment 3

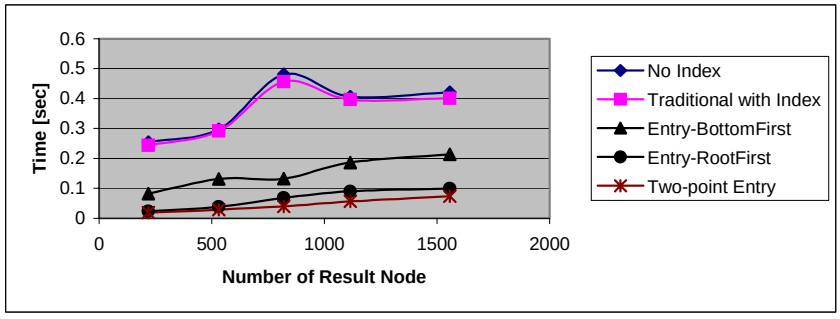

Figure 10. Result of Experiment 3

5.2 Second Phase Experiments

The experiments in the first phase are done on synthetically generated DOM tree and did not represent data from a real world XML data. In second phase experiments, we use ToXin as a benchmark to compare the performance of our proposed algorithms and demonstrate that they work efficiently with regular expressions, such as, root/X_1/X_2/*/…/X_{n-1}/X_n [23]. The benchmark consists of four data sources: the conference papers from the DBLP database (90049 Nodes) [17], a sample of movies from the Internet Movies Database (57858 Nodes) [24], the Shakespeare plays (45868 Nodes) from [7], and religious texts (16234) from [6]. Nindex, Vindex, and Pindex were built and used in the experiments.

Query Type	Query	Characteristics
LL*	Q1*, Q4*, Q7*	Large query answer Large filter selection Relaxed path constraints
LL	Q1, Q4, Q7	Large query answer Large filter selection
LS*	Q2*, Q8*, Q10*	Large query answer Small filter selection Relaxed path constraints
LS	Q2, Q8, Q10	Large query answer Small filter selection
SS*	Q3*, Q5*, Q6*, Q9*	Small query answer Small filter selection Relaxed path constraints
SS	Q3, Q5, Q6, Q9	Small query answer Small filter selection

Table 4. Query Classification

To compare query performance across different documents, Rizzolo [23] has classified the queries with regards to the selectiveness of its path constraints and the sizes of the query answer and node selection in the filter section. Table 4 shows the classification according to these criteria. Following queries have been used in the experiments

1) DBLP

Q1* = " //[year='1998']//title"
Q1 = "/dblp/conference/issues/issue/inproceedings[year='1998']/title"
Q2* = "//conference[title='vldb']/*//title"
Q2 = "/dblp/conference[title='vldb']/issues/issue/inproceedings/title”
Q3* = "//[author='Serge Abiteboul']//title“
Q3 = "/dblp/conference/issues/issue/inproceedings[author='Serge Abiteboul']/title”

2) IMDB

Q4* = "//[genre='Drama']//title";
Q4 = "/movies/movie[genre='Drama']//title"
Q5* = "//[title='Club Sandwich']//year"
Q5 = "/movies/movie[title='Club Sandwich']//year"
Q6* = "//[year='1950']//title"
Q6 = "/movies/movie[year='1950']/title"3) Play

3) Shakespeare
   Q7* = "//[SPEAKER='MARK ANTONY']//LINE"
   Q7 = "/SHAKESPEAR/PLAY/ACT/*/SPEECH[SPEAKER='MARK ANTONY']/LINE"
   Q8* = "//[TITLE='The Tragedy of Antony and Cleopatra']/ACT//LINE"
   Q8 = "/SHAKESPEAR/PLAY[TITLE='The Tragedy of Antony and Cleopatra'] /ACT / * /SPEECH/LINE"
   Q9* = "//[TITLE='The Tragedy of Antony and Cleopatra']//PERSONA"
   Q9 = "/SHAKESPEAR/PLAY[TITLE='The Tragedy of Antony and Cleopatra'] /PERSONAE/PERSONA"

4) Religion
   Q10* = "//[bktshort='Matthew']//v"
   Q10 = "/tstmt/bookcoll/book[bktshort='Matthew']//v"

Query	Query Type	Num of Nodes	Result Nodes	TWNI	TWI	EPBF	EPRF	TPE	ToXin
				Time (in seconds) (0 – non-measurable time)					
Q1*	LL*	90041	1212	1.281	0.843	0.078	0.094	0.094	0.25
Q1	LL	90041	1211	0.391	0.219	0.079	0.093	0.205	0.16
Q2*	LS*	90041	45	2.187	1.703	0	0	0	1
Q2	LS	90041	45	0	0.344	0	0.016	0.469	0.55
Q3*	SS*	90041	2	1.297	0.828	0	0	0	0.21
Q3	SS	90041	2	0.031	0.547	0	0	0	0.15
Q4*	LL*	57889	409	0.828	0.516	0.062	0.062	0.062	0.38
Q4	LL	57889	409	0.125	0.047	0.063	0.078	0.063	0.35
Q5*	LS*	57889	1	0.859	0.594	0	0	0	0
Q5	LS	57889	1	0.032	0.093	0	0	0	0
Q6*	SS*	57889	20	0.843	0.594	0	0	0	0.13
Q6	SS	57889	20	0.032	0.031	0.015	0	0.016	0.09
Q7*	LL*	45618	851	1.703	2.672	0.015	0.016	0.016	0.44
Q7	LL	45618	851	0.39	1	0.015	0.016	0.016	0.22
Q8*	LS*	45618	3560	1.484	2.25	0.125	0.141	0.078	0.81
Q8	LS	45618	3560	0.125	2.141	0.109	0.109	0.079	0.56
Q9*	SS*	45618	35	0.218	0.016	0.015	0.032	0.032	0.58
Q9	SS	45618	10	0	0.016	0	0	0	0.11
Q10*	LS*	16801	1130	0.859	0.844	0.031	0	0	0.1
Q10	LS	16801	1130	0.031	0.015	0	0.032	0.016	0

Table 5. Results of Experiment 4

Experiment 4: Build DOM tree for the databases and build Nindex, Vindex, and Pindex indexes. Compare the processing time of the five methods - Traditional With Index (TWI), Traditional With No Index (TWNI), Entry-point Root-first (EPRF), Entry-point Bottom-first (EPBF), Two-point Entry (TPE) methods, and ToXin for query classifications given in Table 4. Table 5 shows the results of execution times (in seconds) for these methods on the selected queries. For the traditional method with index, bottom-up strategy was used so that Nindex for the leaf nodes could be utilized. In the Entry-point algorithms, the efficient backward navigation is not emphasized. For example, for querying Q2 = "/dblp/ conference [title='vldb']/issues/issue/inproceedings/title", the "conference" nodes should be filtered by their child node

”title”, which has the value “vldb”. If “conference” node is selected as entry-point, Entry-point algorithm returns the already filtered nodes so that the Pindex of the entry nodes is compared with /dblp and then /issues/issue/inproceedings/title is processed by the algorithm. If the “inproceedings” node is selected as entry-point, the Pindex of the entry nodes is compared with /dblp/conference/issues/issue/inproceedings. But an extra process is required for checking that the “conference” nodes have child nodes “title” with value “vldb”. This takes significant time in the program, but it can be solved by an efficient implementation. Vindex is built for all value nodes and Nindex is built for selected nodes. The results of ToXin were retrieved from [23]. Note that 0 in the table shows non-measurable time.

In Table 5, each Q* (shaded rows) query (except Q8*) is short and has only one entry-point, so performance of Entry-point and Two-point Entry must be same. With Q* queries, Entry-point algorithm shows much better performance than Traditional method with or without index because of Vindex and small number of entry-points.

In real database, the structure of tree is usually simpler than the generated DOM tree. The generated DOM tree has variable nodes which have different names and value at each level and have different ancestors, so comparison level in the Two-point Entry algorithm removes many entry nodes, which result in better performance than Entry-point algorithm. However, in the real database tree, the nodes that have the same name usually have the same ancestors in DBLP data. So comparison level in the Two-point Entry algorithm does not remove many entry nodes. This characteristic causes poor performance of Two-point Entry algorithm and it is shown in some query results.

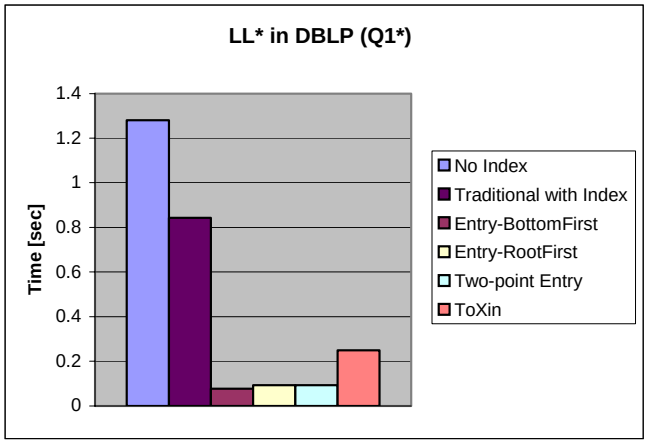

Figure 11. Comparative performance of Query Q1*

Figure 11 shows the results of query Q1* = " //[year='1998']//title". In this case, “1998” was added to Vindex and used in Entry-point and Two-point Entry algorithms. Since there is only one entry-point candidate here, Entry-point and Two-point Entry algorithms show similar performance. Because of the bottom-up strategy, the leaf node “title” was added to Nindex for Traditional method with index. Figure 12

shows the result of query Q1 = "/dblp/conference/issues/issue/inproceedings[year='1998']/title". In this query, “1998” was added to Vindex and “issue” and “inproceedings” were added to Nindex for Entry-point and Two-point Entry algorithms. In this query, Entry-point algorithm shows much better performance than the Traditional method with index because of small number of entry-points (“inproceedings” nodes which have “year” child node with value “1998”). However the Two-point Entry algorithm does not perform very well. If the “inproceedings” is the entry-point, “issue” node would be the next entry-point which has much more nodes. Since the Two-point Entry algorithm needs the comparison of the two entry-point nodes, it consumes significant time to compare if there are many nodes. Also if we carefully review the structure of the XML tree, most “inproceedings” nodes have “issue” nodes as an ancestor node. As a result, only few compared nodes are eliminated after significant time-consumed comparisons.

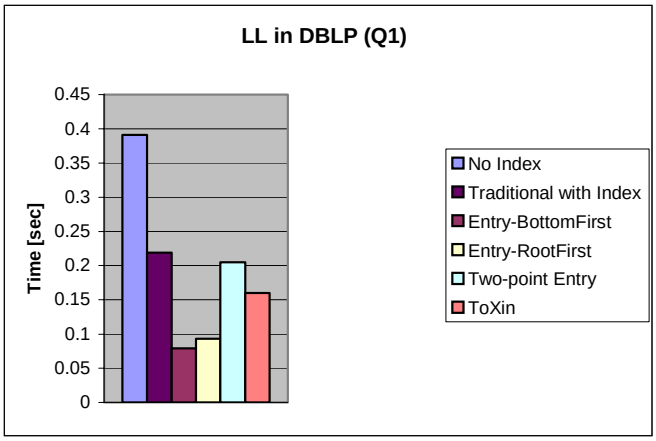

Figure 12. Comparative performance of Query Q1

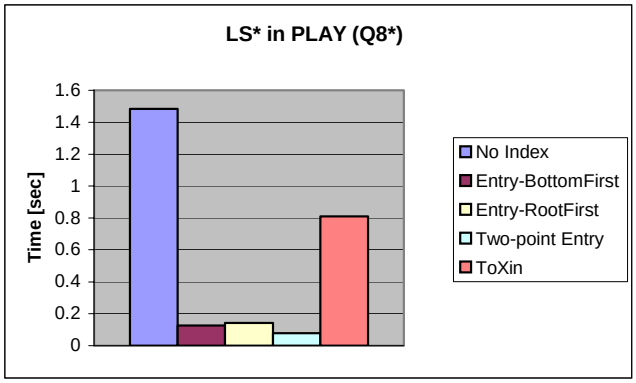

Figure 13. Comparative performance of Query Q8*

Figure 13 shows the results of query Q8* = "//[TITLE='The Tragedy of Antony and Cleopatra']/ACT//LINE“. In this query, “The Tragedy of Antony and Cleopatra” was added to Vindex and “ACT” was added to Nindex. In this query, the Two-point Entry algorithm performs very well because

there are not many entry nodes to compare. The leaf node “LINE” was added to Nindex for Traditional method with index. The results of Traditional method with index are not shown in the chart because of high execution time.

Figure 14 shows the result of query Q8 = "/SHAKESPEAR/PLAY[TITLE='The Tragedy of Antony and Cleopatra']/ACT/*/SPEECH/LINE“. In this query, “The Tragedy of Antony and Cleopatra” was added to Vindex and “ACT” was added to Nindex. In this query, the Two-point Entry algorithm also performs very well because there are not many entry nodes to compare. The leaf node “LINE” was added to Nindex for Traditional method with index. Traditional method without index shows much less execution time for the path without regular expression.

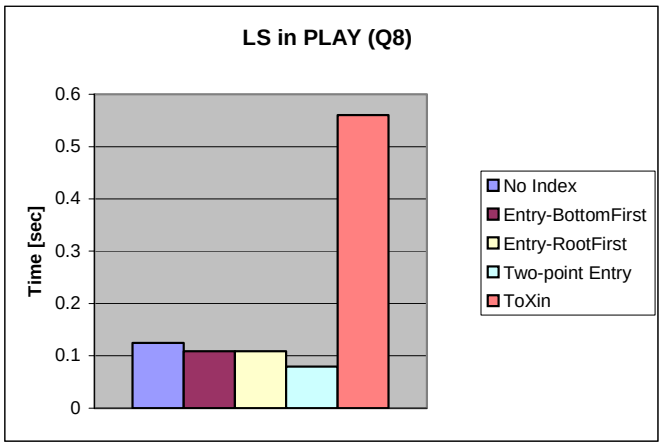

Figure 14. Comparative performance of Query Q8

6. Conclusions

In this paper, we have used three types of indexes on XML data to execute XPath queries efficiently. We proposed two algorithms to process XPath queries using these indexes to optimize their execution time. We have simulated both Root-first and Bottom-first approaches and have observed that processing of XPath queries using the proposed Entry-point and Two-point indexing techniques perform much better than traditional algorithms (with or without indexes) and ToXin. For future work, we plan to implement these indexing techniques within XQuery (XML query language) engine and measure their performance on a large set of XML data.

Acknowledgement: Authors sincerely acknowledge the help provided by Youn J. Heo in doing experiments and repeating them.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom and J.L. Wiener. The Lorel Query Language for Semistructured Data, International Journal on Digital Libraries 1(1) 1997, pages 68-88.

[2] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural

Joins: A Primitive for Efficient XML Query Pattern Matching, In Proceedings of the 18th International Conference on Data Engineering (ICDE), 26 February - 1 March 2002, San Jose, CA. IEEE Computer Society 2002, page 141..

[3] G. Amato, F. Debole, P. Zezula and F. Rabitti. YAPI: Yet Another Path Index for XML Searching, Research and Advanced Technology for Digital Libraries, 7th European Conference, ECDL 2003, Trondheim, Norway, August 17-22, 2003, Proceedings. Lecture Notes in Computer Science 2769 Springer 2003, pages 176-187.

[4] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0, W3C Working Draft, 11 February 2005. http://www.w3.org/TR/2005/WD-xpath20-20050211

[5] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. W3C working draft, 11 February 2005. http://www.w3.org/TR/2005/WD-xquery-20050211/

[6] J. Bosak. Religious Texts in XML. www.ibiblio.org/xml/examples/religion.

[7] J. Bosak. Complete Plays of Shakespeare in XML. www.ibiblio.org/xml/examples/shakespeare.

[8] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002. ACM 2002, pages 310-321.

[9] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi and L. Tanca. XML-GL: A Graphical Language for Querying and Restructuring XML Documents. In Proceedings of the 8th International World Wide Web Conference, Toronto, Canada, May 1999, Computer Networks 31(11-16), 1999, pages 1171-1187.

[10] D. Chamberlin, J. Robie and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources, In Proceedings of the Third International Workshop on the Web and Databases, WebDB 2000, Dallas, Texas, USA, May 18-19, 2000, in conjunction with ACM PODS/SIGMOD 2000, pages 53-62.

[11] B. Cooper, N. Sample, M. Franklin, G. R. Hjaltason, and M. Shadmon, A Fast Index for Semistructured Data. In Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy. Morgan Kaufmann 2001, pages 341-350.

[12] A. Deutsch, M. Fernandez, D. Florescu, A. Levy and D. Suciu, A Query Language for XML, In Proceedings of the 8th International World Wide Web Conference, Toronto, Canada, May 1999, Computer Networks 31(11-16), 1999, pages 1155-1169.

[13] Document Object Model (DOM), http://www.w3.org/DOM/

[14] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Database. In Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97), August 25-29, 1997, , page 436-445, Athens, Greece. Morgan Kaufmann

[15] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering Indexes for Branching Path Queries. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002. ACM 2002, pages 133-144.

[16] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA. IEEE Computer Society 2002, pages 129-140.

[17] M. Ley. DBLP database web site, www.informatik.unitrier.de/~ley/db.

[18] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In Proceedings of the 27th International Conference on Very Large Data Bases, VLDB 2001, September 11-14, 2001, Roma, Italy. Morgan Kaufmann 2001, pages 361-370.

[19] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom, Lore: A Database Management System for Semi-structured Data. SIGMOD Record, 26(3), 1997, pages 54-66.

[20] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing Semistructured Data, Technical report, Stanford University, Stanford CA, February 1998.

[21] T. Milo and D. Suciu. Index Structures for Path Expressions. 7th International Conference on Database Theory (ICDT '99), Jerusalem, Israel, January 10-12, 1999, Proceedings. Lecture Notes in Computer Science 1540 Springer 1999, pages 277-295.

[22] C. Qun, A. Lim, K. W. Ong. D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003. ACM 2003, pages 134-144.

[23] F. Rizzolo, and A. Mendelzon. Indexing XML Data with ToXin. In Proceedings of the Fourth International Workshop on the Web and Databases, WebDB 2001, Santa Barbara, California, USA, May 24-25, 2001, in conjunction with ACM PODS/SIGMOD 2001. pages 49-54.

[24] The Internet Movie Database. www.imdb.com.

[25] H.Wang, S. Park, W. Fan, and P.S.Yu. VIST: A Dynamic Index Method for Querying XML Data by Tree Structures. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003. ACM 2003, pages 110-121.

Note: Recommended by Yannis Ioannidis, Area Editor

Appendix : Tree Schemas of Document Samples

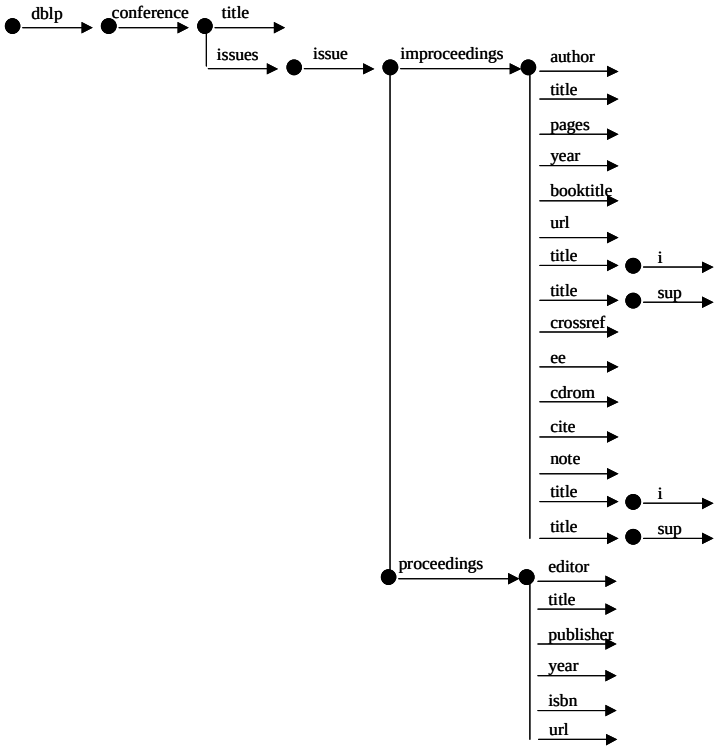

Figure A1. Tree schema of DBLP sample

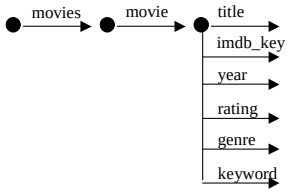

Figure A2. Tree schema of IMDB sample

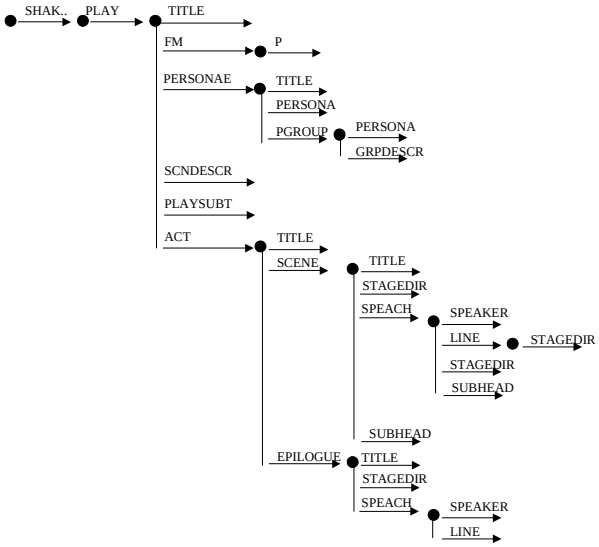

Figure A3. Tree schema of PLAY sample

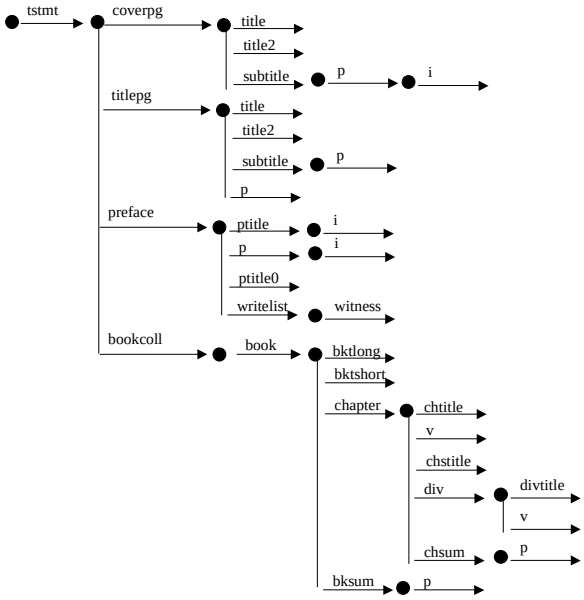

Figure A4. Tree schema of RELIGION sample