Efficient XPath Evaluation

Bing Wang[1], Ling Feng[2], and Yun Shen[1]

[1] Department of Computer Science, University of Hull,
Hull, HU6 7RX, United Kingdom
{B.Wang, Y.Shen}@dcs.hull.ac.uk
[2] Department of Computer Science, University of Twente,
PO Box 217, 7500 Enschede The Netherlands
ling@cs.utwente.nl

Abstract. Inspired by the best querying performance of ViST among the rest of the approaches in the literature, and meanwhile to overcome its shortcomings, in this paper, we present another efficient and novel geometric sequence mechanism, which transforms XML documents and XPath queries into the corresponding geometric data/query sequences. XML querying is thus converted to finding non-contiguous geometric subsequence matches. Our approach ensures correct (i.e., without semantic false) and fast (i.e., without the costly post-processing phase) evaluation of XPath queries, while at the same time guaranteeing the linear space complexity. We demonstrate the significant performance improvement of our approach through a set of experiments on both synthetic and real-life data.

1 Introduction

With the advent of XML as a standard for data representation and exchange on the Web, indexing and querying XML documents becomes increasingly important for current and future data-centric applications. Substantial research efforts [4,7,5,11] have been conducted to structurally index and retrieve data from XML documents.

The first problem of retrieving data from XML documents is how to deal with specific queries containing constraints related to the content of the documents. Providing a uniform index structure [15] for both the structure and content information of an XML document is thereupon desirable. More importantly, the mechanism should be preferably implemented using some well-supported DBMS data structures like B+Tree.

The second problem is that a query compatible to XPath is modeled as a tree, referred to as a *twig*, and can be complicated [6] when wildcards "*" and self-or-descendent axis("//") are presented (for example, Q5 in Table 3). To match such a complex query against a document tree without corresponding preprocessing mechanism is equivalent to the tree inclusion problem and has been proved to be NP-complete [1].

Previous research efforts have been devoted to twig pattern matching for several years. XISS [10] is the first to break twig pattern query into binary twigs,

J. Eder et al. (Eds.): ADBIS 2005, LNCS 3631, pp. 223–237, 2005.

and "stitch" the binary twigs (i.e. two nodes with parent-child relationship) together to obtain the final results. State-of-the-art mechanisms, i.e. structural join [3], holistic twig join [12], have been proposed to stitch root-to-node paths together by using specially designed stacks. Additionally, some index structures, such as XR-Tree [8] and XB-Tree [12], have been proposed to optimize the above twig join operations. However, the performance of all the above mechanisms is suffered from the time-consuming join operations.

Wang et al. proposed a novel ViST mechanism [15], which transforms both XML documents and XPath queries into structure-encoded sequences so that the twig pattern matching problem is converted to subsequence matching problem. The advantage of this approach is that it does not need to break down a twig pattern into root-to-leaf paths and process them individually, thus avoiding the heavy join operations to join intermediate results. This method improves all the previous searching mechanisms significantly. However, ViST has three major shortcomings. First, its structure-encoded sequence model can cause the semantic false problem. That is, an XML fragment which semantically matches a query may not be returned. Second, ViST may lead to false answers (false alarms) because its encoding method can not fully sustain the structures of XML data trees. Time-consuming refinement phase or post-processing phase has to be called to eliminate the false answers. Although Wang et al. [14] further proposed a way to eliminate the post-processing phase with $O(n^2)$ total size complexity (where n is the total node number in a data tree), it depends on specialized *trie + path link* structure to find sibling-cover in the trie and remove the false answers, in which the semantic false still exists. Third, ViST can not guarantee the linear size complexity of structure-encoded sequence. In the worst case, the total size of structure-encoded sequence is $O(n^2)$ when a document is a unary tree.

To overcome the above three problems, in this paper, we present another encoding mechanism to transform XML documents and XML queries into geometric sequences. Our objective is to ensure correct (i.e. without semantic false) and fast (i.e. without the post-processing phase) evaluation of XPath queries, while at the same time guaranteeing the linear size complexity of the sequence. This approach enables us to achieve better storage and query performance than ViST.

2 The Problems with ViST

As proposed in [15], a structure-encoded sequence is derived from a prefix traversal of an XML document, in format of a sequence of (*symbol*, *prefix*) pairs, (a_1, p_1), (a_2, p_2), ..., (a_n, p_n), where a_i represents a node in the XML document tree ($a_1a_2...a_n$ is the pre-order sequence) and p_i is the encoded path from root to a_i. In the same spirit, XML queries are converted into structure-encoded query sequences in which "*" and "//" are explicitly encoded. Querying XML is equivalent to finding non-continuous subsequence matches in ViST. The corresponding structure-encoded sequence of the XML document example in Figure 1 is illustrated in Figure 2. Let T_{Str} denote the structure encode sequence.

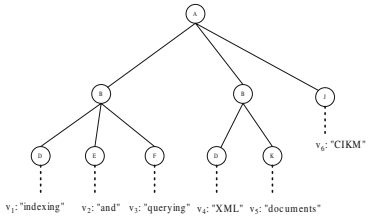

Fig. 1. An Example of XML Document in Tree Structure

T_{Str} = **(A, ε) (B, A) (D, AB) (v_1, ABD) (E, AB) (v_2, ABE) (F, AB) (v_3, ABF) (B, A) (D, AB) (v_4, ABD) (K, AB) (v_5, ABK) (J, A) (v_6, AJ)**

Fig. 2. Structure-Encoded Sequence of the XML Document in ViST Approach

The problem of false answers (a.k.a false alarms) arises immediately in ViST in which an XML document is represented by a structure-encoded sequence. For example, given a query Q2: $/A/B[./E][./K]$, its tree structure is shown in Figure 8(b), and its corresponding structure-encoded query sequence is shown in Figure 3. The underlined non-continuous subsequence in T_{Str} marks a result (matching). However, it is a false answer since the structure expressed in Q2 does not exist in the XML document example. We call this kind of queries *non-existence false*.

$$Q2_{Str} = \textbf{(A, } \varepsilon\textbf{) (B, A) (E, AB) (K, AB)}$$

Fig. 3. Structure-Encoded Sequence of Q2

Consider, for another example, Q3 shown in Figure 8(c), its structure-encoded sequence is shown in Figure 4. In ViST, Q2 and Q3 may return the same results because Q2 is a subsequence of Q3. We call this kind of query pairs *non-equivalence false*. It implies that refinement phase or post-processing phase has to be called to eliminate the false answers in these two cases. However, the process may not be always trivial.

Moreover, ViST has a serious semantic flaw in transforming XPath queries into structure-encoded sequences. Suppose we have an XML fragment:

$$< A >< B >< K >< C >< /C >< /K >< /B >< /A >$$

and its corresponding structure-encoded sequence:

$$Frag_{Str} =< A, \varepsilon >< B, A >< K, AB >< C, ABK >$$

If Q: /A[./B//C][//K] is transformed into a structure-encoded query sequence and evaluated against this fragment:

$$Q_{Str} =< A, \varepsilon >< B, A >< C, AB// >< K, A// >$$

$$Q3_{Str} = \textbf{(A, } \varepsilon\textbf{) (B, A) (E, AB) (B, A) (K, AB)}$$

Fig. 4. structure-encoded Sequence of Q3

we can see that there is no such subsequence matching of Q_{Str} in $Frag_{str}$ since K appear after C in ViST, as shown in Figure 5. However, Q semantically matches the fragment. This flaw can hardly be fixed since the order among the items in a structure-encoded sequence is indispensable in ViST. We call this semantic flaw of ViST *semantic false*.

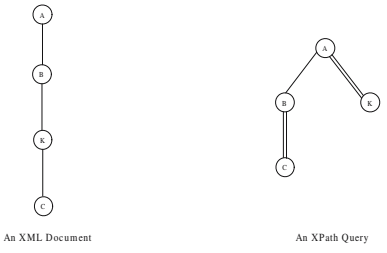

Fig. 5. A Semantic False Query Evaluation in ViST

3 Proposed Method

To overcome the shortcomings of ViST, in this section, we present a geometric-encoding mechanism, which transforms XML documents/queries into geometric data/query sequences. Further enhancement to our geometric encoding approach is also described.

3.1 Mapping XML Documents into Geometric Data Sequences

We firstly model XML as an ordered, node labeled, rooted tree. More formally, consider a graph $\mathbf{T} = (\mathbf{V_G}, \mathbf{V_T}, \mathbf{v_r}, \mathbf{E_G}, \mathbf{label_{node}}, \mathbf{nid}, \sum_{\mathbf{T}})$. V_G is the set of element nodes and V_T is the set of text nodes. $\forall v \in V_T$, v has no outgoing edge. v_r is the root of the XML data tree, where there exists a path from v_r to v, $\forall v \in V_G \cup V_T$. Moreover, it implies that v_r has no incoming edge. Each node $v \in V_G \cup V_T$ is labeled through the function $label_{node}$ over the set of terms, $\sum_T$. The label of a node $v \in V_G$ is referred to as the tag name. The label of $v \in V_T$ is referred to as a distinct keyword contained in the corresponding text. We use quotation mark in future figures to distinguish the label in V_T.

Each edge e, $e \in E_G$, is a parent-child edge, denoting the parent-child relationship. The parent node is denoted as v_{e_p}, and the child node is denoted as v_{e_c}. A path is a sequence of edges starting from the node v_i to the node v_j, denoted as e_i, e_{i+1}, ..., e_j. A node v_i is ancestor of v_j *iff* a path to v_j goes through v_i. The order among the sibling nodes is distinguished. Each node is

assigned a unique nid number for indexing and querying purpose. We refer T_{v_i} as the subtree induced by node v_i. Figure 1 shows an example of our data model. The solid edges represent E_G. The dashed edge denotes a edge e, $v_{e_p} \in V_G$, and $v_{e_c} \in V_T$. The quoted string represents a label of a node $v \in V_T$.

We secondly transform an XML document into a sequence by pre-order traversing the above XML data tree, recording a node's parent when backtracking. For the example in Figure 1, its sequence representation is shown in Figure 6.

$$\mathbf{ABDv_1DBEv_2EBFv_3FBABDv_4DBKv_5KBAJv_6JA}$$

Fig. 6. A Sequence Representation of the Example XML Document

To clearly represent a sequence, we slightly modify the above sequence to indicate the start (s), intermediate(i), end (e) positions of a specific node which appears multiple times in the sequence. The modified sequence representation is shown in Figure 7. Let T_{Geo} denote the modified sequence, and $\mathbf{f}$: T $\rightarrow T_{Geo}$. Easily we can see $\mathbf{f}$ is a bijection between T_{Geo} and T. In the rest of the paper, we call the modified sequence *geometric sequence*. We later show in Section 4 that those extra $symbol_i$ and $symbol_e$ require trivial processing in both indexing and querying process.

$$T_{Geo}=\underline{\mathbf{A}_s\mathbf{B}_s\mathbf{D}_s}\mathbf{v}_1\underline{\mathbf{D}_e}\mathbf{B}_i\mathbf{E}_s\mathbf{v}_2\mathbf{E}_e\mathbf{B}_i\mathbf{F}_s\mathbf{v}_3\mathbf{F}_e\underline{\mathbf{B}_e\mathbf{A}_i}\mathbf{B}_s\mathbf{D}_s\mathbf{v}_4\mathbf{D}_e\mathbf{B}_i\underline{\mathbf{K}_s}\mathbf{v}_5\underline{\mathbf{K}_e}\mathbf{B}_e\mathbf{A}_i\ \mathbf{J}_s\mathbf{v}_6\mathbf{J}_e\underline{A_e}$$

Fig. 7. A Geometric Sequence Representation of the Example XML Document

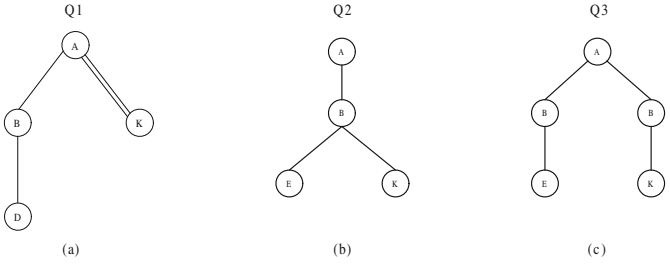

Fig. 8. Example of Query Sequences in Tree Form

3.2 Transforming XPath Query into Geometric Query Sequence

A query compatible to XPath is modeled as a tree, as shown in Figure 8. The core of evaluating an XPath query at an XML document is finding all the answers of such a twig pattern matching the constraints (axes, nested structure, terms etc.) of the query. Moreover, a query can be complicated when wildcards "*" and self-or-descendent axis("//") are presented. When we transform an XPath query

Table 1. List of Q1, Q2, and Q3 in Geometric Query Sequences

Path Expression	Geometric Query Sequence
Q1: /A[B/D][//K]	$Q1_{Geo}$: $A_s\ B_s\ D_s\ {D_e}^p\ {B_e}^p\ {A_i}^u\ K_s\ K_e\ A_e$
Q2: /A/B[./E][./K]	$Q2_{Geo}$: $A_s\ B_s\ E_s\ {E_e}^p\ B_i\ K_s\ {K_e}^p\ {B_e}^p\ A_e$
Q3: /A/B[E]/following-sibling::B/K	$Q3_{Geo}$: $A_s\ B_s\ E_s\ {E_e}^p\ {B_e}^p\ A_i\ B_s\ K_s\ {K_e}^p\ {B_e}^p\ A_e$

into a geometric query sequence in a similar way of mapping XML documents into geometric sequences, we ensure that all the information in the XPath query is preserved. We show this by using example queries Q1, Q2, and Q3 in Table 1. Their tree structures are shown in Figure 8.

Consider the example query Q2: /A/B[./E][./K], its tree structure is shown in Figure 8(b). When we transform it into a geometric query sequence, we must preserve: (1) A is parent of B, and (2) B is parent of both E and K. In this paper, Q2 is transformed into a geometric query sequence: $A_s\ B_s\ E_s\ {E_e}^p\ B_i\ K_s\ {K_e}^p\ {B_e}^p\ A_e$, where p implies that the upcoming item is *parent* of the current item. As we can observe, any internal node is followed by its $parent_e$ or $parent_i$ in the geometric sequence. However, a E_e may be followed by B_i in real data sequence not B_e. This issue can be easily solve by defining B_i *equals to* B_e when determining the parent relationship. If p is not explicitly stated, the relationship is ancestor-descendant ("//") by default.

Similarly, for query Q1: /A[B/D][//K], its tree structure is shown in Figure 8(a). When we transform it into geometric sequence, we must preserve: (1) D is a child node of B which, in turn, a child node of A and (2) K is a descendant of A. As we state in previous section, ViST may incur semantic false when transforming Q1 into structure-encoded query sequence since there is no explicit information of the relationship between K and B (D) stated. In this case, we add "u" to a specific node which has at least two child nodes and meanwhile "//" is involved. Q1 is transformed into a geometric sequence: $A_s\ B_s\ D_s\ {D_e}^p\ {B_e}^p\ {A_i}^u\ K_s\ K_e\ A_e$ as shown in Table 1, where u signifies that semantic uncertainty may occur in the upcoming item.

After an XPath Query is transformed into a geometric query sequence, querying XML documents is equivalent to finding (under the guidance of flag 'p' and/or 'u') non-contiguous subsequence matches in the corresponding geometric data sequences. For query Q1, the underlined non-contiguous subsequence matching in Figure 7 marks a correct matching (i.e. the example document satisfies the query).

Revert to the semantic false problem presented in ViST, as illustrated in Section 2. Let's see how our geometric encoding mechanism avoids the problem. The geometric data/query sequence of the XML fragment and the query (Figure 5)is as follows:

$$Frag_{Geo} = A_s B_s K_s C_s C_e K_e B_e A_e$$

$$Q_{Geo} = A_s B_s C_s C_e^p B_e^p A_i^u K_s K_e A_e$$

To match Q_{Geo} against $Frag_{Geo}$, when we evaluate A_i^u, we resume the range information of A_s.It implies that we will search for K_e in $Frag_{Geo}$ within the range of A_s instead of B_e, starting with which, we can find K_s, K_e and A_e. Section 4 will introduce an elegant stack mechanism to implement the method.

3.3 Numbered Geometric Sequence

Furthermore, consider the fact that in an XML document, the same element names may appear several times. Given the data tree in Figure 1 and query Q2 in Figure 8(b), Q2 should return no result. However, in Table 1, $Q2_{Geo}$ does not provide enough information to eliminate the second B_e, which implies that a result would be returned if the second B_e is included.

$$T_{Geo_{num}} = \mathbf{A}_{1_s}\mathbf{B}_{1_s}\mathbf{D}_{1_s}\mathbf{v}_1\mathbf{D}_{1_e}\mathbf{B}_{1_i}\mathbf{E}_{1_s}\mathbf{v}_2\mathbf{E}_{1_e}\mathbf{B}_{1_i}\mathbf{F}_{1_s}\mathbf{v}_3\mathbf{F}_{1_e}\mathbf{B}_{1_e}\mathbf{A}_{1_i}\mathbf{B}_{2_s}\mathbf{D}_{2_s}\mathbf{v}_4\mathbf{D}_{2_e}\mathbf{B}_{2_i}\mathbf{K}_{1_s}\ \mathbf{v}_5\mathbf{K}_{1_e}\mathbf{B}_{2_e}\mathbf{A}_{1_i}\mathbf{J}_{1_s}\mathbf{v}_6\mathbf{J}_{1_e}\mathbf{A}_{1_e}$$

Fig. 9. Numbered Geometric Sequence Representation of XML Document

To tackle this problem, we enhance the basic geometric-encoding data sequence by numbering each (repeated) item, so that a geometric sequence is sequence of $symbol_{number_{(s|i|e)}}$. Figure 9 gives a numbered geometric data sequence. Note that we do not number the geometric query sequence. We can see that there is no such subsequence matching in $T_{Geo_{num}}$. Additionally, for each query sequence having $symbol_i$, we only choose the first one in T_{Geo} on the basis of the fact that the rest $symbol_i$ is redundant in querying process. Moreover, for queries having the same child nodes in branches, it is equal to find all the non-decreasing subsequence matching in geometric sequence for all the nodes with the same names. "*" is handled as a range query as the same to ViST. If p is not explicitly stated in geometric sequence model, "//" is then default and not instanced on the basis of the fact that "//" only represents ancestor-descendant relationship. By contrasting to ViST's instance step, resource-consuming prefix checking and range query steps connected with "//" are eliminated in our geometric sequence model. Due to lack of space, the correctness of querying XML through numbered geometric data/query sequence matching is not provided.

4 Holistic Sequence Matching

To acclerate XPath evaluation, the challenge of our geometric model is to (i) avoid the semantic false problem, (ii) eliminate the false answers without refinement or post-processing phases, and (iii) provide a linear storage complexity mechanism to reduce the size of index. In section 3, we show that the total size of numbered geometric sequence is O(n). In this section, we demonstrate our subsequence matching can find all the correct answers without refinement or post-processing phase which is inevitable in ViST.

Table 2. List of Q1, Q2, and Q3 in Optimized Geometric Query Sequences

Path Expression	Geometric Sequence
Q1: /A[B/D][//K]	$Q1_{OptGeo}$: $A_s\ {D_e}^p\ {B_e}^p\ {A_i}^u\ K_e\ A_e$
Q2: /A/B[./E][./K]	$Q2_{OptGeo}$: ${E_e}^p\ B_i\ {K_e}^p\ {B_e}^p\ A_e$
Q3: /A/B[E]/following-sibling::B/K	$Q3_{OptGeo}$: ${E_e}^p\ {B_e}^p\ A_i\ B_s\ {K_e}^p\ {B_e}^p\ A_e$

4.1 Index Structure

We adopt a hierarchical indexing structure similar to ViST with some modifications. Each item in a geometric data sequence is in form of $(symbol_{number_{(s|i|e)}})$. Items in a geometric sequence are first put into a trie-like structure. Then each node in the trie is assigned two extra elements "preorder" and "size", where "preoder" is the pre-order traversal position of the node in the data tree, and "size" is used for dynamic scope allocation purpose, whose detail study can be found in the [13]. To build the index structure, each node in the trie, in format of ($symbol_{number_{(s|i|e)}}$, preorder, size), is firstly inserted into a sequence B+Tree index (i.e. SB-Index) using its $symbol_{(s|i|e)}$ as the key. For all the nodes with the same $symbol_{(s|i|e)}$, they are inserted into a position B+Tree (i.e. PB-Index) using its preorder as the key. Figure 10 illustrates the index structure used.

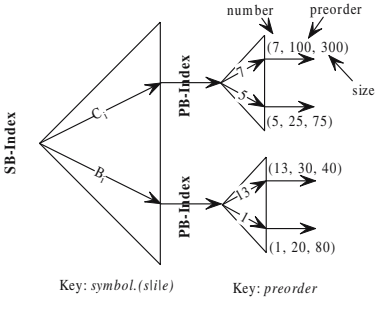

Fig. 10. Index Structure: SB-Index and PB-Index

4.2 Bottom-Up XPath Evaluation

Observing that the performance of evaluating XPath queries over XML documents is significantly affected by the lengths of geometric query sequences, we improve our subsequence matching algorithm on the basis of optimized geometric query sequence transformation. The rational behind is that instead of keeping pairs of nodes like B_s and B_e in a query sequence, we can actually remove one of them without loss of semantics while performing subsequence matching.

Here, we propose a geometric query sequence transformation rule, with an aim to minimize the length of the query sequence. That is: *removing all the*

$symbol_s$ *unless it connects with a* $symbol_i^u$. [3] Examples of the optimized XPath query sequences after transformation are listed in Table 2.

Interestingly, the transformed query subsequences enable us to perform query evaluation in a bottom-up manner, since we start our subsequence matching from a $symbol_e$. For example, given the query $Q_2 : /A/B/[./E][./K]$ in Figure 8 and its optimized geometric query sequence: $E_e^p\ B_i\ K_e^p\ B_e^p\ A_e$, we start the evaluation process from E_e instead of A_s. In comparison, the matching algorithm described in ViST exhibits the top-down flavor.

To facilitate the optimized geometric subsequence matching, we improve our stack mechanism accordingly, where only one set of stacks called **symbol** stacks are involved. We use $Stack_{symbol}$ to denote the stack which accommodates items having *symbol*.

Given an optimized geometric query sequence $q_{1_{v_1}}^{l_1}\ q_{2_{v_2}}^{l_2}\ \ldots\ q_{m_{v_m}}^{l_m}$ and a geometric data sequence $d_{1_{num_{1,V_1}}} d_{2_{num_{2,V_2}}} \ldots d_{n_{num_{n,V_n}}}$, where $(m \leq n)$, $\forall x(1 \leq x \leq m)\ (v_x = s|i|e) \wedge (l_x = u|p|\ )$, and $\forall y(1 \leq y \leq n)\ (V_y = s|i|e)$. Starting with the empty stacks, we scan across the two sequences from left to right. When two equal symbols encounter (i.e., $q_x = d_y$ and $v_x = V_y$) in $q_{x_{v_x}}^{l_x}$ and $d_{y_{num_{y,V_y}}}$, we consider the following situations.

[Case 1] $(v_x = V_y = s)$

We push $d_{y_{num_{y,s}}}$ into the symbol stack $Stack_{d_y}$.

[Case 2] $(v_x = V_y = e)$

There exist two possibilities. 1) When the top item of $Stack_{d_y}$ has a subscript *intermediate* flag i, we check whether $d_{y_{num_{y,i}}}$ has the same num_y as this top item. If they are the same, we push $d_{y_{num_{y,i}}}$ into $Stack_{d_y}$; otherwise a mismatch happens and we start our backtracking process. That is, we pop all those candidate items, which lie between $d_{y_{num_{y,e}}}$ and the top item in $Stack_{d_y}$, out of the corresponding symbol stacks including this top item, and continue to re-search these candidate items in the data sequence.

2) When the top item of $Stack_{d_y}$ has a subscript *end* flag e, we check whether $d_{y_{num_{y,i}}}$ has the same num_y as this top item. If they are not the same, we push $d_{y_{num_{y,i}}}$ into $Stack_{d_y}$; otherwise a mismatch happens and we start the above backtracking process.

[Case 3] $(v_x = V_y = i)$

We check whether $d_{y_{num_{y,i}}}$ has the same num_y as the top item in $Stack_{d_y}$. If they are the same, we push it into $Stack_{d_y}$; otherwise, a mismatch happens, and we start our backtracking process.

Note that when we encounter $q_{x_i}^u$ in the query sequence, we need to shift the search pointer in the data sequence backward to $d_{y_{num_{y,s}}}$ to avoid the semantic false problem (as specified in Section 3).

To illustrate our optimized geometric subsequence matching procedure, let's take query Q_2 as the example. A snapshot of the symbol stacks is given in Figure 11. Detailed algorithmic description can be found in the Algorithm 1(*pc*

[3] Recall in Section 4, $symbol_s^u$ signifies that we need to *resume* the range information of $symbol_s$ so as to cope with the semantic false problem.

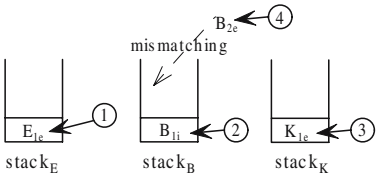

Fig. 11. Stack Status Avoiding Non-existence Query in OptGeoMatching

denotes parent-child relationship and *ad* denotes ancestor-descendant relationship). Firstly, E_{1e} is pushed into $Stack_E$ (Step 1). Since p is in E_e^p in the query sequence, the only item in Figure 9 that satisfies the parent-child constraint is B_{1i}, and is thus pushed into $Stack_B$ (Step 2). K_{1e} is further pushed into $Stack_K$ (Step 3). As p is in K_e^p, B_{2e} is the only possible parent item. However, its number 2 does not conform to the number 1 of the top item B_{1i} in $Stack_B$ (Step 4). Thus B_{2e} cannot be pushed into $Stack_B$, and a mismatch happens. We need to backtrack to B_i and re-start the searching in the data sequence from B_{2i}, returning no satisfactory query answer in the end.

5 Experimental Results

We implement our proposed sequence matching mechanism, *OptGeoMatching*, in C++. We also implemented ViST, and a classical indexing and querying mechanism, XISS [10], for comparison purpose. XISS breaks down the queries into binary twigs and "stitches" them together to obtain the final results. ViST treats both XML documents and XML queries as sequences and obtains the final results by using subsequence matching phase to get preliminary results and post-processing phase to eliminate false answers. We encode the string as they are in ViST and use substring matching algorithm to detect the prefix matching.

We use the B+Tree library in Berkeley DB provided by Sleepycat software. All the experiments are carried out on a Pentium III 750MHZ machine with 512MB main memory. We use disk pages of 8k for Berkeley B+Tree index. To evaluate both the efficiency and scalability of the proposed method, we perform the experiments on both real-world datasets and synthetic datasets.

Experiments on Real-World Datasets

Data Sets

For our experiments, we use public XML databases DBLP [9] and the public XML benchmark XMARK [2].

- DBLP is popularly used in benchmarking XML indexing methods. In the version we used in this study, it has 3,332,130 elements and 404,276 attributes, totally 130,726KB data. The maximum depth of DBLP is 6. The average length of geometric sequence is 39.
- XMARK is widely used in benchmarking XML indexing mechanism with complex nesting structure. In this version we used in this study, it has 1,666,315 elements and 381,878 attributes, totally 115,775KB. The maximum depth of XMARK is 12.

input: **SB-Index**: index of symbol names; **PB-Index**: index of (preorder, size) labels; $\mathbf{Q_{Geo}} = Q_{Geo_1}$, ..., $Q_{Geo_{len}}$: XML query in geometric sequence format; **j**: the j*th* point in Q_{Geo}; **range**: in format of (preorder, size); **len**: length of XPath query sequence.
output: all the matchings of Q_{Geo} in the XML data

```
if j ≤ len then
    if u is in Q_{Geo_j} then
        resume range of corresponding symbol_s, say (n', size');
        OptGeoMatching(n', size', j + 1);
    else
        T ← All the matchings of Q_{Geo_j} in SB-Index;
        R ← All the matchings of T in PB-Index satisfying range;
        for each r_k ∈ R do
            if stack_symbol.isempty() or s is in Q_{Geo_j} then
                stack_symbol.push(r_k);
            else
                if r_k.number = stack_symbol.top().number and i is in Q_{Geo_j}
                then
                    stack_symbol.push(r_k);
                if r_k.number = stack_symbol.top().number and e is in Q_{Geo_j}
                and i is in stack_symbol.top() then
                    stack_symbol.push(r_k);
                if r_k.number != stack_symbol.top().number and e is in Q_{Geo_j}
                and e is in stack_symbol.top() then
                    stack_symbol.push(r_k);
            if r_k = stack_symbol.top() then
                Assume range of r_k is (n', size');
                if size' ≥ len - j then
                    if p is in Q_{Geo_j} and parent constraint is satisfied then
                        if i is in Q_{Geo_{j+1}} then
                            OptGeoMatching(n', size', j + 1) //pc;
                        else
                            OptGeoMatching(n', size', j + 1) //ad;
                    else
                        OptGeoMatching(n', size', j + 1);
                        if i or e is in Q_{Geo_j} then
                            skip to r_h, where r_h.n ≥ (r_k.n + r_k.size)
                    stack_symbol.pop();
else
    output a matching of Q_Geo;
```

Algorithm 1: OptGeoMatching

Table 3. List of XPath Queries

T: title; A: article; AU: author; I: inproceedings; N: namerica; P: payment; PE: personref; PER: person; O: open_auction; C: closed_auctions; CA: closed_auction; B: bidder; BU: buyer;

XPath Queries	Data Sets
Q1: //T[text()="On views and XML"]	DBLP
Q2: //A[./AU[text()="Dan Suciu"]][./AU[text()="Tan"]]	DBLP
Q3: /*//I/AU[text()="Peter Buneman"]/following-sibling::AU	DBLP
Q4: //N/*/P[text()="Cash"]	XMARK
Q5: //*/O[./B/PE[@PER="person0"]][./B/PE[@PER="person23"]]	XMARK
Q6: //C/CA/BU[@PER="person11"]/following-silbing::BU	XMARK

Performance of Query Processing

We used 6 queries on the DBLP and XMARK, and compared the proposed method with ViST and XISS. Table 3 lists 6 different queries for DBLP and XMARK, respectively. The experimental results of using the proposed method, ViST and XISS are shown in Table 4.

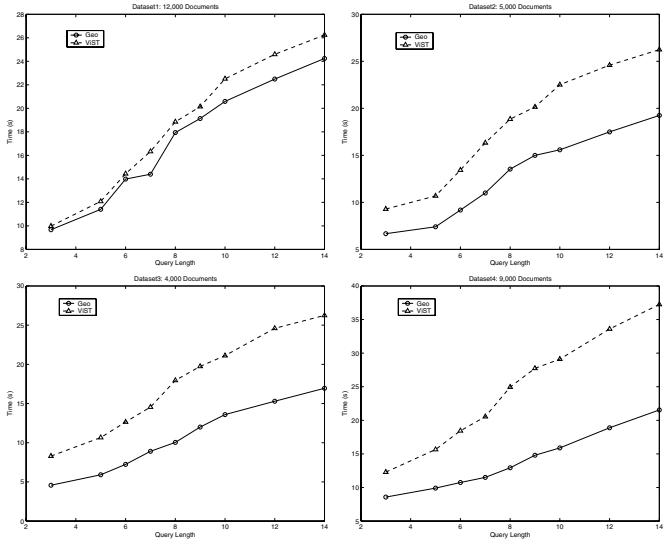

Fig. 12. Queries over Synthetic Data

Q1 is a simple query, "find all the titles with 'On views and XML'". We find out our geometric sequence model performs slightly better that ViST cause there is no instantiation step in geometric sequence model which is inevitable in ViST. Q2 and Q3 are relatively complex queries, respectively, "find all the articles written by 'Dan Suciu' and 'Tan'" and "find the authors co-writting inproceeding papers with 'Peter Buneman'". This time, our geometric sequence model

outperforms ViST because (1) we do not need to perform substring matching in validating and instancing structure-encoded query sequences. The substring matching increases the disk I/O since enormous data is retrieved from the index; (2) there exists no post-processing phase in our proposed method; (3) most importantly, we performs bottom-up query evaluation strategy. Since the number of the nodes with specific authors' names are comparatively small and their ranges are narrow, we can thereupon achieve significant evaluation performance. Q6 is a query which should return no result since there exists only one buyer in one closed auction. The structure expressed by Q6 is a kind of *false alarm*. Again, without exception, our geometric sequence model is significantly faster than ViST because there is no answer during the subsequence matching in our proposed method. We can confidently say that there is no such structure existing in XMARK file, while time-consuming refinement phase has to be called by ViST to eliminate enormous false answers.

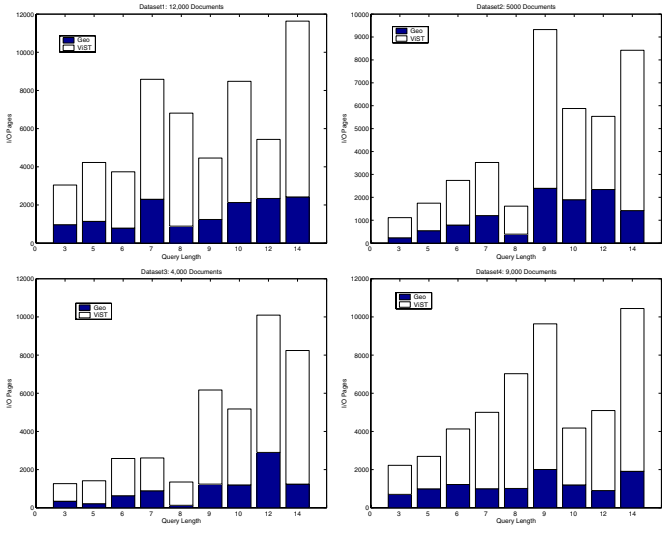

Fig. 13. I/O Performance: Geo vs. ViST

Experiments on Synthetic Data

Datasets

To evaluate the extensibility of the proposed method, we generate our own synthetic datasets. In our experimental environment, there are totally 30,000 documents with 20 different symbols. The maximum depth of our datasets is 16, and maximum fan-out of a node is set to 4. We still use 8KB disk page for B+Tree index and 8-byte integer for pre-order number. We generate geometric sequences directly instead of generating documents.

Table 4. Proposed Method vs. ViST and XISS

Query	Our Method (s)	ViST (s)	XISS (s)
$Q1$	2.81	2.94	7.22
$Q2$	7.14	13.33	319.28
$Q3$	17.49	69.82	612.13
$Q4$	7.86	9.12	467.26
$Q5$	12.27	18.13	392.85
$Q6$	9.73	39.20	729.21

Performance of Query Processing
We set the length of queries to 3, 5, 6, 7, 8, 9, 10, 12, and 14 respectively. All the queries are non-existence queries. To focus on the impact of refinement or post-processing phase in ViST, we do not use queries with content constraints since our bottom-up OptGeoMatching is naturally more superior than top-down ViST. We also do not use queries related to semantic false since ViST can not handle these queries at all. In the scalability test, We found out that the performance of ViST depends on distribution of nodes which are chosen as ancestors or descendants in the queries, referred to as *selectivity*. The high selectivity of both ancestors and descendants generates a considerable number of false answers in ViST if non-existence queries or non-equivalence queries are executed, implying that the query performance of ViST degrades in these cases.

In order to demonstrate the extensibility and stability of our proposed method, we divide the above 30,000 documents into 4 different categories on the basis of the distribution of nodes chosen as ancestors or descendants in the queries.

- $Dataset_1$ (12,000 documents): low selectivity of ancestors and descendants
- $Dataset_2$ (5,000 documents): high selectivity of ancestors and low selectivity of descendants
- $Dataset_3$ (4,000 documents): low selectivity of ancestors and high selectivity of descendants
- $Dataset_4$ (9,000 documents): high selectivity of ancestors and descendants

The results are shown in Figure 12 and Figure 13. We find out that our proposed method performs better than ViST in $Dataset_1$ because the post-processing phase is trivial in dataset1. However, for the rest of the three datasets, our proposed method performs significantly better than ViST since refinement phase requires enormous efforts to eliminate the false answer. Contrasting to ViST, our proposed method performs stably in these three datasets. We notice that even content constraint is not involved in our synthetic data experiments, we can see that OptGeoMatching demonstrates significant disk I/O performance comparing with ViST since top-down ViST is uncertain of its descendants and has to search its full *range* for correct answers. In contrasting to top-down ViST, OptGeoMatching performs a bottom-up subsequence matching and only needs to search a more specific *range* where an ancestor node may exist.

6 Conclusion

In this paper, we report an efficient mechanism for accelerating XPath evaluation steps based on the proposed geometric sequence. A bottom-up holistic subsequence matching algorithm is proposed on the basis of a novel geometric sequence model for XML documents. We demonstrate that our proposed mechanism can significantly improve the current best approach ViST, finding all the correct answers without refinement or post-processing phase with linear size complexity of geometric sequence and guaranteeing the completeness of XPath evaluation without semantic false.

References

1. Aho, A. V., Hopcroft, J. E., Ullman, J. D.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
2. Busse, R., Carey, M., Florescu, D., Kersten, M., Manolescu, I., Schmidt, A., Florian Waas, F.: Xmark an xml benchmark project (2001) http://monetdb.cwi.nl/xml/index.html
3. Chien, S. Y., Tsotras, V. J., Zaniolo, C., Zhang, D.: Efficient complex query support for multiversion XML documents. In EDBT (2002) 161–178
4. Cooper, B., Sample, N., Franklin, M. J., Hjaltason, G. R., Shadmon, M.: A fast index for semistructured data. In The VLDB Conference (2001) 341–350
5. Goldman, R., Widom, J.: Dataguides: Enabling query formulation and optimization in semistructured databases. In VLDB, Springer-Verlag (1997) 436 – 445
6. Gottlob, G., Koch, C., Pichler, R.: The complexity of xpath query evaluation. In PODS, ACM (2003) 179–190
7. Grust, T.: Accelerating xpath location steps. In SIGMOD, ACM Press (2002) 109–120
8. Jiang, H., Lu, H., Wang, W.: Xr-tree: Indexing xml data for efficient structural joins. In 19th International Conference on Data Engineering (2003) 253–264
9. Ley, M.: Dblp bibliography (2004) http://www.informatik.uni-trier.de/ ley/db
10. Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In The VLDB Journal (2001) 361–370
11. Milo, T., Suciu, D.: Index structures for path expressions. In Proceedings of the 8th International Conference on Database Theory (1999) 277–295
12. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: Optimal xml pattern matching. In ACM SIGMOD (2002)
13. Shen, Y., Feng, L., Shen, T., Wang, B.: A self-adaptive scope allocation scheme for labeling dynamic xml documents. In DEXA (2004) 811–821
14. Wang, H.: On the sequencing of tree structures for xml indexing (technical report) (2004) http://magna.cs.ucla.edu/ hxwang/publications/xmlrpt.pdf
15. Wang, H., Park, S., Fan, W., Yu, P. S.: Vist: a dynamic index method for querying xml data by tree structures. In SIGMOD, ACM Press (2003) 110–121