Query Acceleration by Preprocessing Heterogeneous Documents in Distributed Systems

Farhad Moghimifar

Information Security Institute, Queensland University of Technology, Brisbane, Australia (+61) 487673084

Summary
Distributed system consists of multiple databases which are interconnected via a communication network. While these systems have long been used by organizations, researchers have always attempted to solve the problem of making coordination between databases with different structures. This problem may be solved by XML language which can be easily converted to any format. With the increased volume of distributed documents, optimal query processing has become crucially important. XML queries consist of a series of elements which are interconnected under a tree structure. Therefore, finding the pattern between query and document is the central core of query processing. While many methods have been proposed for query processing, they all have the problem of processing the nodes which are not involved in the final answer. Consequently, these methods tend to waste time by processing useless nodes.
The present paper proposes a new method for query processing. This method processes the nodes which are definitely involved in final answer. In contrast to other methods, this method works with a lot of indexes and efficiently answers different kinds of query. We examined the efficiency of this method using famous databanks such as DBLP, TreeBank, XMark as well as balanced and unbalanced random databanks. We also tested simple query, single-branch query and multi-branch query with and without extraction point. The results indicated that the proposed method was more efficient than the existing popular methods in terms of the number of processed nodes, used memory and execution time.
Key words:
Distributed database, query, optimal processing, document guidance and guide index

1. Introduction

With the increased volume of heterogeneous documents, XML query processing has become crucially important. These documents have a tree structure. In other words, structure and content are close to each other in such documents. The following is an example of XML query:
ϕ1: S⁄⁄B [T='Value']
ϕ1 query represents the structural relationship between S, B and T on the one hand and the value of T on the other. At present, old indexes such as Tree+B have acceptable efficiency in XML documents. But the problem of structural pattern matching in these queries has been addressed by many researchers.

So far, different guides have been introduced for document structure, such as XML SCHEMA, DTD, STRUCTURAL SUMMARY and DATAGUIDE. These guides are used for query guidance [1-13]. Moreover, many path indexes such as Index k(A), Indexl, APEX, ToXin, Fabric DataGuide Strong and F&B have been introduced. These path indexes are used to index the paths and answer path queries more quickly [3-9]. Furthermore, there are many methods for answering structural queries such as holistic twig join and structural join. These methods do not make optimal use of guides and involve the nodes which are not present in final answer [14-21]. In answer to ϕ1 query by structural join method, for example, the query is converted to a number of binary links (S//B & /T). Also, a large volume of middle data is produced by decomposing the query into parent-child or ancestor-descendant relationships. Holistic twig join attempts to remove this problem by not decomposing the query. This method is efficient for ancestor-descendant relationships but not for parent-child relationships. Moreover, this method processes all nodes present in ϕ1 query.
In the method proposed in [22], only leaf nodes are involved in the query (in ϕ1 of T node) and each node has a code. Ancestor information is obtained by encoding the code prefix of leaf node. This method compares the nodes blindly and without any guide, so it is not efficient for large documents.
In the present paper, we aim to:

- Process only leaf nodes which are present in the query.
- Minimize the number of comparisons between leaf nodes, so that each comparison ideally produces one part of the answer.

We are going to propose an optimal method for query processing. This method uses query guide and pattern matching guide and processes only those nodes which definitely produce some part of the final answer.

2. Proposed Method

The proposed method is a combination of path index and containment join methods. As shown in the Figure 1, this method has three steps (Figure 1).

Manuscript received September 5, 2016
Manuscript revised September 20, 2016

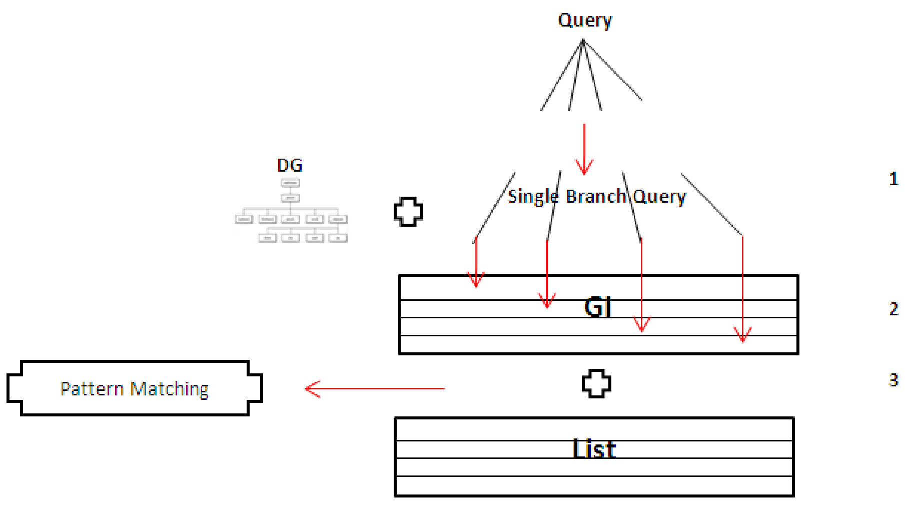


Fig 1. Proposed Method

2.1 Encoding

We encoded the documents using Dewey decimal system [25]. In this system, if node U is nm child of node V, code number of node U would have the same code of node V as prefix followed by n. For example, if Dewey code is V=<1 ⁄3 ⁄7> and node U is the fifth child of node V, the code would be U=<1⁄3⁄7⁄5>.

2.2 Document Guidance

Query guide is basically like document schema. DG is highly correlated with DTD and XML Schema of the document, represents the schema of structural documents and the general relationships between its elements, and is scarcely correlated with the volume and bigness of document data. The structure and size of document guidance is normally unchanged or undergoes very few changes.
DG volume is far smaller than the volume of real data. For example, TReeBank [27], DBLP [28] and Xmark [29] with the respective sizes of 130, 897 and 532 megabytes have a schema volume of 3, 4.2 and 2.8 kilobytes.
DG of a document is scarcely correlated with the size of document data. In [26], DG has been tested for Sports and Synthetic banks using Strong Dataguide method which has a relatively high volume compared to other similar methods. These two banks were added by 30695 and 375449 nodes respectively, but their DGs were added by only 2 and 12 nodes.

2.3 Path Index

YAPI [30] is the best choice for meeting the above-said two requirements and is the fastest and cheapest choice for answering single-branch queries.

2.4 Connection Point

If A and B are two branches of the query with the traveled paths of ɑ1⁄ɑ2⁄…⁄ɑj⁄ɑx1⁄…⁄ɑn and ɑ1⁄ɑ2⁄…⁄ɑj⁄ɑx2⁄…⁄ɑm and ɑx1 ≠ ɑx2, then the connection point of the two branches is ɑ1 ⁄ɑ2 ⁄… ⁄ɑj. It should be noted that the symbol / between query elements does not refer to parent-child relationship and may also be interpreted as /, //, *, etcetera (Figure 2).

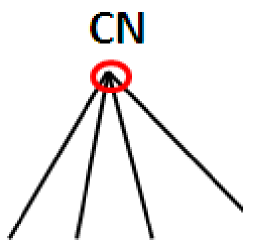


Fig.2 An example of connection node

2.5 Decomposition

If ɸ is a multi-branch query with n connection nodes and m branches, it is decomposed into single-branch queries of Sɸ1,….,Sɸm after breaking down. In this case, each Sɸi (from root to leaf) is one of the query branches and both Sɸi and Sɸj have the same prefix (from root to one of the connection nodes). Total number of these connection nodes may vary. In ɸ2 query, for example, since point A is the connection node of two branches of A//B and A//C/D, so ɸ2 query is decomposed into two single-branch queries; i.e. A//B and A//C//D.

ɸ2: A[.∕∕B][.∕∕C∕∕D];

2.6 Solution

In this step, we compared all single-branch queries with document DG. Since the document was encoded by Dewey method and lower nodes had heterogeneous information of upper nodes (the path traveled from the root), we only needed to keep query leaves for each branch. As the result, we obtained a list of DG points for each single-branch query. Figure 3 illustrates the address of these points from root to node.

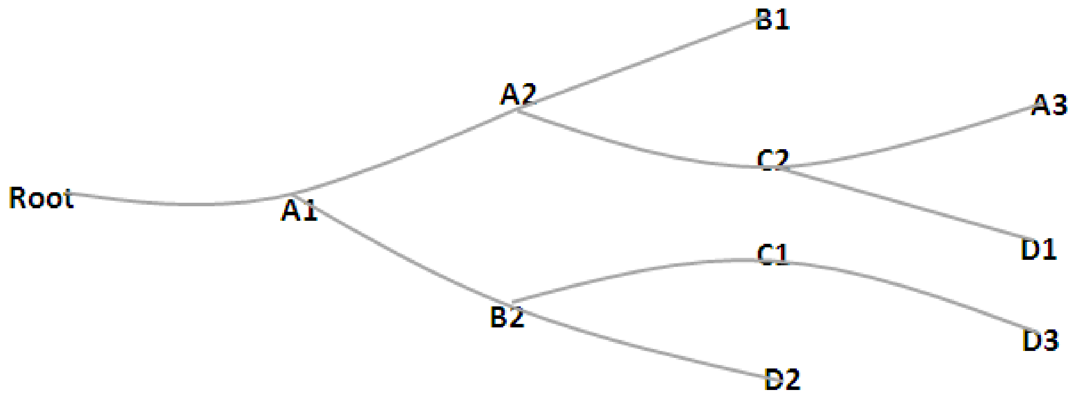


Fig. 3 An example of DG

These points have absolute paths of ROOT ∕A1 ∕A2 ∕B1 and ROOT ∕A1 ∕B2 for A//B branch and ROOT ∕A1 ∕A2 ∕C2 ∕D1 and ROOT ∕A1 ∕B2 ∕C1 ∕D3 for A//C//D branch.

Primary definition of guide Index: G1 is a three-column table with the first two columns being leaf nodes of each query branch in DG and the third column being the connection point between two nodes. Therefore, each record of this table represents an action called pattern action.

Pattern Finding: In pattern finding, we compared two or more nodes in the documents in order to achieve a part of the answer. After the nodes were decomposed into single-branch queries in step 1 and the leaf nodes of single-branch queries in DG were obtained, we had to obtain the connection node between the leaves. Running single-branch queries on DG resulted in a list of nodes for each single-branch query. In this step, we had to make a paired comparison between the elements of each branch. If A and B were two members with the traveled paths of ɑ1 ∕ɑ2 ∕… ∕ɑj ∕… ∕ɑn and ɑ1 ∕ɑ2 ∕… ∕ɑj ∕… ∕ɑm and ɑj was the connection node between two branches (i.e. if both groups had the same traveled path from root to connection node of the query), we would add a record together with connection node.

The following represents the initial algorithm of GI production for a two-branch query. This is an initial code and is specific to two-branch queries. The algorithm for more complicated queries has been represented in the following paragraphs.

```
Input: ɸ as Query Pattern
Output: GI as Guide index
1: Let A and B the two leaves of ɸ
2: Let CN = Connection N between A and B
3: Let AL = list of DG for match A branch
4: Let BL = list of DG for match B branch
5: for each an ∈ AL do
6:     for each bn ∈ BL do
7:         for each $CN_1$ in an , $CN_2$ in bn do
8:             if $an._{Prefix}(CN_1) = bn._{prefix}(CN_2)$ then
9:                 $GI._{addREC}(an, bn, CN_1.level))$
```

A and B are two leaf nodes of the query. Line 3 specifies the connection node of two branches. AL and BL specify the lists of A and B nodes. These nodes are the same nodes found for each query branch.

Each record has three fields for producing final results. The first two columns have two nodes and the third column is the connection point level between two nodes. The real nodes of the document are arranged in the lists according to their Dewey codes. Now, we have to compare the lists of both nodes. If the two nodes have the same prefixes until connection point level, they are involved in the answer. This process continues until one of the two lists reaches the end. This action is called pattern matching.

Let’s consider the record of <B1, D, 2>. B1 and D1 are two nodes in DG. Let’s suppose that the following two lists represent these two nodes. Root level has been set on zero.
B1-list = {1/3/1, 1/3/6, 1/7/1}
D1-list = {1/2/2/1, 1/3/3/1, 1/3/5/7, 1/6/2/2, 1/7/1/2}
Since level 2 is the connection point level between two nodes, only those elements with the same prefix up to level 2 of Dewey code can have a successful pattern matching. The following list produces the output.
OutputList = {(1/3/1, 1/3/3/1), (1/3/1, 1/3/5/7), (1/3/6, 1/3/3/1), (1/3/6, 1/3/5/7), (1/7/1, 1/7/1/2)}
L is the third field of the table and represents the connection point level. This is the level in which the elements of the two lists must be compared.

```
while NOT(one of L1 or L2 reach the end) do
    for each a in L1 , b in L2 that
    a.prefix(L) = b.prefix(L) then
        (a, b) → output
    else if N1.prefix(L) > N2.prefix(L) then
        N2= L2.Next()
    else
        N1= L1.Next()
```

The second and third lines represent the nodes which are equal until connection point level and are involved in pattern matching answer. In other words, these nodes have experienced a successful pattern action. For the nodes which are not equal to any node until this level, the list cursor of the smaller node has to prepare the next element for processing.

3. Method Development

3.1 Connection points with more than two sub-branches

For example, let’s consider ϕ3:
ϕ3= //A[./C][./D]/B ;

3.2 The second definition of guide index

G1 is a M+1 column table for a connection point with M sub-branches. The first to Mm columns of the table are the branch leaves and the last column is the level shared by all nodes.

```
For each a1∈ A1, a2∈A2,…,.an∈An do
    IF a1.prefix(L)= a2.prefix(L)=……= an.prefix(L) then
        Add( a1,a2,…,an) to output
    Else
        Min(a1,a2,…an).Next()
```

These lines make a comparison between the list elements. The elements would be included in the final answer only if they have the same prefix up to this level. For each unsuccessful pattern, the smallest element must go to the next node.

3.3 Connection points with more than two sub-branches

As the first change, a GI_model must be used instead of GI.
Definition of GI_Model: A set of n GI for a query with n connection nodes which indicate the correlation between them.
The Second change must be made in node processing order for producing index. You can see this change in the following pseudocode:

```
GI1→ GI 2→…→ GI n
Pattern_Matching_Proc(GI_1,1 )
```

The above pseudocode acts in bottom-up fashion. In other words, it first processes the lower connection points in ϕ tree. Where a correlation exists between several GIs, the process is started from the first GI (second line). This pseudocode uses a recursive procedure called Pattern_Match_Proc which complies with processing order.

3.4 Optimization

As mentioned earlier, guide index is used as the guidance for processing documents. In other words, guide index prevents the processor from blind processing. Producing a guide index merely requires the document schema. The real processing action needs access to real nodes only in the third step. In other words, from each successful pattern action between the elements, we only consider its extraction point as the query answer. For example, node a is the extraction point and the other three elements are query conditions and are merely used for testing the correctness of extraction points.

3.5 Deletion of Repeated Nodes

After comparing two nodes, we sometimes reach a node which has been previously produced. This means that the operator has not produced a new answer. In ϕ3 example, (b1 Δ d3) U (b1 Δ d1) may give repeated a1 nodes because b1 acts independently in both comparisons. So we can write the expression thus:
(((b1 – (b1 Δ d1)) Δ d3)

This means that b1s can be compared with d3 only if they have not previously produced an answer by d1 comparison.

3.6 Deletion of useless patterns

Sometimes we do not need to compare two nodes because the existence of one node confirms the existence of another. In (d3 Δ b2), for example, the existence of d3 confirms the existence of d2. We can delete these cases from the graph and consider all d3s as semantic answer. On the other hand, where all d3s are considered as final answer, we delete all d3s from the graph in the algebraic expression of pattern matching guide.

3.7 Merging several equations into one

Sometimes we make only one comparison to reach two or more answers. In (b1 Δ d1), for example, we reach two nodes of α1 and α2 in each comparison.

3.8 Order of queries with more than two sub-branches

Sometimes we compare three or more nodes to reach one answer and select the node which has an equivalent in all lists. In such cases, we can perform the intersection operator in the form of ((α1 Δ α2) ….) Δ αn) which will be α1 < α2 <……< αn in the order of node number in lists.

Deletion of repeated extraction points: We can develop pattern matching guide so that it includes extraction points too. For example, since we searched α in the previous query, we can change the pattern matching guide so that two numbers appear above each equation (output number and shared level number). This changes the definition of records in pattern matching guide to E=<n , m ,(L1 ,E1),….,(Ln ,En)>.

NOT Operators

In heterogeneous databanks, there exist many queries with NOT branch. For example, ϕ4 may be presented as follows:

ϕ4: A//B//[NOT(. //C//D)];

Single-Branch Queries: Single-branch queries are the queries which have a NOT operator. As mentioned earlier, these queries have positive and negative sections.

Method: In this method, we first have to find the positive and negative leaves and then the negative leaves delete positive leaves. The rest of positive leaves will have a positive answer. This method has bottom-top fashion. In other words, we have to start from lower nodes towards upper nodes. Each node in the lower list deletes one node from the upper list. In such queries, we only have two nodes, with the lower nodes being negative leaves and the upper nodes being positive leaves.

```
do while (α AND b)
    if      b.prefix(L) < α.prefix(L) then
            b=B.Next()
    elseif  b.prefix(L) > α.prefix(L) then
            α=A.Next()
    else
            Delete b from B
    End if
OUT_PUT= B
```

Two-branch queries with NOT operator: Multi-branch queries are the queries which have multiple branches, with every two branches connecting in one point called connection point.

Method: The method is like the previous one, with the difference that it only searches for positive leaves of the query. In this method, likewise, the answers are divided into two groups:

• The answers which lack negative sections

• The answers which must be checked for the lack of negative section

For the second group of answers, we have to find the leaves of the negative section. Then, the leaves of negative section delete the leaves of positive section. The rest of positive leaves will be included in the answer. This method first lists all negative sections and then cuts the level shared by two branches from the number of negative nodes. After listing the positive sections, we have to delete the nodes which start from this number.

Nested NOT operator: NOT operator usually operates in single-branch or multi-branch fashions. However, it rarely operates inside another NOT operator in one or more steps. For example, see the following query:

ϕ6: A//B[NOT .//C (NOT .//D)]

Method: This method has a bottom-up fashion too. After listing the leaves of positive and negative sections, we start from the nodes in the lower lists and proceed to the nodes in upper lists. Each node (whether positive or negative) deletes a node from the upper list. The remained nodes in the highest list will be the final answer.

```
N-U proc (αi , αi-1)
    Do while (αi AND αi-1)
        If  αi.prefix(Li,i-1) < αi-1.prefix(Li,i-1)
            αi=Ai.NextN
        elseif  αi.prefix(Li,i-1) > αi-1.prefix(Li,i-1)
            if (i-1 = 1 )then // the highest level
                output=αi-1
            else
                N-U Proc(αi-1 , αi-2)
                αi-1=Ai-1.NextN
            end if
        else
            Delete αi-1 from Ai-1
        End if
    END DO
Output = A1
```

Useless Nodes
Useless nodes are the nodes which do not give any answer or produce repeated answer.

Jumping the nodes which are not involved in the final answer: The nodes not involved in the final answer are useless nodes and must be skipped.

Jump: If node A is **ɑ1 ⁄ɑ2 ⁄… ⁄ɑj** ⁄.... ⁄ɑn and we want to jump the level C, the next node will be B on the condition that B is the smallest node to be bigger than A and has a different prefix form node A until level C.

Result: So, it is better to have the expression of N.J(LCN) instead of N.next() in pseudocode lines. By LCN is meant the connection between two branches.

Jumping the nodes which produce repeated answer:
Jump: If node A is ɑ1 ⁄ɑ2 ⁄… ⁄ɑj ⁄.... ⁄ɑn and we want to jump the level C, the next node will be B on the condition that B is the smallest node to be bigger than A and has a different prefix form node A until level C. Therefore, we have to add the following code before FOR circle for each node of the list which has had a successful pattern action.

```
IF $L_i.N_j$ has Successful matching Process then
        $L_i$ .J ($L_{Etraction Point}$)
EndIF
```

4. Result

In this part, we are going to evaluate the guide method. In doing so, we compared this method with two popular methods as mentioned in reference [21] as the representative of Twing group and reference [24] as the representative of TJfast group. First, we should explain the system selected for the purpose of comparison:
The set of selected data:
We used four datasets of TreeBank, DBLP and XMark, which are very popular in the world of XML (Table 1).

Table 1: Specifications of Datasets

	XMark	*DBLP*	*Treebank*
Data size(MB)	582	130	82
Nodes(million)	8	3.3	2.4
Max/Avg depth	12/5	6/2.9	36/7.8

RandomDataSet: To make this dataset, a schema with the depth of 12 and maximum number of 10 children for each node is randomly produced. The tags of this graph are the words ɑ, b, c, d, e, f. (Table 2).

Table 2: Specifications of dataset random table

Dataset name	*Data size(MB)*	*N(million)*	*Depth*	*Wight of N*
Random dataset	890	8.3	12	10

Array of Inputs: Sometimes we randomly produce an array of inputs in order to run the method. The maximum number of these arrays is 100,000 and the length of Dewey code of each element is 5. We need some jumps in the levels 2, 3 and 4 of these Dewey codes. The maximum number of children of each node is assumed to be 9.

Dewey Code Saving Method: In this method, an array of numbers (.\1.2.3.4") is divided by dot. Table 3 represents the average volume of data maintenance for the tested datasets (Table 3).

Table 3: Dewey code size

	XM	*DBLP*	*TB*	*Random dataset*
Original Dewey(MB)	56.2	18.1	22.8	61.3

Hardware environment: AMD Athlon 7750 dual core 2.7 GHZ Processor; 2G Memory; 160G HDD ;Software environment: Windows 7, 32 bit.

We test five queries with unique specifications using the mentioned two methods [13] and guide index (Table 4).

Comparing guide index method with leaf method

The number of called elements: In both methods, only ϕ leaves are processed, but there are two fundamental differences:

• Leaf method first investigates whether a node meets the single-branch condition. In our method, all nodes which are accessed are a member of one of the query branches.

• Leaf method attempts to reach the answer by direct comparison of the leaves of each branch and is likely to compare many leaves with no structural relationship between them. But our method only compares the leaves with structural relationship. Many nodes do not need any access because they do not have a counterpart in other branches. This difference is more noticeable in father-child queries.

The size of main memory used: In leaf method, we need to save a number of nodes to compare two groups, because they might constitute a part of answer through comparison with another group. Leaf method attempts to reach the answer through direct and blind comparison of the nodes, while our method does not need to save any middle data because node processing manner and answering method are identified in GI.

Execution time: As you can see in Table 4, the execution time in leaf method is longer than in guide index method, because leaf method needs to decode each input data and wastes time for each node.

Table 4: Results in comparison with [24]

	Query	*No of elements Read (k)*		*Size of disk scanned (KB)*		*Exe Time(ms)*	
		GI	*[24]*	*GI*	*[24]*	*GI*	*[24]*
Q1	/site/people/person/gender	56	57	967	1005	387	254
Q2	/S[//VP/IN]//NP	510	595	3154	6251	4005	11245
Q3	/S/VP/PP[IN]/NP/VBN	101	195	1997	2025	1674	2854
Q4	//article[//sup]//title//sub	32	44	117	425	951	1874
Q5	//inproceedings//title[//i]//sup	24	35	114	674	847	1628

Comparing guide index method with branch method
Table 5 represents the queries. Reference [21], like other methods in its group, accesses all ϕTP nodes to answer the query. Compared to branch method, our method accesses more nodes for processing the query. However, it offers a better execution time than branch method because it doesn't needs to convert the code to the name of elements.

Table 5: Results of comparison with [21]

	Query	*No of elements Read (k)*		*Exe Time(ms)*	
		GI	*[21]*	*GI*	*[21]*
Q1:	//dblp/article[author]/[//title]//year	8.5	22	11457	20145
Q2:	//people/person[//address/zipcode]/profile/education	0.2	1.1	147	457
Q3:	//S/VP/PP[IN]/NP/VBN	3	14	419	13985

Random Data
Single-Branch Queries: We executed eight single-branch queries of A1, A2, …, A8 with the respective lengths of 2, 3 … and 9 using [21] and GI methods. All queries are partial and start with //. As you can see in the figure, the more the number of single-branch query nodes, the less the number of elements accessed in GI. Let's assume that ϕ1 and ϕ2 are the queries with the respective lengths of N1 and N2. Since GI first executes the query on DG, less answers will be found in DG for ϕ2. and consequently less nodes are accessed in the document.
Multi-branch queries: We compared the queries of A1, A2,…, A4 with 2, 3, 4 and 5 branches using Gi and [24] methods. In both methods, the increased number of branches resulted in the increased number of accesses, but the growth rate of GI method was far less than [24] (Table).

Table 5: Single-branch and multi-branch queries

	No of elements Read (mill)		
	GI	*[21]*	*[24]*
S1	1	1.9	---
S2	0.9	1.12.2	---
S3	0.4	142.3	---
S4	0.3	2.2	---
S5	0.23	3.4	---
S6	0.22	3.9	---
S7	0.11	4.7	---
S8	0.08	5.6	---
M1	0.9	---	1.5
M2	1.2	---	2.8
M3	2.1	---	4.2
M4	2.8	---	6.3

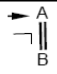
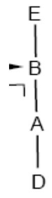
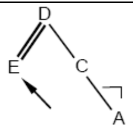
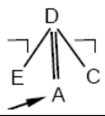
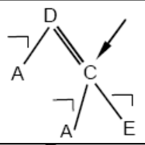
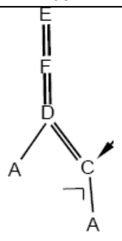
Optimization of guide index: In this part, we tested the queries with specified extraction points and compared GI and OptimalGI methods with a final number of answer data. The criterion was the number of comparisons made for reaching the answer. The right column indicates the number of real extraction points (Table 6).

Table 6: The number of called nodes

	GI	*OptimalGI*	*ETRACTION POINT*
ϕ1	6807	4251	1800
ϕ2	32567	23156	11205
ϕ3	483989	234801	125098

NOT Queries: In this part, we tested our method with PathStack [16] and TwingStackList [31] methods in terms of NOT queries. In doing so, we executed six ϕTP with distinct structures and specifications on this dataset (Table 7).

Table 7: NOT queries with extraction point

1.1.1.5	1.1.1.4 [16]	1.1.1.3 [31]	1.1.1.2 GI	1.1.1.1 Number of ETRACTION POINT
ϕ(a): 	63,200	63,200	28,900	12,300
ϕ(b): 	132,221	111,798	20,000	8,050
ϕ(c): 	115,001	84,993	28,200	211
ϕ(d): 	173,291	114,992	87,769	5,912
ϕ(e): 	100,251	93,075	17,900	890
ϕ(f): 	215,011	183,700	9,000	51

In (ɑ) query, the difference between the numbers of processed nodes is explained by query processing on DG. All the three methods must check two lists of A and B, but they differ in how to process the lists. Our method jumps the nodes which produce repeated answer. In (b), (c) and (f) queries, only a NOT operator has been used. But our method checks only two lists in (b) query and three lists in (c) and (f) queries in order to reach the answer. Consequently, our method processes less nodes than two other methods do. This advantage is explained by TJFast which only searches the leaves (compared with the other two groups which search positive queries). This difference is not significant in (d) query. The only difference between our method and the other two methods is that our method has a bottom-up fashion, in which many of C nodes are destroyed by lower negative A and E nodes and consequently a smaller linkage is made between A and C. The same is true in (e) query.

We compared the queries represented in Table 8 in terms of execution time using three datasets of TreeBank, DBLP and XMark.

Table 8: Execution time in NOT queries

1.1.1.8	1.1.1.7	1.1.1.6 Exe Time (s)		
		1.1.1.11 GI	1.1.1.10 [16]	1.1.1.9 [31]
ϕ(1):	∕∕ɑrticle[.∕∕sup]∕∕T[**NOT**.∕∕sub]	2	4	5
ϕ(2):	∕∕ɑrticle[**NOT**(.∕∕sup)]∕∕T[**NOT**.∕∕sub]	2.2	3	10
ϕ(3):	∕∕Text[**NOT**(.∕bold)]∕emph[.∕keyword]	11	23	28
ϕ(4):	∕∕Text[**NOT**(.∕bold)]∕emph[**NOT**(.∕keyword)]	21	29	51
ϕ(5):	∕∕S∕VP∕PP[.∕IN]∕NP[**NOT**(.∕VBN)]	14	29	32
ϕ(6):	∕∕S∕VP[**NOT**(.∕PP[.∕IN]∕NP(**NOT**[.∕VBN]))]	25	42	62

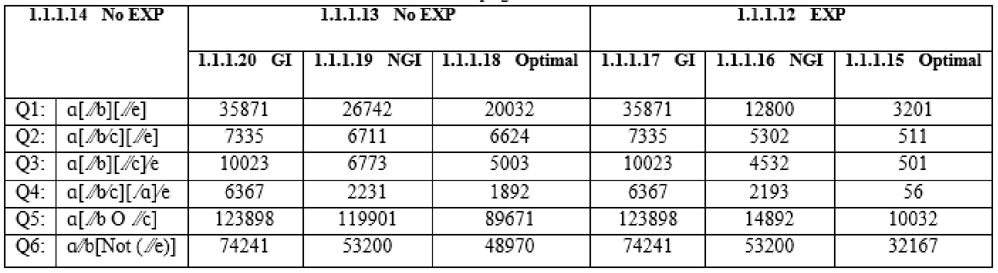
As you can see in the queries (1) and (2), our method does not differ significantly with the other two method, which is explained by the simple structure of DBLP. But the difference is more significant in queries 2-6, particularly in the queries 3 and 5 where nested NOT operators have not been used.

The impact of jumping the useless nodes in guide index method

We compared the original GI with new GI. The advantage of new GI over original GI is explained by the fact that the former jumps the useless nodes. The new GI indexes the query leaves by B+Tree method but the original GI processes the nodes in sequential manner. Table 9 contains the results for six queries executed with these two versions:

The first four queries have AND branch. In AND queries, the majority of useless nodes are not involved in the final answer. This is particularly seen in the queries with a large number of leaves which produce fewer answers (see the number of useless nodes processed for ϕ3 and ϕ4 for example). On the other hand, in the queries where the number of extraction points is more than the number of branches, there are more useless nodes which produce repeated answer (see ϕ1 query for example). In the queries with OR operator, there are obviously more useless nodes which produce repeated answer. But in NOT queries, there exist more useless nodes which are not involved in the final answer and are fruitlessly processed, because each negative leaf deletes one positive leaf. In ϕ6 query, for example, each leaf (e) deletes a leaf (b). These leaves deleted are not involved in the final answer and it is very likely that several leaves (e) delete one leaf (b).

Table 9: Jumping useless nodes



1.1.1.14 No EXP		1.1.1.13 No EXP			1.1.1.12 EXP		
		1.1.1.20 GI	1.1.1.19 NGI	1.1.1.18 Optimal	1.1.1.17 GI	1.1.1.16 NGI	1.1.1.15 Optimal
Q1:	ɑ[⁄⁄b][⁄⁄e]	35871	26742	20032	35871	12800	3201
Q2:	ɑ[⁄⁄b⁄c][⁄⁄e]	7335	6711	6624	7335	5302	511
Q3:	ɑ[⁄⁄b][⁄⁄c]⁄e	10023	6773	5003	10023	4532	501
Q4:	ɑ[⁄⁄b⁄c][⁄ɑ]⁄e	6367	2231	1892	6367	2193	56
Q5:	ɑ[⁄⁄b O ⁄⁄c]	123898	119901	89671	123898	14892	10032
Q6:	ɑ⁄⁄b[Not (⁄⁄e)]	74241	53200	48970	74241	53200	32167

Optimization of document schema: We performed optimization for these six queries based on random document schema. Table 10 represents the results. ϕ1 and ϕ5 queries have simpler conditions and produce more answers. Here, it is assumed here extraction points of the query are specified (Table 10).

Table 10: Optimization on document schema

		1.1.1.12 EXP		
		1.1.1.15 GI	1.1.1.14 NGI	1.1.1.13 Optimal
Q1:	ɑ[⁄⁄b][⁄⁄e]	35871	11745	3201
Q2:	ɑ[⁄⁄b⁄c][⁄⁄e]	7335	1452	511
Q3:	ɑ[⁄⁄b][⁄⁄c]⁄e	10023	957	501
Q4:	ɑ[⁄⁄b⁄c][⁄ɑ]⁄e	6367	175	56
Q5:	ɑ[⁄⁄b O ⁄⁄c]	123898	12478	10032
Q6:	ɑ⁄⁄b[Not (⁄⁄e)]	74241	42571	32167

Execution Time: We executed the query using both versions. Table 11 contains the results. In new GI version, we didn't consider the time of optimization on document schema.

Table 11: Execution time with optimization of document schema

1.1.1.18		1.1.1.17 GI	1.1.1.16 NGI
Q1:	ɑ[⁄⁄b][⁄⁄e]	3854	2221
Q2:	ɑ[⁄⁄b⁄c][⁄⁄e]	1245	954
Q3:	ɑ[⁄⁄b][⁄⁄c]⁄e	957	415
Q4:	ɑ[⁄⁄b⁄c][⁄ɑ]⁄e	712	275
Q5:	ɑ[⁄⁄b O ⁄⁄c]	6217	3004
Q6:	ɑ⁄⁄b[Not (⁄⁄e)]	5412	2745

Queries: We selected four queries of Table 12 for test. Each of these queries has unique specifications. Query (ɑ) is based on father-child relationship and queries (b) and (c) are based on ancestor-descendant relationship. Query (b) has been selected in a way that a short distance exists between CN point and the leaves. But the query (b) has been selected in a way that a long distance exists between CN point and the leaves. Query (d) is a three-branch query with AND operator between the branches.

Balanced random dataset: Balanced dataset is the one in which each node has a fixed number of children. The tree of this dataset is an isosceles triangle. The number of nodes in each level is obtained by the following equation:

I= ɑ* (I-1)

Where, I is level number and ɑ is the number of children of each node which are fixed in the tree.

Table 12: Balanced dataset

1.1.1.22 No EXP		**1.1.1.21 GI**	**1.1.1.20 NGI**	**1.1.1.19 [13]**
Q1:	a/*/b[/c]	38457	19475	28473
Q2:	a//b//c[//d]	125485	72145	98452
Q3:	a//e[//b//c//d]	78451	22694	58417
Q4:	a//b[//c][//d]	99416	59729	29746

Unbalanced random dataset: This dataset is defined in a way that the left side of the tree has more weight. For example, if a, b and c are three sibling nodes with a being in the left side, b being in the middle and c being in the right side, the following relationship exists between the children of each node:

ɑ>b>c

The same is true for all nodes of the tree. In other words, the lower part of each node tree in the left side is bigger than the lower part of each node tree in the right side (Table 13).

Table 13: Unbalanced dataset

1.1.1.26 No EXP		**1.1.1.25 GI**	**1.1.1.24 NGI**	**1.1.1.23 [13]**
Q1:	a/*/b[/c]	19457	14251	19421
Q2:	a//b//c[//d]	81247	59417	78451
Q3:	a//e[//b//c//d]	49328	22485	44157
Q4:	a//b[//c][//d]	68241	38451	28471

Random dataset with many children and few similar tags: refers to a tree in which each node has many children with different tags and very few children with similar tags. Such trees have very complicated schemas. If we want to make the schema less complicated, the volume will exceed the standard level. These trees are fat in the waist (Table 14).

Table 14: Random dataset with many children and few similar tags

1.1.1.30 No EXP		**1.1.1.29 GI**	**1.1.1.28 NGI**	**1.1.1.27 [13]**
Q1:	a/*/b[/c]	19241	16547	19457
Q2:	a//b//c[//d]	78555	69417	94218
Q3:	a//e[//b//c//d]	46218	32417	48753
Q4:	a//b[//c][//d]	54198	51488	41579

Random dataset with many similar children: refers to a tree in which each node has many tags with abundant similar tags. This tree has a very simple schema and low volume. Such trees are fat in their lower part and in their leaves (Table 15).

Table 15: Random dataset with many similar children

1.1.1.34 No EXP		**1.1.1.33 GI**	**1.1.1.32 NGI**	**1.1.1.31 [13]**
Q1:	a/*/b[/c]	38411	14257	22451
Q2:	a//b//c[//d]	124575	46874	78416
Q3:	a//e[//b//c//d]	78459	22485	45241
Q4:	a//b[//c][//d]	101698	42699	23477

5. Conclusion

In this paper, we explained the methods and languages used for query processing, classified the existing methods, and explored the advantages and disadvantages of each method. As an improvement on earlier methods, we proposed the guide index method which has three steps as follows:

1. First step: simplification of the query and reduction of search domain

2. Second step: production of guide index as the guidance for query processor
3. Third step: processing document nodes based on the guide index produced in the previous step

In doing so, we used the datasets of TreeBank, DBLP and XMark and selected Dewey system for data saving. We developed this method to be efficient for multi-branch and complicated queries as well. Then we expressed the method in the form of algebraic expressions and optimized it using the expressions and concept of extraction points. Next, we attempted to develop this method for the queries with NOT operator. The results indicated that the proposed method was more efficient and flexible than other methods. In the process of optimization, we faced useless nodes. Useless nodes are those nodes which produce no answer or repeated answer. By using an index, we managed to jump these nodes and significantly reduce the number of processed nodes. In the next step, we introduced a level index to increase the efficiency and jump more useless nodes.

GI method was found to be significantly more efficient than branch and leaf methods in three main parameters:

5.1 The number of called elements:

Thanks to GI, the proposed method only compares the leaves with a structural relationship. Many nodes don’t need any access because they lack a counterpart in other branches. This difference is more significant in parent-child queries.

5.2 The size of main memory used:

In other method, we need to save a number of nodes to compare two groups, because they might constitute a part of answer through comparison with another group. Leaf method attempts to reach the answer through direct and blind comparison of the nodes, while the proposed method doesn’t need to save any middle data because node processing manner and answering method are identified in GI.

5.3 Execution time:

The execution time in other methods is longer than in guide index method, because other methods blindly searches and decodes each input data and wastes time for each node.

References

[1] Garofalakis, Minos, Aristides Gionis, Rajeev Rastogi, Sridhar Seshadri, and Kyuseok Shim. "XTRACT: a system for extracting document type descriptors from XML documents." In ACM SIGMOD Record, vol. 29, no. 2, pp. 165-176. ACM, 2000

[2] Nestorov, Svetlozar, Jeffrey Ullman, Janet Wiener, and Sudarashan Chawathe. "Representative objects: Concise representations of semistructured, hierarchical data." In Data Engineering, 1997. Proceedings. 13th International Conference on, pp. 79-90. IEEE, 1997

[3] Goldman, Roy, and Jennifer Widom. "Dataguides: Enabling query formulation and optimization in semistructured databases." (1997)

[4] Milo, Tova, and Dan Suciu. "Index structures for path expressions." In International Conference on Database Theory, pp. 277-295. Springer Berlin Heidelberg, 1999

[5] Cooper, Brian F., Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. "A fast index for semistructured data." In VLDB, vol. 1, pp. 341-350. 2001

[6] Rizzolo, Flavio, and Alberto O. Mendelzon. "Indexing XML Data with ToXin." In WebDB, vol. 1, pp. 49-54. 2001

[7] Kaushik, Raghav, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. "Exploiting local similarity for indexing paths in graph-structured data." In Data Engineering, 2002. Proceedings. 18th International Conference on, pp. 129-140. IEEE, 2002

[8] Chung, Chin-Wan, Jun-Ki Min, and Kyuseok Shim. "APEX: An adaptive path index for XML data." In Proceedings of the 2002 ACM SIGMOD international conference on Management of data, pp. 121-132. ACM, 2002

[9] Kaushik, Raghav, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. "Covering indexes for branching path queries." In Proceedings of the 2002 ACM SIGMOD international conference on Management of data, pp. 133-144. ACM, 2002

[10] Ley, Michael. "DBLP computer science bibliography." (2005)

[11] Liu, Peng, Weiwei Sun, Jian Zhang, and Baihua Zheng. "An automaton-based index scheme supporting twig queries for on-demand XML data broadcast." Journal of Parallel and Distributed Computing 86 (2015): 82-97.

[12] Chamarthy, Ravi Chandra. "Self-parsing XML documents to improve XML processing." U.S. Patent 9,087,140, issued July 21, 2015

[13] Xiaoping, Ye, Lin Yanchong, Chen Zhaoying, Zheng Fanqing, and Peng Peng. "A Temporal XML Index: Txmlsindex." Journal of South China Normal University (Natural Science Edition) 1 (2015): 020

[14] Wu, Yuqing, Jignesh M. Patel, and H. V. Jagadish. "Structural join order selection for XML query optimization." In Data Engineering, 2003. Proceedings. 19th International Conference on, pp. 443-454. IEEE, 2003

[15] Bruno, Nicolas, Nick Koudas, and Divesh Srivastava. "Holistic twig joins: optimal XML pattern matching." In Proceedings of the 2002 ACM SIGMOD international conference on Management of data, pp. 310-321. ACM, 2002

[16] Jiang, Haifeng, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. "Holistic twig joins on indexed XML documents." In Proceedings of the 29th international conference on Very large data bases-Volume 29, pp. 273-284. VLDB Endowment, 2003

[17] Kaushik, Raghav, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. "On the integration of structure indexes and inverted lists." In Proceedings of the 2004 ACM SIGMOD international conference on Management of data, pp. 779-790. ACM, 2004

[18] Yang, Beverly, Marcus Fontoura, Eugene Shekita, Sridhar Rajagopalan, and Kevin Beyer. "Virtual cursors for XML

joins." In Proceedings of the thirteenth ACM international conference on Information and knowledge management, pp. 523-532. ACM, 2004

[19] Pamila, JC Miraclin Joyce, and Divya Rajagopal. "Intra and Inter XML Query Answering Using Holistic Boolean Twig Pattern Matching." Asian Journal of Information Technology 15, no. 4 (2016): 756-764

[20] Shnaiderman, Lila, and Oded Shmueli. "Multi-Core Processing of XML Twig Patterns." IEEE Transactions on Knowledge and Data Engineering 27, no. 4 (2015): 1057-1070

[21] Kung, Yi-Wei, Hsu-Kuang Chang, and Chung-Nan Lee. "A novel twig-join swift using SST-based representation for efficient retrieval of internet XML." Journal of Web Engineering 14, no. 3-4 (2015): 234-250

[22] Lu, Jiaheng, Tok Wang Ling, Chee-Yong Chan, and Ting Chen. "From region encoding to extended dewey: On efficient processing of XML twig pattern matching." In Proceedings of the 31st international conference on Very large data bases, pp. 193-204. VLDB Endowment, 2005

[23] Zhi-xian, Tang, Feng Jun, Xu Li-ming, and Shi Ya-qing. "A Bottom-up Algorithm for XML Twig Queries." International Journal of Database Theory and Application 8, no. 4 (2015): 49-58.

[24] Kung, Yi-Wei, Hsu-Kuang Chang, and Chung-Nan Lee. "A refined twig-join swift query algorithm for diversification issues of XML." Journal of Information Science (2015): 0165551515601004

[25] Anderson, Dewey C., and David J. Anderson. "System and method for retrieving information from a database using an index of XML tags and metafiles." U.S. Patent 6,510,434, issued January 21, 2003

[26] Papadimos, Vassilis, and David Maier. "Distributed queries without distributed state." In WebDB, pp. 95-100. 2002

[27] Braumandl, Reinhard, Markus Keidl, Alfons Kemper, Donald Kossmann, Stefan Seltzsam, and Konrad Stocker. "Objectglobe: Open distributed query processing services on the internet." IEEE Data Eng. Bull. 24, no. 1 (2001): 64-70

[28] Jim, Trevor, and Dan Suciu. "Dynamically distributed query evaluation." In Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 28-39. ACM, 2001

[29] Smiljanic, Marko, Henk Blanken, Maurice Keulen, and Willem Jonker. "Distributed XML database systems." (2002): 1-43

[30] Wu, Yuqing, Jignesh M. Patel, and H. V. Jagadish. "Structural join order selection for XML query optimization." In Data Engineering, 2003. Proceedings. 19th International Conference on, pp. 443-454. IEEE, 2003

[31] Camillo, Sandro Daniel, Carlos Alberto Heuser, and Ronaldo dos Santos Mello. "Querying heterogeneous XML sources through a conceptual schema." In International Conference on Conceptual Modeling, pp. 186-199. Springer Berlin Heidelberg, 2003