A Path-Based Labeling Scheme for Efficient Structural Join

Hanyu Li, Mong Li Lee, and Wynne Hsu

School of Computing, National University of Singapore
3 Science Drive 2, Singapore 117543
{lihanyu,leeml,whsu}@comp.nus.edu.sg

Abstract. The structural join has become a core operation in XML query processing. This work examines how path information in XML can be utilized to speed up the structural join operation. We introduce a novel approach to pre-filter path expressions and identify a minimal set of candidate elements for the structural join. The proposed solution comprises of a path-based node labeling scheme and a path join algorithm. The former associates every node in an XML document with its path type, while the latter greatly reduces the cost of subsequent element node join by filtering out elements with irrelevant path types. Comparative experiments with the state-of-the-art holistic join algorithm clearly demonstrate that the proposed approach is efficient and scalable for queries ranging from simple paths to complex branch queries.

1 Introduction

Standard XML query languages such as XQuery and XPath support queries that specify element structure patterns and value predicates imposed on these elements. For example, the query "//dept[/name='CS']//professor" retrieves all the professors from the CS department. It comprises of a value predicate "name='CS' " and two structural relations "dept//professor" and "dept/name" where '/' and "//" denote the child and descendant relationships respectively.

Traditional relational database access methods such as the B^+-tree can be easily extended to process value predicates in XML queries. Hence, the support of tree structured relationships becomes the key to efficient XML query processing.

Various node labeling schemes have been developed to allow the containment relationship between any two XML elements to be determined quickly by examining their labels or *node ids*. [4] identifies the structural join as a core operation for XML query pattern matching and develops a structural join algorithm called *Stack-Tree* which utilizes the interval-based node labeling scheme to evaluate the containment relationship of XML elements. Index-based methods such as B^+-tree [7], XR-tree [11], and XB-tree [5] speed up the structural join operation by reducing the number of elements involved in the node join.

In this work, we design a novel path-based approach to further expedite the structural join operation. The idea is to associate path information with the element nodes in an XML document so that we can filter out nodes that clearly do not match the query, and identify a minimal set of nodes for the regular node join. The proposed approach has the following unique features and contributions:

1. *Path Labeling Scheme.* We design a path-based labeling scheme that assigns a *path id* to every element to indicate the type of path on which a node occurs. The scheme is compact, and the path ids have a much smaller size requirement compared to the node ids (see Section 5 on space requirement).
2. *Containment of Path Ids.* The well-known node containment concept allows the structural relationship between any two nodes in an XML document to be determined by their node labels. Here, we introduce the notion of *path id containment* and show how the path labeling scheme makes it easy to distinguish between parent-child and ancestor-descendant relationships.
3. *Path Join.* We design a path join algorithm as a preprocessing step before regular node join to filter out irrelevant paths. The path join algorithm associates a set of relevant path ids to every node in the query pattern, thus identifying the candidate elements for the subsequent node join. Experimental results indicate that the relatively inexpensive path join can greatly reduce the number of elements involved in the node join.

The rest of this paper is organized as follows. Section 2 briefly reviews the related work. Section 3 presents the path-based labeling scheme. Section 4 describes the query evaluation. Finally, Section 5 presents the experimental results, and we conclude in Section 6.

2 Related Work

There has been a long stream of research in XML query evaluation. Early works develop various path index solutions such as DataGuides [10] and 1-Index [14] to summarize the raw paths starting from the root node in an XML document. These index structures do not support branch queries and XML queries involving wildcards and ancestor-descendant relationships efficiently. Index Fabric [9] utilizes the index structure Patricia Trie to organize all the raw paths, and provides support for the “refined paths” which frequently occur in the query workload.

The work in BLAS [6] also utilizes path information (p-labeling) to prefilter out unnecessary elements. BLAS employs integer intervals to represent all the possible paths, regardless of whether or not they occur in the XML dataset. Hence, BLAS will perform best for suffix queries, i.e., queries that start with optional descendant axis followed by child axes. In contrast, the proposed path encoding scheme utilizes bit sequences to denote the paths that actually occur in the XML datasets. Therefore, the proposed solution will yield optimal performance for simple queries, which are a superset of suffix queries.

The structural join is a core operation in many XML query optimization methods [4, 5, 7, 11–13]. [13] uses a sort-merge or a nested-loop approach to process the structural join. This method may scan the same element sets multiple times if the XML elements are recursive. *Stack-Tree* [4] solves this problem by applying an internal stack to store the subset of elements that is likely to be used later. Index-based binary structural join solutions such as B^+-tree [7], one dimensional R-tree [7], XB-tree [5] and XR-tree [11] employ different ways to “skip” elements involved in the query pattern without missing any matches. Holistic twig join methods such as XB-tree based TwigStack [5] and XR-tree based TSGeneric [12] are designed to process XML queries involving more than two nodes.

Sequence based approaches such as ViST [16] and PRIX [15] apply different ways to transform both the XML documents and queries into sequences. Query evaluation is carried out using sub-sequence matching. However, ViST may produce false alarms in the results, and PRIX requires a substantial amount of post-processing to guarantee the accuracy of the query results.

3 Path-Based Labeling Scheme

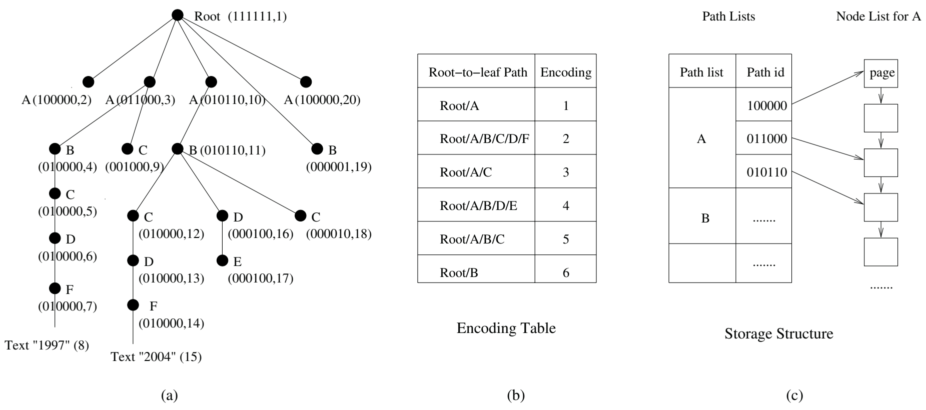
We design a path-based labeling scheme that assigns a *path id* to every element node in an XML document to indicate the type of path on which the node occurs. Each element node is now identified by a pair of (path id, node id). The node id can be assigned using any existing node labeling scheme, e.g., interval-based [13], prefix [8], prime number [17]. Text nodes are labeled with node ids only.

A path id is composed of a sequence of bits. We first omit the text nodes from an XML document. Then we find distinct root-to-leaf paths in the XML document by considering only the tag names of the elements on the paths. We use an integer to encode each distinct root-to-leaf path in an XML document. The number of bits in the path id is given by the number of the distinct root-to-leaf element sequences of the tag names that occur in the XML document. Path ids are assigned to element nodes in a bottom-up manner as follows.

1. After omitting the text nodes in an XML document, let the number of distinct root-to-leaf paths in the XML document be k. Then the path id of an element node has k bits. These bits are initially set to 0.
2. The path id of every leaf element node is given by setting the *ith* bit (from the left) to 1, where i denotes the encoding of the root-to-leaf path on which the leaf node occurs.
3. The path id of every non-leaf element node is given by a bit-or operation on the path ids of all its element child nodes.

Consider the XML instance in Figure 1(a) where the node ids have been assigned using a pre-order traversal. Figure 1(b) shows the integer encodings of each root-to-leaf paths in the XML instance. Since there are 6 unique root-to-leaf paths, 6 bits are used for the path ids.

The path id of the element leaf node F (node id = 7) is 010000 since the encoding of the path $Root/A/B/C/D/F$ on which F occurs is 2. The path id of the non-leaf node A (node id = 3) is obtained by a bit-or operation on the path ids of its child nodes B and C, whose path ids are 010000 and 001000 respectively. Therefore, the path id of A is 011000. Note that each text node is only labeled by a node id.



Root–to–leaf Path	Encoding
Root/A	1
Root/A/B/C/D/F	2
Root/A/C	3
Root/A/B/D/E	4
Root/A/B/C	5
Root/B	6

Fig. 1. Path-Based Labeling Scheme and Its Storage Structure

3.1 Storage Structure

In order to facilitate the direct retrieval of elements with a specified path id, we design the following storage structure:

1. All the path ids of one element tag comprise the path id list of this element.
2. All the node ids of one element tag comprise the node id list of this element. The node list is first clustered by element path ids, and then sorted on the node ids.
3. Each path id in the path id list points to the first element with this path id.

Figure 1(c) shows how the example XML document in Figure 1(a) is stored. Tag A has four occurrences with three distinct path ids. There are two occurrences corresponding to the first path id (100000) and one each corresponding to the other two (011000 and 010110).

3.2 Containment of Path IDs

In this section, we introduce the notion of path id containment that is based on the proposed path labeling scheme and examine the relationship of path id containment with node containment.

Definition (Path ID Containment): Let Pid_A and Pid_B be the path ids of nodes A and B respectively. If all the bits with value 1 in Pid_A cover all the bits with value 1 at corresponding positions in Pid_B, then we say Pid_A contains Pid_B.

The containment relationship between the path ids can simply be determined with a bit-and operation. That is, if $(Pid_A \mathbin{\&} Pid_B) = Pid_B$ where & denotes the "bit-and" operation, then Pid_A contains Pid_B. For example, in Figure 1, the path id of A(010110,10) contains the path id of C(010000,12).

Definition (Strict Path ID Containment): Let Pid_A and Pid_B be the path ids of nodes A and B respectively. If Pid_A contains Pid_B and $Pid_A \neq Pid_B$, then we say Pid_A strictly contains Pid_B.

In Figure 1, the path id of A (010110,10) strictly contains the path id of C (010000,12).

The node containment between nodes can be deduced from path id containment relationship. Theorem 1 introduces this.

Theorem 1: Let Pid_A and Pid_D be the path ids for elements with tags A and D respectively. If Pid_A strictly contains Pid_D, then each A with Pid_A must have at least one descendant D with Pid_D.

Proof: Since Pid_A strictly contains Pid_D, then all the bits with value 1 in Pid_D must occur in Pid_A at the same positions. Further, Pid_A will have at least one bit with value 1 such that the corresponding bit (at the same position) in Pid_D is 0. Consequently, elements with tag A will occur in the same root-to-leaf paths as the elements with tag D, and there will exist at least one root-to-leaf path such that elements with tag A occur, and elements with tag D do not occur. As a result, all the elements with tag A must have elements with tag D as descendants. $\Box$

Consider again Figure 1. The path id 010110 for node B strictly contains the path id 010000 for node C. Therefore, each node B (node id=11) with path id 010110 must be the ancestor of at least one node C (node id=12) with path id 010000.

In the case where there are two sets of nodes with the same path ids, we need to check their corresponding root-to-leaf paths to determine their structural relationship. For example, the nodes A and B (node ids are 10 and 11) in Figure 1 have the same path id 010110. We can decompose the path id 010110 into 3 root-to-leaf paths with the encodings 2, 4 and 5 since the bits in the corresponding positions are 1. Thus, by looking up any of these paths (in the encoding table) where nodes A and B occur, we know that all the nodes A with path id 010110 have B descendants with this path id.

Similarly, the encoding table can help us to determine the exact containment relationship between any two sets of nodes, that is, parent-child, grandparent-grandchild, etc. For example, given the path id 010110 for nodes B and the path id 010000 for nodes C, we know that all the B nodes have C descendants. Further, from the root-to-leaf path with value 2, we also know that the B nodes are parents of the corresponding C nodes.

In other words, if Pid_A and Pid_D are the path ids of two sets of nodes A and D respectively, then we can determine the exact relationship (parent-child, grandparent-grandchild..) between these two sets of nodes from the encoding table, provided that a tag name occurs no more than once in any path. For example, suppose nodes A and D have the same path ids, and their corresponding root-to-leaf path is "A/D/A/D". In this case, the structural relationship between A and D can only be determined by examining their node labels (node ids).

4 Evaluation of Structural Join

The structural join operation evaluates the containment relationship between nodes in given XML queries. Our path-based approach processes structural join in two steps: (1) path join, and (2) node join. The algorithms for carrying out these two steps are called $PJoin$ and $NJoin$ respectively.

4.1 *PJoin*

The $PJoin$ algorithm (Algorithm 1) aims to eliminate as many unnecessary path types as possible, thus minimizing the elements involved in the subsequent $NJoin$.

Given an XML query modelled using a tree structure T, $PJoin$ will retrieve the set of path ids for every element node in T. Starting from element leaf nodes in T, $PJoin$ will perform a binary path join between each pair of parent-child nodes. This process is carried out in a bottom-up manner until the root node is reached. After that, a top-down binary path join is performed to further remove unnecessary path ids.

A binary path join takes as input two lists of path ids, one for the parent node and the other for the child node. A nested loop is used to find the matching pairs of path ids based on the path id containment property. Any path id that does not satisfy the path id containment relationship is removed from the lists of path ids of both parent and child nodes.

Algorithm 1 $PJoin$ (T)

Input: T - An XML Query.
Output: Path ids for the nodes in T.

```
1. Associate every node in $T$ with its path ids.
2. Perform a bottom-up binary path join on $T$.
3. Perform a top-down binary path join on $T$.
```

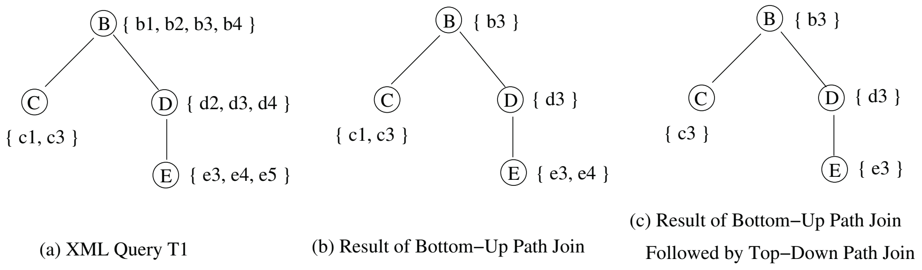


Fig. 2. Example of $PJoin$

Consider the XML query $T1$ in Figure 2(a) where the lists of path ids have been associated with the corresponding nodes. We assume that the path ids with the same subscripts satisfy the path id containment relationship, that is, b_1 contains c_1, b_3 contains d_3 and e_3, etc.

The $PJoin$ algorithm evaluates the query $T1$ by first joining the path ids of node B with that of node C. The path id c_1 and c_3 are contained in the path id b_1 and b_3 respectively. Thus, we remove b_2 and b_4 from the set of path ids of B.

Next, the algorithm joins the set of path ids of D with that of E. This is followed by a join between the sets of path ids of B and D. The result of the bottom-up path join is shown in Figure 2(b).

Finally, the algorithm carries out a top-down path join on $T1$ starting from the root node B. Figure 2(c) shows the final sets of path ids that are associated with each node in $T1$. Compared to the initial set of path ids associated with each node in Figure 2(a), the *PJoin* algorithm has greatly reduced the number of elements involved in the query. The subsequent node join is now almost optimal.

Note that omitting either the bottom-up or top-down tree traversal will not be able to achieve this optimal result. This is because a single tree traversal cannot project the result of each binary path join to the nodes which have been processed earlier. In Figure 2(b), elements C and E contains unnecessary path ids $c1$ and $e4$ compared to the final optimal results.

4.2 *NJoin*

The output of *PJoin* algorithm is a set of path ids for the element nodes in a given query tree. Elements with these path ids are retrieved for a node join to obtain the result of the query. Algorithm 2 shows the details of *NJoin*.

We modify the holistic structural join developed in [5] to perform the node join. The element nodes are retrieved according to the path ids obtained from the *PJoin*, while all the value (text) nodes are retrieved directly. Finally, a holistic structural join is carried out on all the lists obtained.

Algorithm 2 *NJoin*(T)

```
Input: T - An XML Query.
Output: All occurrences of nodes in T.

1. Retrieve the elements according to the path ids associated with nodes in T
2. Retrieve the values imposed on the element nodes.
3. Perform holistic structural join on T.
```

We observe that the structural join in Line 3 of Algorithm 2 requires that the input stream for every node in the query must be an ordered list of node ids. However, Line 1 of Algorithm 2 produces a set of ordered sublists, each of which is associated with a path id obtained in the *PJoin*. Therefore, when performing the structural join, we will need to examine these multiple sublists of node ids for an element tag to find the smallest node id to be processed next.

4.3 Discussion

The path join is designed to reduce the number of elements involved in the subsequent node join. In this section, we analyze the effectiveness of the proposed path join.

Definition (Exact Pid Set): Let P be a set of path ids obtained for a node n in an XML query T. P is an *exact Pid set* with respect to T and n if the following conditions hold:

1. for each path id $p_i \in P$, the element with tag n and path id p_i is a result for T, and
2. for each path id $p_j \notin P$, the element with tag n and path id p_j is not a result for T.

Definition (Super Pid Set): Let P be a set of path ids obtained for a node n in an XML query T. P is a *super Pid set* with respect to T and n if each element with a tag n in the final result (after node join) is associated with a path id p_i such that $p_i \in P$.

Clearly, each element node is associated with its super *Pid* set after the path join. The result is optimal when these super *Pid* sets are also the exact *Pid* sets.

Next, we examine the situations where path join will yield exact *Pid* sets. We assume that the XML elements are non-recursive.

Simple Path Queries. Suppose query T is a simple path query without value predicates. Then each node in T will have an exact *Pid* set after the path join. This is because all the path ids that satisfy the path id containment property are reserved in the adjacent nodes of T. Since this containment property is transitive, all the path ids of a node n in T will contain the path ids of its descendant nodes, and vice versa. Moreover, the encoding table for the paths can identify the exact containment relationship (parent-child or ancestor-descendant) between the nodes in T. Therefore, given a simple path XML query without value predicates, the path join will eliminate all the elements (path ids) that do not appear in the final result sets.

Branch Queries. If a query T is a branch query, then we cannot guarantee that the nodes on the branch path have exact *Pid* sets because of the manner in which the path ids are assigned to the elements. In other words, the path id is designed to capture the containment relationship, but not the relationship between sibling nodes.

Consider the query in Figure 3(a) which is issued on the XML instance in Figure 1. After the path join, node F will be associated with a path id 010000. However, we see that only F(010000,14) is an answer to this query while F(010000,7) is not. This is because we can only detect 010000 (path id of F) is contained by 010110 (path id of B), but do not know whether an F element with path id 010000 will have sibling E. Finally, note that the path id set of B in Figure 3(a) is guaranteed to be an exact *Pid* set since B has no sibling nodes.

Queries with Value Predicates. Figure 3(b) shows an XML query with value predicates that will lead to super *Pid* sets after a path join. The node D (010000,13) and F (010000,14) in Figure 1 are not answers to the query although the path id 010000 occurs in the path id sets of D and F respectively after the path join. This is because we do not assign path information to value

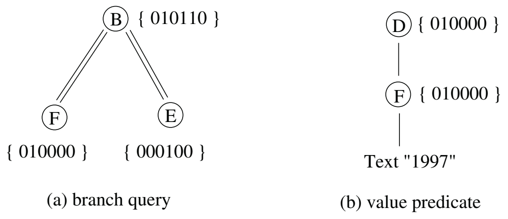


Fig. 3. Examples of Super Pid Set

nodes. Therefore, the element nodes with the matching path id can only satisfy the structural relationship, and not the value constraints. As a result, if an XML query has value predicates, the path id set of each node in the query pattern may not be the exact Pid set.

To summarize, for non-recursive XML data, the exact Pid sets will be associated with the element nodes in simple XML query patterns after the path join, while only the super Pid sets (which are much less than the full path id sets of elements as shown in our experimental section) can be guaranteed for the nodes in branch queries and queries with value predicates.

5 Experiments

This section presents the results of experiments to evaluate the performance of proposed path-based approach. We compare the path-based approach with the state-of-the-art XB-tree based TwigStack [5]. Both solutions are implemented in C++. All experiments are carried out on a Pentium IV 2.4 GHz CPU with 1 GB RAM. The operating system is Linux 2.4. The page size is set to be 4 KB.

Table 1 shows the characteristics of the experimental datasets which include Shakespeare's Plays (SSPlays) [1], DBLP [2] and XMark benchmark [3]. Attributes are omitted for simplicity.

Table 1. Characteristics of Datasets

Datasets	Size	♯(Distinct Elements)	♯(Elements)
SSPlays	7.5 MB	21	179,690
DBLP	60.7 MB	32	1,534,453
XMark	61.4 MB	74	959,495

5.1 Storage Requirements

We first compare the space requirement of the path-based approach with the XB-tree [5]. Our implementation of the XB-tree bulkloads the data and keeps every node half full except for the root node. The page occupancy for the node lists in the path-based approach is also kept at 50%. To be consistent with the XB-tree, the path-based solution also utilizes the interval-based node labeling scheme to assign the node ids. The storage requirements are shown in Table 2.

It can be observed that the sizes of encoding tables are very small (0.24K, 0.38K and 2.9K respectively), and hence we load them into memory in our experiments. The space required by the path lists is determined by the degree of regularity of the structures of the XML documents (see Table 3). The real-world datasets typically have a regular structure, and thus have fewer distinct paths (40 distinct paths in SSPlays and 69 in DBLP) compared to the 344 distinct paths in the synthetic XMark dataset. Since the number of bits in the path id is given by the number of distinct paths, the path ids for the SSPlays and DBLP are only 5 and 9 bytes respectively. In contrast, the irregular structure in XMark needs 43 bytes for the path id.

Table 2. Space Requirements

Datasets	XB	Path		
		Encoding Table	Path Lists	Node Lists
SSPlays	8.0MB	0.24KB	5.9KB	6.5MB
DBLP	69.6MB	0.38KB	9.1 KB	57.2MB
XMark	40.4MB	2.90KB	884.2KB	32.3MB

Table 3. Storage for Path Ids

Datasets	♯(Distinct Path)	Path Id Size(Bytes)
SSPlays	40	5
DBLP	69	9
XMark	344	43

We also observe that the size of the path lists is relatively small compared with that of node lists. Even for the most irregular structure dataset XMark, the size of path lists takes only 2.7% of its node lists size (884K and 32M respectively). This feature fundamentally guarantees the low cost of path join.

5.2 Query Performance

Next, we investigate the query performance of the path-based approach and compare it with the XB-tree based holistic join [5]. Table 4 shows the query workload for the various datasets. The query workload comprises short simple queries, long path queries and branch queries (Q1-Q8). To examine the effect of parent-child relationship, we replace some ancestor-descendant edges in queries Q1, Q3 and Q8 with parent-child relationship. Moreover, value constraints are imposed on queries Q1, Q2, Q5 and Q6 respectively to test the influence of value predicates.

Effectiveness of Path Join. In this set of experiments, we demonstrate the effectiveness of the path join algorithm in filtering out elements that are not relevant for the subsequent node join.

A metric called "Filtering Efficiency" is first defined to measure the filtering ability of path join. This metric gives the ratio of the number of nodes after a

Table 4. Query Workload

	Query	Dataset	♯ Nodes in Result
Q1	//PLAY//TITLE	SSPlays	1068
Q2	//PLAY/ACT/SCENE/SPEECH/LINE/STAGEDIR	SSPlays	2259
Q3	//SCENE//STAGEDIR	SSPlays	6974
Q4	//proceedings/booktitle	DBLP	3314
Q5	//proceedings[/url]/year	DBLP	5526
Q6	//people/person/profile[/age]/education	XMark	7933
Q7	//closed_auction/annotation[//emph]//keyword	XMark	13759
Q8	//regions/australia/item//keyword[//bold]//emph	XMark	74
Q1pc	//PLAY/TITLE	SSPlays	74
Q3pc	//SCENE/STAGEDIR	SSPlays	5010
Q8pc	//regions/australia/item//keyword[/bold]/emph	XMark	74
Q1v	//PLAY//TITLE="ACT II"	SSPlays	111
Q2v	//PLAY/ACT/SCENE/SPEECH/LINE/STAGEDIR="Aside"	SSPlays	1044
Q5v	//proceedings[/url]//year="1995"	DBLP	432
Q6v	//people/person[/age="18"]/profile/education	XMark	2336

path join over the total number of nodes involved in the query. That is, given a query Q, we have

$$Filtering\ \ Efficiency = \frac{\sum |N_i^p|}{\sum |N_i|}$$

where $|N_i^p|$ denotes the number of instances for node N_i after a path join and $|N_i|$ denotes the total number of instances for N_i in the query.

We also define "Query Selectivity" to reflect the percentage of nodes in the result set compared to the original number of nodes involved in the query. Given a query Q, we have

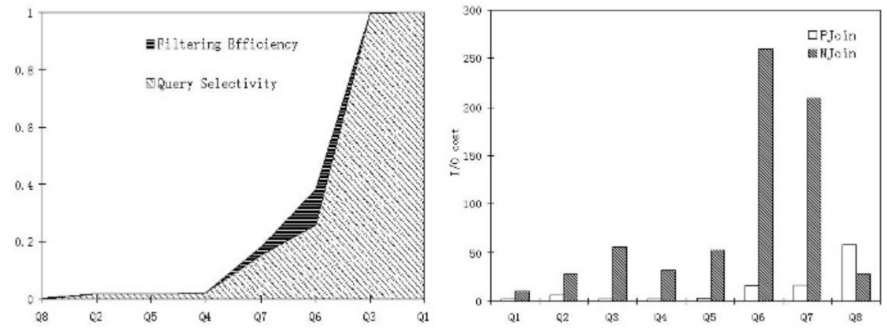
$$Query\ \ Selectivity = \frac{\sum |N_i^n|}{\sum |N_i|}$$

where $|N_i^n|$ denotes the number of instances for node N_i in the result set after a node join and $|N_i|$ is the same as above.

The effectiveness of path join can be measured by comparing the values of its Filtering Efficiency and Query Selectivity. Based on the definitions of these two metrics, we can see the closer the two values are, the more effective the path join is for the query. The optimal case is achieved when the Filtering Efficiency is equivalent to the Query Selectivity, indicating that path join has effectively filtered out all irrelevant elements for the subsequent node join.

Figures 4(a) compares the Filtering Efficiency with Query Selectivity for queries Q1 to Q8 whose Query Selectivity values are in ascending order. Except for queries Q6, Q7 and Q8, the rest queries have the same values for two metrics. This shows that path join has effectively removed all irrelevant elements for the node join.

Queries Q6, Q7 and Q8 have higher Filtering Efficiency values compared to their Query Selectivity. This indicates that the path join algorithm does not produce exact *Pid* sets for the subsequent node join for these queries. As we



(a) Filtering Efficiency Vs Query Selectivity (b) PJoin and NJoin (I/O cost)

Fig. 4. Effectiveness of Path Join

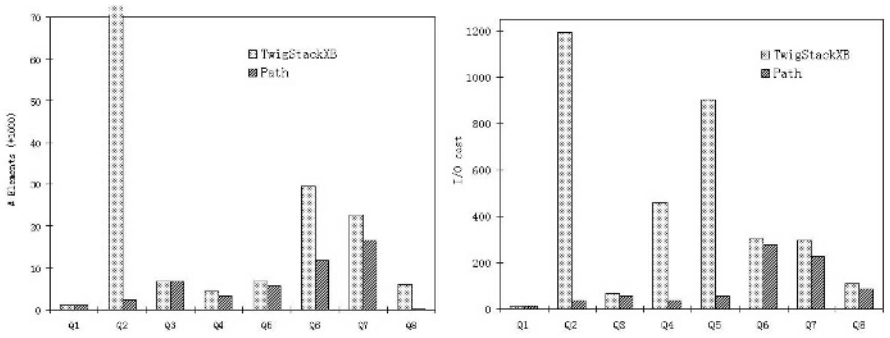
have analyzed in Section 4.3, the *Pid* sets associated with nodes after the path join may not be the exact *Pid* sets for branch queries. Since Q6, Q7 and Q8 are all branch queries, this result is expected. Note that path join still remains efficient in eliminating unnecessary path types even for branch queries, which can be seen from the close values of filtering efficiency and query selectivity of queries Q5, Q6, Q7 and Q8 (all are branch queries, and the two values are same for Q5).

Figure 4(b) compares the I/O cost of path join and node join. The graph shows that the cost of path join is very marginal for the majority of the queries compared to that of node join. This is because the size of path lists involved in the query is much smaller than that of node lists (recall Table 2).

The costs of path join for queries Q1 to Q5 are negligible because of the regular structures of SSPlays and DBLP. The path join is more expensive for the queries over XMark dataset (Q6 to Q8) due to its irregular structure, which results in a larger number of path types and longer path ids. Among these queries on synthetic dataset (Q6 to Q8), query Q8 is the only one where the cost of path join is greater than the node join. This can be explained by the low selectivity of Q8 (74 nodes in result, Table 4), which directly contributes to the low cost of node join. Finally, the result in Figure 4(a) clearly demonstrates that the path join remains effective in filtering out a large number of elements for queries even with the influence of irregularity in synthetic dataset.

Efficiency of Approach. In this set of experiments, we compare the performance of path-based approach with XB-tree based holistic join [5]. The metrics used are the total number of elements accessed and I/O cost. Figure 5 shows that the path-based approach performs significantly better than the XB-tree based holistic join. This is because path join is able to greatly reduce the actual number of elements retrieved.

We observe that the underlying data storage structure of path-based approach has an direct effect on the query performance. For queries Q4 and Q5, the I/O costs are smaller than the number of elements accessed in path-based

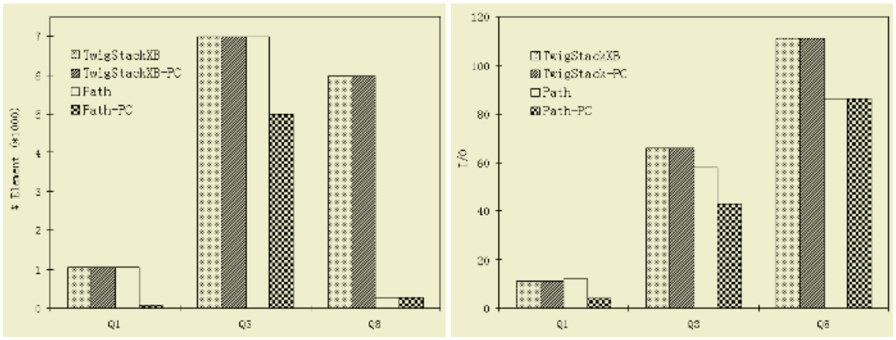


(a) Elements Accessed (b) I/O Cost

Fig. 5. Queries with Structural Patterns Only

approach (see Figure5(a) and (b)). This is because the path-based approach clusters the node records based on their paths. This further reduces the I/O cost during data retrieval. In contrast, the I/O cost for XB-tree is determined by the storage distribution of matching data. In the worst case, the elements to be accessed are scattered over the entire list, leading to high I/O costs.

Effect of Parent-Child Relationships. We examine the effect of parent-child relationship on query performance by replacing some ancestor-descendant edges in queries Q1, Q3 and Q8 with parent-child edges. Figure 6 shows the results.



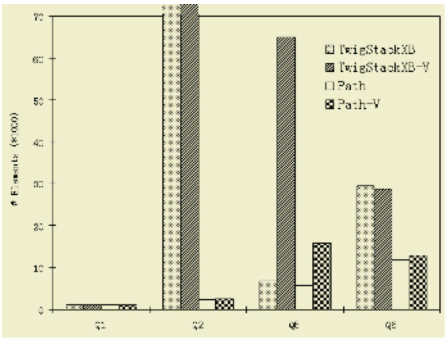
(a) Elements Accessed (b) I/O Cost

Fig. 6. Parent-Child Queries

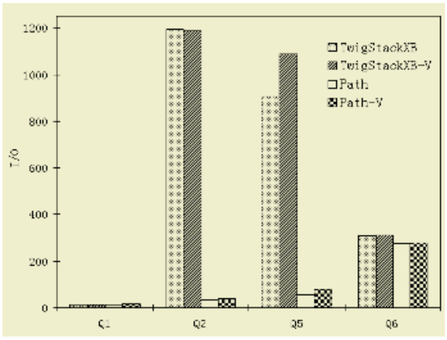
The XB-tree based holistic join utilizes the same method to evaluate the parent-child queries and ancestor-descendant queries. Therefore, XB-tree based holistic join has the same evaluation performance for parent-child and ancestor-descendant queries. To avoid incorrect result, each parent-child edge is (inexpensively) verified before it is output.

In contrast, the proposed path-based approach checks for parent-child edges during the path join. This task is achieved by looking up the encoding table (see Figure 1(b)). In the case where the results of parent-child queries are subsets of the ancestor-descendant counterparts, the cost to evaluate queries may be further reduced since fewer elements are involved in the node join. For example, queries Q1pc and Q3pc have smaller result sets compared to Q1 and Q3 (see Table 4) respectively. Thus Q1pc and Q3pc show better performance in Figure 6.

Effect of Value Predicates. Finally, we investigate how the proposed approach and XB-tree perform for queries involving value predicates. We add value constraints on queries Q1, Q2, Q5 and Q6 respectively. The results are shown in Figure 7.



(a) Elements Accessed



(b) I/O Cost

Fig. 7. Queries with Value Predicates

When evaluating XML queries involving value predicates, the path-based solution first carries out a path join to process the structural aspects of the queries. To determine the final set of results, the subsequent node join will retrieve the value nodes and element nodes obtained by path join. Therefore, the path-based solution needs to access more nodes to evaluate the value predicates in the queries compared to the corresponding queries without value predicates. This can be observed in Figure 7.

The XB-tree based holistic join solution treats value nodes the same way as element nodes. The additional value predicates will incur more costs during the retrieval of nodes. However, the value constraints may reduce the total number of element nodes accessed. This is because the XB-tree approach employs the XB-tree index to search for the matching nodes. Thus, it may skip some element nodes that match the structural query pattern but not the value predicates. Figure 7 shows that the addition of value predicates have different effects on performances of Q5 and Q6.

Overall, the evaluation of structural patterns still dominates the query performance even for queries involving value predicates. This is shown clearly in Figure 7.

6 Conclusion

In this paper, we have presented a new paradigm for processing structural join. The proposed solution includes a path-based labeling scheme and a path join algorithm that is able to compute the minimal sets of elements required for the subsequent node join. Experimental results clearly show that the proposed approach outperforms existing structural join methods for the following reasons:

1. The path join filters out nodes with path types that are not relevant to the subsequent node join;
2. The cost of path join is marginal compared to node join in the majority of the queries;
3. The element records are clustered according to the path types, which further reduces the I/O cost during element retrieval.

References

1. http://www.ibiblio.org/xml/examples/shakespeare.
2. http://www.informatik.uni-trier.de/˜ley/db/.
3. http://monetdb.cwi.nl/.
4. 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 ICDE, USA*, 2002.
5. N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of SIGMOD, USA*, 2002.
6. Y. Chen, S. B. Davidson, and Y. Zheng. BLAS: An Efficient XPath Processing System. In *Proceedings of SIGMOD, France*, 2004.
7. S-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of VLDB, China*, 2002.
8. E. Cohen, H. Kaplan, and T. Milo. Labelling Dynamic XML Tree. In *Proceedings of PODS, USA*, 2002.
9. B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *Proceedings of VLDB, Italy*, 2001.
10. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of VLDB, Greece*, 1997.
11. H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proceedings of ICDE, India*, 2003.
12. H. Jiang, W. Wang, and H. Lu. Holistic Twig Joins on Indexed XML Documents. In *Proceedings of VLDB, Germany*, 2003.
13. Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of VLDB, Italy*, 2001.
14. T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proceedings of ICDT, Israel*, 1999.
15. P. Rao and B. Moon. PRIX: Indexing and Querying XML Using Prüfer Sequences. In *Proceedings of ICDE, USA*, 2004.
16. 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 SIGMOD, USA*, 2003.
17. X. Wu, M. Lee, and W. Hsu. A Prime Number Labelling Scheme for Dynamic Ordered XML Trees. In *Proceedings of ICDE, USA*, 2004.