ABSTRACT
We introduce a new technique for fast computation of structural join “pattern trees” in XML. Using a small amount of pre-computed path information, typically small enough to fit easily in main memory, we are able to render structural join computation almost independent of data set size. Our technique is amenable to bit-mapped processing, leading to further speed-up. In this paper, we present our technique and experimentally evaluate its performance.

1. INTRODUCTION
Structural join computation is at the heart of XML query processing, and there is a significant recent body of work on this topic. A typical approach to evaluate an XPath expression is to first select a list of XML tree nodes that satisfy the node-associated constraints for each pattern tree node, and then pair wise join the lists based on their structural relationships (such as parent-child or ancestor-descendant). Through the use of suitable node labels, structural joins can be evaluated reasonably efficiently by various join techniques (merge join, stack-based structural join, holis-tic twig joins, and subsequence matching) Join algorithms. However, all these algorithms depend on the size of their inputs, usually linearly. It is often the case that these inputs are very large while the set of matching results (the output) is small. Our objective in this paper is to minimize this dependence on input size, and thereby obtain an algorithm likely to be very fast in most real situations.

1.1 Outline and Contributions
1. Join indices are well known. They are effective at reducing the cost of join computation but are notoriously expensive to store and even more expensive to maintain. As such, they are popular only for read only systems, such as data warehouses. We develop new index structures in Section 2.1 based on the conceptual ideas of join indices, path indices and extending them in crucial ways. Specifically, we propose a new Signatures-PathType index that has the benefits of a join index, but not the drawbacks. We develop a new style of computing chain structural joins that uses Signatures-PathType index to reduce the number of comparisons needed to produce the join and transform the join operation to an in-memory intersection operation between the contributing labels inside a chain query. Our algorithm, BPI, described in Section 3, can be used to evaluate a single structural join or a whole chain query as one structural join no matter how many labels occur in that chain query.
2. Since our central computation is an intersection, it is very efficient to evaluate over a bit vector. Therefore, we encode a core portion of our indices as a bitmap. We store the bitmap vectors inside a B-tree index. These optimizations are presented in Section 4.
3. Index construction and maintenance are described in Section 5.
4. We present experimental results in Section 6 on a range of data and queries. We show that the performance of the BPI algorithms is much better than that of LCS-TRM, a current algorithm considered by many to be the best structural join computation technique today. We briefly present related research in XML structural joins in Section 7, and conclude in Section 8.

2. DEFINITIONS AND BASIC DATA STRUCTURES
In this section, we present index structures used in our proposed algorithm, BPI, for chain query evaluation and some core definitions.

2.1 Index Structures
Several path indices have been proposed in the literature for various purposes. These differ from one another in minor ways. To be precise about what we mean, and also for keeping this paper self-contained, we present our definitions in Sections 2.1.2 and 2.1.3 below, noting that these are not a novel contribution on our part. We then present our new Signatures-PathType index structure in Section 2.1.4 and index size analysis in Section 2.2. We begin with some basic definitions.

2.1.1 Core Definitions

DEFINITION 1 (XML Path). A path $P = \{n_1, ..., n_k\}$ in an XML document $d$, where each $n_{i+1}$ is a child node of $n_i$ for $i = 1, ..., k - 1$, is said to be an XML path of length $k$ if $n_1$ is the XML document root and $n_k$ is a leaf node.
Consider Figure 1, the left most path of the tree from the root element dblp until the element title including the element book is considered an XML Path.

**Definition 2** (Query Labels). Consider a query \( Q = \{ \text{tagName}_1, \ldots, \text{tagName}_k \} \) with \( k \) different tags \( \text{tagName}_i \), for \( 1 \leq i \leq k \), are called query labels, where each tag \( \text{tagName}_i \) is its height. This is likely to be much smaller than the count of nodes in the XML document and is its label or tag name, encoded as an integer for convenience. We store only distinct labels.

Consider a query \( Q = \text{dblp}/\text{title} \), the query labels inside this query are both \( \text{dblp} \) and \( \text{title} \).

**Definition 3** (Signature). The signature of a node is its label or tag name, encoded as an integer for convenience. We store only distinct labels.

Consider Figure 1, the left most path of the tree from the root element \( \text{dblp} \) until the element \( \text{title} \) including the element \( \text{book} \) is the signature of a node \( n_i \), along this path and \( k \) is the count of nodes in the path.

Consider a query \( Q = \text{dblp}/\text{title} \), the query labels inside this query are both \( \text{dblp} \) and \( \text{title} \).

Figure 2a shows the distinct labels available in the XML document in Figure 1 along with their signatures.

**Definition 4** (XML Path Type). The type of an XML path \( P = \{ n_1, \ldots, n_k \} \) is \( S = \{ s_1, \ldots, s_k \} \) where each \( s_i, 1 \leq i \leq k \), is the signature of a node \( n_i \), along this path and \( k \) is the count of nodes in the path.

Figure 2b shows the XML path types available in the XML document in Figure 1 along with their signatures.

### 2.1.2 Path of Signatures Index

In this index, we store the signatures that occur along each XML path type. The **Path of Signatures** index for the XML tree in Figure 1 is presented in Figure 2d.

### 2.1.3 Path of Nodes Index

Nodes are identified by node identifiers, obtained through a traversal of the XML document in depth first order. In this index, for each path type, we store the nodes that occur along each XML path of that type.

Consider, for example, the left most path of the tree in Figure 1. The node numbers of nodes labeled \( \text{dblp} \), \( \text{book} \) and \( \text{title} \) are \( n_1 \), \( n_2 \) and \( n_3 \) respectively. The path of nodes index for the XML tree in Figure 1 is presented in Figure 2c.

### 2.1.4 Signatures-PathType Index

Consider all distinct signatures \( \{ s_i \} \) available in the XML document where \( 1 \leq i \leq |\text{signatures}| \) and XML path type identifiers \( \{ t_j \} \) where \( 1 \leq j \leq |\text{PathTypes}| \). The **Signatures-PathType** index is a matrix where the rows represent the signatures \( \{ s_i \} \), columns represent the path type identifiers \( \{ t_j \} \) and the existence of each signature inside each path type is stored in element \( (i,j) \) of the matrix.

A row in this index represents all path types having the same signature. The **Signatures-PathType** index is presented in Figure 2b.

### 2.2 Index Size Analysis

The space requirement for **Path of Nodes** index is of \( O(\ell \cdot h) \) where \( \ell \) is the count of nodes in the XML document and \( h \) is its height.

The space requirement for the **Signatures-PathType** index is \( O(\ell \cdot L) \) where \( L \) is the count of distinct labels in the XML document and \( \ell \) is the count of XML path types. Usually \( L \ll \ell \) and \( t \ll \ell \), leading to an index whose size does not necessarily go up with \( \ell \), the count of nodes in the XML document. In contrast, a true join index could have size \( O(\ell^2) \). The space requirement for **Path of Signatures** index is \( O(\ell \cdot h) \) where \( \ell \) is the count of XML path types and \( h \) is its height. This is likely to be much smaller than the count of XML paths or nodes in the XML document.

### 3. CHAIN QUERY JOIN EVALUATION ALGORITHMS
To efficiently answer path queries, we propose the BPI algorithm whose modules are shown in Figure 3. The input to the algorithm is a chain query. Query label signatures are passed to the Intersect Path Types algorithm that uses the Signatures-PathType index to find path types containing all query signatures. The Check Relationships algorithm is then applied to each result path type to determine if the query label conforms to the same query structure. The output of the Check Relationships algorithm is the result of the structural join operation between the query labels. The following subsections describe in detail each of the above mentioned algorithms.

3.1 Intersect Path Types Algorithm

The primary function of this algorithm is to match query labels with path type labels based on signatures. The input to this algorithm are query labels, translated to signatures by the trivial Translate to Signatures algorithm. Consider a query Q = \{s1, ..., sq\} where si are query signatures, 1 ≤ i ≤ q and q is the number of query signatures. Using Signatures-PathType index, for each signature si in query labels, we retrieve path types that contain this signature: \{t1, ..., tk\} where ti is the set of path types containing the signature si, 1 ≤ i ≤ q. The output of the Intersect Path Types algorithm is \( \cap \{t1, ..., tk\} \).

**Example 1.** Using Figure 1 data set, consider the path expression: article//name/Fname. The signatures of the three query labels are s11, s5 and s8 respectively as shown in Figure 2a. Using the Signatures-PathType index shown in Figure 2b, we retrieve the path types associated with each signature. The path types for signature s11 are t8 to t12, for signature s5 are t2 and t9, and for s8, the path types are t5 and t9. The intersection between these three lists is path type t9, the only path type that contains all three signatures.

**Definition 5 (Intersection Path Types).** The Intersection Path Types are the XML Path Types that contain all query labels. We call the output from the Intersect Path Types algorithm, the set of Intersection Path Types.

From the previous example, the set of Intersection Path types contains only path type t9.

3.2 Check Relationships Algorithm

While every query label appears in each intersection path type, not all of these may satisfy the given query since the query labels may appear in the wrong order. The Check Relationships algorithm performs structure matching between the query and the path types.

3.2.1 Check Relationship for Non-Recursive Schemas

Consider input intersection path types \( P = \{t1, ..., tm\} \) where ti are path types that contain all query labels and 1 ≤ i ≤ m and m is the number of intersection path types. Using Path of Signatures index, one can retrieve path signatures for each input path type. For a path type in the input, let \( S = \{s1, ..., sh\} \) be the signatures and h be the depth of this path type. Consider query signatures \( Q = \{q1, ..., qk\} \) where k is the total number of labels in the query. Consider also relationships \( R = \{R1, ..., Rk-1\} \) between Q signatures, where each Ri relates qi with qj+1, 1 ≤ i < k. A path type is considered one of the result path types if and only if all the following conditions hold:

- For each qi in \( \{q1, ..., qk\} \), qi occurs exactly once in \( \{s1, ..., sh\} \).
- For each Ri in R:
  - If Ri is a parent-child relationship between qi, qi+1, and qi = sj, we must have qi+1 = sj+1 for some \( j ≥ 1 \).
  - If Ri is an ancestor-descendant relationship between qi, qi+1, and qi = sj, we must have \( qi+1 = sj+2 \) for some \( j ≥ 1 \).

We use the Path of Nodes index to retrieve the paths of nodes for each successful result path type. The positions of the nodes of interest in the path of signatures index are used to read off directly the result nodes from the path of nodes index. The pseudo code of this algorithm is presented in Algorithm 1.

**Example 2.** Continuing Example 1, the intersection path type is t9. We check the relationships of this path type to be sure that it satisfies the relationships between the three nodes article, name and Fname. Path of Signatures for t9 is s11, s11, s5 and s8 as shown in Figure 2d. We traverse the path of signature scanning forward to find signatures s11, s5 and s8 in the same order, with no intervening signatures between s5 and s8. For each path of nodes entry associated with each passing path type, the node numbers that occur in the same query signatures positions in the Path of Signatures entry are returned as a join result. Path number p19 is the only path that has type t9. Result Nodes returned from this path are n29, n32 and n33 in the second, fourth and fifth positions respectively.

3.2.2 Check Relationships Algorithm for Recursive Schemas

Using the same notation in Section 3.2.1, a path type is considered one of the result path types if and only if all the following conditions hold:

- For each qi in \( \{q1, ..., qk\} \), qi occurs at least once in \( \{s1, ..., sh\} \).
- There exists a mapping \( r : [1...k] \rightarrow [1...h] \) such that \( qi = s_{r(i)} \) \( \forall 1 ≤ r ≤ k \) and for each Ri in R:
  - If Ri is a parent-child relationship between qi, qi+1, and qi = sj, we must have \( r(i) + 1 = r(i + 1) \).
Algorithm 1 Non Recursive Check Relationship Algorithm

Input:
- \( P = P_1, ..., P_m \): path types that contain all query labels
- \( R = R_1, R_2, ..., R_k \): are query relationships
- \( m : |P| \) and \( m \geq 1 \)
- \( k : |Q| \) and \( k \geq 1 \)

Internal Variables:
- \( P_{\text{pos}}_j \) is the position of query signature \( q_j \) inside the current path type
- \( S \) is the path of signatures of the current path type

Output:
- \( N = n_1, ..., n_r \) : \( n_i \) are query result node numbers satisfying query labels and structure, \( r \) is the number of result nodes per each path type and \( 1 \leq i \leq r \)

1: \( S = \text{getPathOfSignatures} (P) \);
2: for all \( i = 1 \) to \( d \) do
3: \( g = 0; \) satisfy = false;
4: for all \( i = 1 \leq |S| ; i + + , g + + \) do
5: if \( S[i] == q_j \) then
6: \( P_{\text{pos}}_j = i; \) \( g = i; \) break;
7: end if
8: if \( R_i == \text{PARENT} \&\& \text{CHILD} \) then
9: if \( S[i+1] == q_{j+1} \) then
10: satisfy = true; \( P_{\text{pos}}_{j+1} = i+1; \) \( g = i+1; \) break;
11: end if
12: end if
13: if \( R_i == \text{ANCESTOR} \&\& \text{DESCENDANT} \) then
14: for all \( u = g + 1; u < |S| ; u + + \) do
15: if \( S[u] == q_{i+1} \) then
16: satisfy = true; \( P_{\text{pos}}_{j+1} = u+1; \) \( g = u+1; \) break;
17: end if
18: end for
19: end if
20: end if
21: end for
22: end for
23: if satisfy == true then
24: \( N = \text{getPathOfNodes} (P) \);
25: for all \( i = 1 \) to \( k \) do
26: \( \text{return} N[P_{\text{pos}}_i] \);
27: \( \text{end for} \)
28: \( \text{end if} \)

- If \( R_i \) is an ancestor-descendant relationship between \( q_j, q_{j+1} \), and \( q_i = s_r(i) \), we must have \( r(i) + s = r(i + s) \) for some \( s \geq 1 \).

Since the query could be recursive, searching for the first query signature \( q_j \) inside path of signatures of the current path type \( S = \{ s_1, ..., s_n \} \), might result in more than one position for the same signature. Therefore, we first search for the first query signature which yields the construction of \( p \) position arrays \( P_{\text{pos}} = \{ P_{\text{pos}}_1, ..., P_{\text{pos}}_p \} \) for each path type where \( p \) is the number of times \( q \) occurs in the current path type. Each position resulting from searching for the first signature is assigned to the first element on the position array in the same order. Then the we check the rest of signatures, starting from the second signature, \( q_j \), where \( 2 \leq j \leq k \) and for every array constructed, if the current relation \( R_i \) is a parent-child relationship and the position stored in the current array is less than the position of the current signature by one, we store the position of the current signature in the current position array. If the current relation \( R_i \) is an ancestor-descendant relationship, the last position in the current array is less than the position of the current signature by one, we store the position of the current signature in the current position array.

3.3 Complexity Analysis for BPI Algorithm

The complexity analysis for algorithm \( \text{Intersect Path Types} \) algorithm is \( O(|P| \times |S|) \) where \( P \) is the count of path types in the XML document and \( |S| \) is the count of query signatures. The space requirement for this algorithm is \( O(|IP| \times |Q|) \) where \( IP \) is the count of intersection path types.

Time complexity for the \( \text{Check Relationships} \) algorithm for non-recursive schemas is \( O(|IP| \times h + |resultsize|) \) where \( |resultsize| \) is the total size (in bytes) of the result paths read out from the Path of Nodes index, \( IP \) are the intersection path types and \( h \) is the height of the XML document tree. Space Complexity for the \( \text{Check Relationships} \) algorithm for non-recursive schemas is \( O(|IP| \times h) \). Time complexity for \( \text{Check Relationships} \) algorithm recursive schemas is \( O(|IP| \times |Position_{IP}| + |resultsize|) \) where \( Position_{IP} \) are the result positions of signatures within each intersection path. Space Complexity for the \( \text{Check Relationships} \) algorithm for recursive schemas is \( O(|IP| \times |Position_{IP}|) \).

4. OPTIMIZATIONS

Our algorithm can benefit from several crucial optimizations discussed below.

4.1 Bitmap Representation

We observe that a central operation in the structural join computation above is an intersection (in the \( \text{Intersect Path Types} \) algorithm). We recall that intersections are most efficiently performed on bit operations. Since the \( \text{Signatures-PathTypes} \) index is a binary matrix, it begs for a bit representation. Bitmap indices are popular in data warehousing and in fact are frequently used as bitmap join indices. Bitmap indices offer two important advantages over conventional hash and tree indices. First, they allow the use of efficient bit operations to answer queries. Second, bitmap indices can be much more compact than a traditional tree index and are amenable to the use of compression techniques.

A bitmap representing the \( \text{Signatures-PathTypes} \) index is presented in Table 1. Rows in the bitmap represent signatures. Columns represent XML path types. The intersection between signatures in the \( \text{Intersect PathTypes} \) algorithm is done using a bitwise AND of the relevant rows. We call our algorithm Bit-mapped Path Index (BPI) because of this crucial optimization.

Definition 6 (Group). Each \( G \) path types (bits in this case) in the bitmap index is transformed to an integer and is called a group.

The \( \text{Intersect PathTypes} \) algorithm fetches each group rather than fetching individual path types. The bit positions inside the intersection result is used to extract path types in the same positions. Grouping paths and fetching the groups
Algorithm 2 Recursive Check Relationship Algorithm

Input:
- \( P = t_1, ..., t_m \) : path types that contain all query labels
- \( R = R_1, ..., R_k \) : are query relationships
- \( k \) : number of signatures within each path type, \( k \geq 1 \)

Internal Variables:
- \( P_{\text{path}} \) = is the position array number \( p \) that contains one of the position combinations of the output node numbers for the current path type
- \( S \) = contains the path of signatures of the current path type
- \( h \) = is the maximum depth for \( S \)

Output:
- \( N = n_1, ..., n_r \) : \( n_i \) are query result node numbers satisfying query labels and structure, \( r \) is the number of result nodes per each path type and \( 1 \leq i \leq r \)

1: \( S = \text{lastAdded}=0; p=0; \)
2: \( S = \text{getPathOfSignatures}(P); \)
3: for all \( u=1 \) to \( h \) do
4: if \( S[a] == q_1 \) then
5: \( Pos_p[0] = u;p++; \)
6: end if
7: end for
8: for all \( i=0 \) to \( k \) do
9: for all \( j=\text{Pos}_{n_0}[0] \) to \( h \) do
10: if \( q_i == S[j] \) then
11: if \( R_i = \text{PARENT}_\text{CHILD} \) then
12: for all \( y=1 \) to \( p \) do
13: if \( \text{Pos}_y[\text{lastAdded}] + 1 == j \) then
14: \( \text{Pos}_y[\text{lastAdded} + 1] = j; \)
15: \( \text{lastAdded}++; \)
16: end if
17: end for
18: else
19: if \( R_i = \text{ANCE}_\text{DESC} \) then
20: for all \( y=1 \) to \( p \) do
21: if \( \text{Pos}_y[\text{lastAdded}] < j \) and \( S[j] = S[j-1] \) then
22: construct \( \text{Pos}_{p+1}; \)
23: copy \( \text{Pos}_y[0 \to \text{lastAdded}] \) to \( \text{Pos}_{p+1}; \)
24: \( p++; \)
25: end if
26: if \( \text{Pos}_y[\text{lastAdded}] < j \) then
27: \( \text{Pos}_y[\text{lastAdded}+1] = j; \)
28: \( \text{lastAdded}++; \)
29: end if
30: end if
31: end for
32: end if
33: end if
34: end if
35: end for
36: end for
37: for all \( y=0 \) to \( p \) do
38: if \( \text{Pos}_y \neq 0 \) then
39: delete \( \text{Pos}_y[p-1]; \)
40: end if
41: end for
42: \( N = \text{getPathOfNodes}(P); \)
43: for all \( i=0 \) to \( p \) do
44: for all \( j=0 \) to \( \text{Pos}_{n_0} \) do
45: output \( R[N[\text{Pos}_y[j]]]; \)
46: end for
47: end for

Table 1: Signatures-PathTypes Index (Bitmap Index)

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>...</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>1</td>
<td>...</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>s2</td>
<td>1</td>
<td>...</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s3</td>
<td>1</td>
<td>...</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s4</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>s12</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Algorithm 3 Value-Predicate Evaluation Algorithm

Input:
- \( \text{value} \) : value predicate
- \( Q = q_1, ..., q_k \) : are query signatures
- \( R = R_1, ..., R_k \) : are query relationships
- \( k \) : number of query signatures and \( k \geq 1 \)

Internal Variables:
- \( PTIds \) : path type identifiers
- \( PIds \) : path identifiers
- \( Pos_i \) : query signatures positions in the path of signatures for path type identifier \( i \)

Output:
- \( S \) : an array of \( PTId, PId \) pairs

1: \( S = \text{getPathTypeIds}(value, PTIds, PIds); \)
2: for all \( \text{unique} PTId \) in \( S \) do
3: \( Pos_i = \text{checkRelationship}(PTId_i, R, Q); \)
4: end for
5: for \( S[i] \) in \( array \) \( S \) do
6: if \( Pos_i \neq 0 \) then
7: return PathofNodes[S[i].PId];
8: end if
9: end for

rather than fetching individual path types decreases the cost by a constant factor \( G \). We can further decrease the I/O cost by a blocking factor \( B \). Each \( B \) groups are stored in one array in the bitmap index as one entry. Groups are fetched in blocks of \( B \).

4.2 Value-Predicate Evaluation Plan

When selective attribute values are used, it makes sense to create a values index. The standard way to do this is to return nodes (i.e., node identifiers) that match the specified value, using this index. A smaller set of input nodes is thus sent to the structural join. In \( BPI \), there is no input set of nodes, so there is no obvious way to use values index and save effort. We derive a value filtering that can be applied to Path Types, using a new kind of values index that returns Paths rather than nodes.

Each entry in the \( Values \) Index contains a Path Type Identifier and a Path Identifier. Searching by a value will result in Path Type identifiers and Path identifiers for paths that contain this value. The result path type identifiers are sent to the \( Check \ Relationships \) algorithm that matches the query structure to the path type structures. The \( Check \ Relationships \) algorithm in this case will not return the node numbers, it will just match the structure and return the signatures positions inside the path type. Then the \( Value \ Predicate \ Evaluation \) algorithm extracts node numbers in the corresponding positions for the path of nodes related to the current path identifier. The pseudo code for this algorithm is presented in Algorithm 3.

The time complexity of the \( value \ predicate \ evaluation \) al-
algorithm is \(O(ValuePT1ds \times h + |resultpaths|)\); \(ValuePT1ds\) are path type identifiers for paths that contain the input value, \(h\) is the height of the XML document and the result paths are paths that have the same structure and value of the query. The space complexity of the value predicate evaluation algorithm is \(O(ValuePT1ds \times h)\). Like previous algorithms, this algorithm does not store intermediate path results.

5. INDEX CONSTRUCTION AND MAINTENANCE

5.1 Index Construction

The overall architecture of BPI structural join evaluation is presented in Section 5. The load function builds all indices described in Section 2.1. All indices specified can be constructed in one pass over the data. The XML document is parsed in depth first order. Each new label is given a new signature and each new node is given a node number. In path of signatures and path of nodes, the signatures and nodes respectively along each path are stored as an array of integers. Reaching a leaf node of a path triggers the insertion of a new entry in both indices. We keep track of paths with different types. Each new path type is given a path type identifier. The key for both indices is the path type identifier. The data is the array of integers representing signatures or node numbers. We use a B-tree index to store both indices. Reaching a leaf node of each new path type triggers the insertion of new entry in the Signatures-PathType index where we record the existence of each signature in the path type being parsed.

5.1.1 Complexity Analysis

The time needed to construct all of the aforementioned indices depends linearly on the number of XML nodes and the number of path types. The XML document is traversed once to build all indices. The information kept in memory at one point of time is the nodes on the path being parsed, its signatures, its path type and all previous path types. The number of path types is small in practice, so this information is likely to fit in memory, even for very large data sets.

5.2 Index Maintenance

Bitmap indices are notorious for being hard to maintain. In consequence, they are rarely used in updatable database systems – they are typically used only in data warehouses where data is batch loaded. There are two reasons to worry about the cost of bitmap indexing, in general. One has to do with the number of indices – often multiple bitmap (and bit slice) indices are required to perform the function of a single conventional index. Another worry is that the bitmap assumes a certain order of tuples (or other indexed entities). If a tuple is deleted, we are left with a hole in the bit map.

In the case of BPI, these difficulties are avoided. First, there is only one bitmap index – not several. Second, there is an explicit lookup table, Path of Signatures index, to map from the bitmap index position to the XML path type represented. The existence of this lookup table permits us to fill holes in the bit map as they arise, and even to reorganize bitmap index allocations, without having to touch the actual paths. Furthermore, note that path types are stored in the bitmap index and not individual paths.

It is also worthwhile to compare these low costs for incremental index maintenance with BPI to traditional labeling techniques for XML structural joins. Most XML label assignments, which are at the core of algorithms in the stack-based family, are static, with high costs to change. There has been some limited innovative work on dynamic label assignment [10, 25]. But these schemes tend to be quite complicated, and have not yet found widespread use. Thus, the BPI technique actually does better with incremental maintenance than traditional techniques.

6. PERFORMANCE EVALUATION

In this section, we evaluate the performance of the proposed structural join algorithm, BPI. We compare BPI performance with the state of the art LCS-TRIM algorithm [22]. We chose LCS-TRIM as one base line because it is regarded as one of the “best” structural join algorithms proposed thus far. We are grateful to the inventors of LCS-TRIM for sharing their code with us to enable this comparison. All experiments were run on a 2GHz CPU with 1 GB RAM running WinXP.

6.1 Absolute Time Comparison

We used two popular XML data sets: DBLP [11] and SwissProt [21]. The data set characteristics are presented in Table 5. As we can see, the size of the bitmap index structure is only a small fraction of the size of the data set; less than \(10^{-4}\). In other words, the space overhead for the Signatures PathTypes index to support BPI is negligible. Number of path types is very small compared to the number of paths in the XML document. Path of Signatures index is also small relative to the data set. This is because it is storing path type signatures. The path of nodes index is used only to report output. Indeed, as we see below, most of the evaluation time is taken in reporting node numbers to the output. Against both data sets we ran a number of queries both with and without value-based selection as shown in Tables 2 and 3.

Figure 4(a) presents the evaluation times for the first six queries of the DBLP query set. The input to these queries varies from 32000 to 920000 nodes. We find that the improvement of BPI over LCS-TRIM for most queries is more than 40%. The improvement in \(Q1\), \(Q2\) and \(Q3\) is more than 80%. That is because \(Q1\), \(Q2\) and \(Q3\) have small output relative to the rest of the queries. We see the strong effect of input size on LCS-TRIM evaluation times. For instance \(Q2\) and \(Q3\), the input of \(Q2\) is 107002 and \(Q3\) is 32983. LCS-TRIM performs better in \(Q3\) although it produces larger output than \(Q2\).

As the output cardinality increases, the time for query evaluation in BPI increases. When the output is small, the number of intersection path types is likely to be small and checking relationships is performed over that small number of path types. As the output becomes larger, the improvement of BPI over LCS-TRIM becomes smaller. But even with queries that have larger output, BPI still has a good of improvement. For instance, in the last query, \(Q6\) with output more than half the size of the input, the improvement is 68%.

Figure 4(c) shows the query evaluation time for the DBLP queries with value predicates. The existence of value predicates decreases the size of the output. Therefore, these queries have smaller output size. The improvement of BPI
over LCS-TRIM for all queries is more than 89% except Q11. The improvement in Q11 is 62%. This is due to Q11 having a larger output size. Consider queries Q8 and Q10. The input size for Q8 is 719495 and the output is only 12 results. While the input size for Q10 is 329616 which is much smaller than Q8 and the output size is 36 results, LCS-TRIM performs in query Q10 better than Q8 in spite of the larger output in Q10. On the other hand, BPI performs better in Q8.

Since DBLP data set is shallow, it is hard to define many meaningful chain queries. For the SwissProt data set, the results obtained for single pair joins were similar to those for DBLP. As such, we focus our reporting here on chain queries, shown in Table 3. Figure 4(b) shows the queries evaluation time. BPI performs better than LCS-TRIM in all queries by more than 50% in query evaluation time. For instance, in Q12, the improvement of BPI over LCS-TRIM is 65% and in Q14 it is 62%.

### 6.2 Input Cardinality Effect

The experiments described above suggest that the time taken by BPI does not depend on the input size, unlike LCS-TRIM. However, too many variables change between queries for us to be completely sure this is true. To isolate the effect of input size on query evaluation time, we introduce a synthetic XML document that has 84001 element, 8 path types, 46000 paths, 10 labels and a depth of 3. The DTD for this XML document is presented in Figure 5(a). The cardinality of element Y is 8000, distributed evenly between the four parents X, A, C and E. The queries used in this experiment are presented in Table 4. To show input effect, we fixed the output for all queries. However, the cardinalities of elements X, A, C and E are all different, leading to different input cardinalities. It is clear in Figure 5(b) that BPI has the same query evaluation time for all queries even though the input size is varied from 4000 nodes to 22000 nodes.

### 6.3 Tree Fan-out and Query Level Effect

In this section, we show the effect of varying depth and tree fan-out over query evaluation time. Documents for the level and fan-out effect experiment were generated using Toxgene [2]. The DTDs for these documents are presented in Figure 6. Characteristics of the generated documents are presented in Table 6. If the cardinality of all elements in these three documents is 3000, we need to check 1 path type in the first document in Figure 6a to retrieve the output for the query //node2//node3. There will be 3000 result path of nodes associated with this path type. However, in Figure 6b we need to return 6000 paths because node2 and node3 contribute in 6000 paths; half of them ends with node4 and the other half ends with node5. The maximum number returned from any query is:

\[ ([\text{fanout}]^{h-\text{level}(f)} \times \text{cardinality}(f)) - R \]

where h is the height of the tree and f is the first label in the query, R is the number of paths after subtraction of the intersection path types that contain the contributing signatures.

Figure 7 shows the evaluation of the query node2//node3 over the three XML documents. It is clear that whenever both query label’s level and the fan-out increases, the query evaluation time increases. This is because when query labels exist in tree levels other than the leaf level, it may contribute to more than one path and the extra paths must be returned in the output. Although the difference between the three documents query evaluation times is small, it is clear that there is a height and fan out effect on the query time evaluation.

### 6.4 Query Execution Breakdown

Most of the query time evaluation is spent on fetching nodes of each path associated with the intersection path types. In Figure 8, DBLP queries are used. We mean by

---

**Table 2: DBLP CHAIN Query Set**

<table>
<thead>
<tr>
<th>Q#</th>
<th>Query Expression</th>
<th>#Input</th>
<th>#Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>//journal//number</td>
<td>218616</td>
<td>0</td>
</tr>
<tr>
<td>Q2</td>
<td>//phdthesis//number</td>
<td>107002</td>
<td>3</td>
</tr>
<tr>
<td>Q3</td>
<td>//phdthesis//year</td>
<td>32893</td>
<td>92</td>
</tr>
<tr>
<td>Q4</td>
<td>//article//year</td>
<td>430440</td>
<td>117069</td>
</tr>
<tr>
<td>Q5</td>
<td>//article/author</td>
<td>828097</td>
<td>221465</td>
</tr>
<tr>
<td>Q6</td>
<td>//proceedings/author</td>
<td>928761</td>
<td>491783</td>
</tr>
<tr>
<td>Q7</td>
<td>//article/author = “Antonin Guttman”</td>
<td>327558</td>
<td>3</td>
</tr>
<tr>
<td>Q8</td>
<td>//proceedings/author = “E. F. Codd”</td>
<td>719495</td>
<td>12</td>
</tr>
<tr>
<td>Q9</td>
<td>//article/author = “M. Tamer Zsu”</td>
<td>327558</td>
<td>30</td>
</tr>
<tr>
<td>Q10</td>
<td>//book/year = 1995</td>
<td>329676</td>
<td>36</td>
</tr>
<tr>
<td>Q11</td>
<td>//article/year = 1999</td>
<td>440440</td>
<td>7408</td>
</tr>
</tbody>
</table>

**Table 3: SwissProt CHAIN Query Set**

<table>
<thead>
<tr>
<th>Q#</th>
<th>Query Expression</th>
<th>#Input</th>
<th>#Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q10</td>
<td>//Entry//Features/Signal/Descr</td>
<td>5261</td>
<td></td>
</tr>
<tr>
<td>Q14</td>
<td>//Entry/Features/CHAIN/Descr</td>
<td>16947</td>
<td></td>
</tr>
<tr>
<td>Q14</td>
<td>//Entry/Ref/Comment</td>
<td>51852</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4: Query Set for the Synthetic Data Set**

<table>
<thead>
<tr>
<th>Query Expression</th>
<th>#Input</th>
<th>#Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>X/Y</td>
<td>4000</td>
<td>2000</td>
</tr>
<tr>
<td>A/Y</td>
<td>7000</td>
<td>2000</td>
</tr>
<tr>
<td>C/Y</td>
<td>12000</td>
<td>2000</td>
</tr>
<tr>
<td>E/Y</td>
<td>22000</td>
<td>2000</td>
</tr>
</tbody>
</table>

**Table 5: Data set characteristics**

<table>
<thead>
<tr>
<th>DataSet</th>
<th>FileSize</th>
<th>PathofNodes</th>
<th>PathofSignatures</th>
<th>Signatures-PathType</th>
<th>#paths</th>
<th>#path Types</th>
<th>#labels</th>
<th>#groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>130MB</td>
<td>1651MB</td>
<td>16KB</td>
<td>16KB</td>
<td>3005840</td>
<td>116</td>
<td>35</td>
<td>4</td>
</tr>
<tr>
<td>SwissProt</td>
<td>112MB</td>
<td>1238MB</td>
<td>16KB</td>
<td>16KB</td>
<td>204844</td>
<td>44</td>
<td>85</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 5:** (a) Document DTD. (b) BPI execution time is independent of input cardinality.
7. RELATED WORK

Structural join algorithms have been extensively studied in the literature [12, 19, 20, 24, 17, 5, 9, 16, 14]. [26] proposed a merge-based join algorithm called multi-predicate merge join MPMMCIN. [8] utilized the B+ tree index on the algorithm to skip descendant nodes which do not participate in the join. [13] proposed XR-Tree to skip both ancestors and descendants nodes that do not participate in the join. [1] proposed Stack-Tree-Desc/Anc algorithm which uses a stack to improve the merge-based structural join algorithm. All these algorithms depend on the query input size. Note, that the size of input lists for the stack-based structural join algorithms will also be linear in the size of the database.

The P-Filter algorithm in [17] pre-filters nodes and paths that are not part of the result. It performs a structural join operation for rest of the nodes. Space requirement for this technique is dependent on the number of nodes in the XML document. There are two indices. The first one, LPTree, stores paths in the XML document and stores all XML elements that can be reached by each path. The other index, UPET, contains an index over the LPTree index. The UPET index can fit in memory. A join based on containment relationship is done over the UPET index to filter out the nodes that have different labels. Another join is done over the LPTree index to filter out paths. The resulting nodes are structurally joined together with any basic structural join algorithm.

The difference between BPI and P-Filter algorithm is that BPI pre-filters all nodes and paths in one intersection operation over a bitmap index which is much faster than doing a multiple containment join for each result node from the UPET index inside the LPTree index. The other difference is that P-Filter uses any structural join algorithm to obtain the output. In our algorithm we do not need to join the rest of the nodes because they are the output; we output them directly. Therefore, we expect BPI to be faster than P-Filter algorithm.

The use of indices to speed up structural join evaluation has been considered in previous work [3, 8, 17]. None of these papers explicitly discuss the size of their indices. However, one can see that the sizes of indices in these papers are all dependent on the number of nodes in the XML document. On the other hand, in BPI, the size of the Signatures-Pathtypes index is negligible compared to the XML data size. The path of nodes index that we use to fetch the data nodes has a size that is dependent on the number of leaf nodes but this is consulted only at the final output stage.

Recently, new approaches have been introduced such as VIST [23], PRIX [18] and LCS-TRIM [22]. These approaches rely on subsequence matching for the query sequence inside the data sequence to get the result of the join. This is done in two steps, label matching and structure matching. In order to do the matching, some indices are built over the data sequences either on the fly indices or stored indices. If the index is built on the fly as the case in LCS-TRIM, it takes a long time the data set, building
new algorithm, ships between XML elements. In this paper, we propose a
8. CONCLUSION
have also shown that
out storing or sorting lists of the contributing nodes. We
algorithms, we are able to evaluate the structural join with-
index which results in a small evaluation time. Unlike other
algorithms. It is based on using intersection operation in
9. REFERENCES
![Table 6: Documents Characteristics](image)

<table>
<thead>
<tr>
<th>#elements</th>
<th>Doc1</th>
<th>Doc2</th>
<th>Doc3</th>
</tr>
</thead>
<tbody>
<tr>
<td>#paths</td>
<td>3000</td>
<td>3000</td>
<td>2700</td>
</tr>
<tr>
<td>#groups</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fan-out</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Depth</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

the index and then applying the matching algorithm. In
addition to the high maintenance cost for the built indices.
Other drawbacks comes from the data representation itself.
For instance, the worst case storage requirement in VIST for
a B-tree index used is higher than linear in the total num-
ber of nodes of the XML document. For example, consider a
unary tree with n nodes. In this case, the total size of the
structure-encoded sequence is $O(n^2)$.

![Figure 8: Query Execution Breakdown](image)

8. CONCLUSION
XML structural joins evaluate the containment relationships
between XML elements. In this paper, we propose a
new algorithm, BPI, to evaluate structural joins. It requires
only a limited amount of storage space and is capable of
generating outputs independent of input size, unlike legacy
algorithms. It is based on using intersection operation in
evaluating the join. The intersection is done over a bitmap
index which results in a small evaluation time. Unlike other
algorithms, we are able to evaluate the structural join with-
out storing or sorting lists of the contributing nodes. We
have also shown that BPI can be optimized using bitmap
indexes and grouping index entries together. In our exper-
iments, we have also presented the effect of changing the
grouping factor and changing query labels depths. Our ex-
periments have shown that BPI is faster than LCS-TRIM
join algorithms by more than 80% in most of the cases and
in cases where the output is large, BPI performs faster by
more than 50%.

9. REFERENCES
[1] 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
[2] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and
K. A. Lyons. ToXgene: An extensible template-based
data generator for XML. In WebDB, pages 49–54,
2002.
query optimization using path indexes. In XIME-P,
2004.
Indexing xml data for efficient handling of branching
joins: optimal XML pattern matching. In
efficient xpath processing system. In SIGMOD, pages
and C. Zaniolo. Efficient structural joins on indexed xml
and C. Zaniolo. Efficient structural joins on indexed XML
adaptive path index for xml data. In SIGMOD’02:
Proceedings of the 2002 ACM SIGMOD international
conference on Management of data, pages 121–132,
New York, NY, USA, 2002. ACM.
[11] DBLP computer science bibliography dataset:
XML data using an rdmb. IEEE Data Engineering
Indexing XML data for efficient structural join. In
joins on indexed xml documents. In VLDB, pages
of xml indexes for structural join. SIGMOD Record,
[16] Q. Li and B. Moon. Indexing and querying XML data
for regular path expressions. In The VLDB Journal,
Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald.
Efficiently publishing relational data as XML
[20] J. Shamugasundaram, K. Tuftu, C. Zhang, G. He,
D. J. DeWitt, and J. F. Naughton. Relational
databases for querying XML documents: Limitations
[21] SwissProt curated protein dataset:
Lcs-trim: Dynamic programming meets xml indexing
dynamic index method for querying xml data by tree
structures. In SIGMOD Conference, pages 110–121,
2003.

