Bitmap Indexes for Relational XML Twig Query Processing

Kyong-Ha Lee
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
bart7449@gmail.com

Bongki Moon
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
bkmoon@cs.arizona.edu

ABSTRACT

Due to an increasing volume of XML data, it is considered prudent to store XML data on an industry-strength database system instead of relying on a domain specific application or a file system. For shredded XML data stored in the relational tables, however, it may not be straightforward to apply existing algorithms for twig query processing, because most of the algorithms require XML data to be accessed in a form of streams of elements grouped by their tags and sorted in a particular order. In order to support XML query processing within the common framework of relational database systems, we first propose several bitmap indexes for supporting holistic twig joins on XML data stored in the relational tables. Since bitmap indexes are well supported in most of the commercial and open-source database systems, the proposed bitmap indexes and twig query processing algorithms can be incorporated into the relational query processing framework with more ease. The proposed query processing algorithms are efficient in terms of both time and space, since the compressed bitmap indexes stay compressed during query processing. In addition, we propose a hybrid index which computes twig query solutions with only bitvectors, without accessing labeled XML elements stored in the relational tables.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Language—*Query languages*; H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Languages, Experimentation, Performance

1. INTRODUCTION

Due to its simplicity and flexibility, XML is widely adopted for information representation and exchange. With the growing popularity and increasing volume of XML data, much research has been done for managing XML data with relational database systems [5, 16, 12, 4]. For example, a *node approach* [16, 15, 12, 3] has been suggested as a way of mapping XML data into relational tables. In this approach, each XML node (*i.e.*element, attribute) is labeled and stored as a row in a relational table. To assist query processing, there may also be supplementary tables that store such information as path-materialized strings [15].

An XML query is commonly modeled as a *twig pattern* whose nodes are connected by either *Parent-Child* (P-C) or *Ancestor-Descendant* (A-D) axes. Most existing approaches to twig pattern matching are based on decomposing a twig into several linear paths. Once all the instances matching the linear paths are found from XML data (by virtue of a labeling scheme or a structural summary), they are joined together to find the instances matching the twig pattern as a final result. Numerous efficient algorithms for twig query processing have been reported in the literature, and some of them have been proven to be I/O optimal for a certain class of twig queries. Readers are referred to a recent survey for various XML query processing techniques [3].

One of the common assumptions most twig query algorithms rely on is that input data are stored and accessed as a form of inverted lists whose items are grouped by certain criteria and sorted in a specific order [1, 2, 10]. Since a storage scheme like that is not genuinely supported by relational database systems, the twig query algorithms cannot be directly applied without preprocessing XML data stored in tables. Typically, relevant elements are retrieved from tables, partitioned by their tags and then sorted by document order of the elements or an order given by a labeling scheme (*e.g.*, interval based [7, 16]).

For example, to answer a twig query Q shown in Figure 2(a) for XML data stored in relational tables shown in Figure 3(b), Q is first decomposed into 3 paths(*i.e.*`//A`, `//A//B` and `//A//C`) and SQL queries are executed for the linear paths. Then, the path solutions are joined together to obtain the final result for Q.

EXAMPLE 1. *Pseudo SQL statements to answer a twig query in Figure 2(a):*

```
path_sol1 ← (select * from node N, path P  where
   N.pid = P.pid and P.pathString like "A%"
   order by N.start);
path_sol2 ← (select * from node N, path P where
   N.pid = P.pid and P.pathString like "B%A%"
   order by N.start);
path_sol3 ← (select * from node N, path P where
   N.pid = P.pid and P.pathString like "C%A%"
   order by N.start);
select * from path_sol1 A, path_sol2 B, path_sol3 C
```


CIKM'09, November 2–6, 2009, Hong Kong, China

```
where A.start < B.start ∧ A.start < C.start ∧
      A.end > B.end ∧ A.end > C.end;
```

To facilitate XML query processing with relational database systems, we propose several bitmap based indexes for XML data. A bitmap index has been known as a powerful tool for OLAP queries and supported by most commercial and open-source database systems. Since data items with a specific value from a discrete domain can be found quickly by accessing a corresponding bit-vector, a bitmap index will be a very effective tool for retrieving relevant XML elements separately by their tags or other criteria, so that twig queries can be processed efficiently for XML data stored in relational tables.

We first present various *cursor-enabled* bitmap indexes built on different domains, namely, (1) ***tag***, (2) ***path***, (3) ***tag+level***. These bitmap indexes will provide quick access to relevant XML data stored in a table. Their optimization techniques will be discussed as well. We also present two hybrid indexes. **bitTwig**, one of the hybrid indexes, is distinguished from most existing ones in that **bitTwig** identifies element relationships with only a pair of cursor-enabled bitmap indexes. All twig instances matching a given query can be found even without accessing XML data (*e.g.*, labels associated with XML elements) stored in the relational tables. Since the amount of data fetched from the database can be minimized, this index improves the performance of twig query processing considerably. Our experiments show that **bitTwig** outperforms existing indexes in most cases.

The remainder of this paper is organized as follows. Section 2 reviews existing holistic twig join algorithms and XML storage schemes for RDBMS. Section 3 and Section 4 introduce our indexes and a way to determine the element relationship using bit-vectors. Section 5 describes query processing algorithms with the indexes. Section 6 presents our experimental results. Related work is presented in Section 7.

2. BACKGROUND

2.1 Twig Patterns and Holistic Twig Join

Answering a twig pattern query, whose nodes are connected by a /-axis or a //-axis, is to find all matching occurrences of the pattern in XML documents. The solution to a twig query Q with k nodes is given by a set of k-ary tuples, each of whose elements represents an XML element stored in the relational tables. Figure 1 shows a sample XML document, each of whose elements are augmented by a document-ordered value (*e.g.*, 0 for the element a_1) and a label written in interval-based labeling scheme (*e.g.*, (1, 32, 1) for the element a_1). Figure 2 shows examples of twig pattern queries.

Most holistic twig join algorithms [9, 2] use an interval-based labeling scheme [7, 16] to determine the relationship between two XML nodes. In addition, these algorithms require XML data to be accessed in a form of streams of elements grouped by their tags, associated with their labels, and sorted in a particular order. They typically operate in two steps. A twig pattern is first decomposed into multiple linear paths and the solutions to the paths are then computed. The intermediate results from the linear paths are then joined together for the final result.

Note that holistic join algorithms typically return as a final query result a set of k-ary of labels in the form of (*startPos, endPos, level*) without including actual XML data. To return actual XML data for a given twig query, the references to matching XML data stored in the relational tuples need to be provided as well.

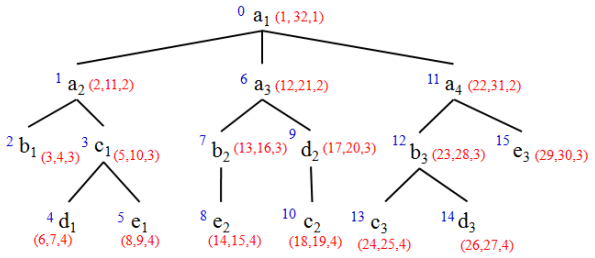


Figure 1: A Sample XML Document

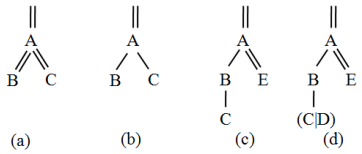


Figure 2: Twig Pattern Queries

2.2 XML Storage Scheme in DBMS

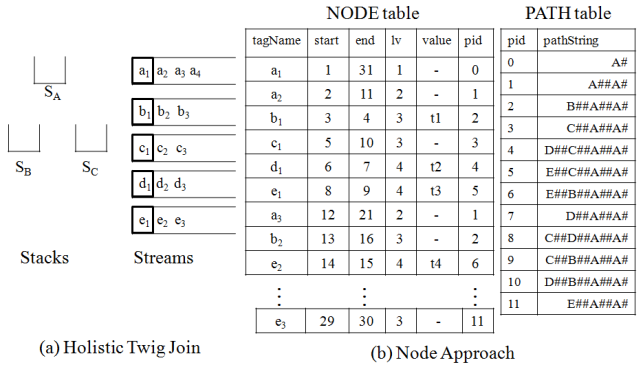
We assume that XML data are stored in relational tables as suggested in the *node approach* [16]. Each XML element is labeled by a labeling scheme (*e.g.*, interval-based labels) and stored as a row in a table (known as a *node table*) as shown in Figure 3(b)). The structural relationship between two elements (or rows) stored in the table is determined by labels associated with the elements. The node table can be joined with a supplementary table (known as a *path table*) that stores path-materialized strings (shown in Figure 3(b)) to assist query evaluation further.

The node approach allows XML data to be indexed by traditional indexing schemes provided by RDBMS [12]. If XML data stored in the node table are indexed by bitmaps, a twig query can be processed more efficiently by using the bitmap indexes alone or together with the node table rather than joining the node table with the path table. The bitmap indexes and query processing algorithms will be presented in Section 3 and Section 5 in more detail. For bitmap indexing and query processing, we further assume that XML elements are stored in the table in the *document order* (as shown in Figure 3(b)), which can be naturally maintained without additional processing just by following order the elements appear in an XML document.

3. INDEX STRUCTURES

3.1 Basic Bitmap Indexes

A bitmap index is a collection of *bit-vectors*. The bit-vectors in a bitmap index correspond to distinct values in a certain value domain. The 1's in a bit-vector identify a group of XML elements (or tuples in a table) having a specific value corresponding to the bit-vector. The *value domain* can be a set of XML tags, a set of distinct paths, and so on. For example, if a bit-vector represents a tag name A, then the 1's in the bit-vector identify all XML elements whose tag name is A with their row-ids in the relational table (shown in Figure 3(b)).



tagName	start	end	lv	value	pid
a_1	1	31	1	-	0
a_2	2	11	2	-	1
b_1	3	4	3	t1	2
c_1	5	10	3	-	3
d_1	6	7	4	t2	4
e_1	8	9	4	t3	5
a_3	12	21	2	-	1
b_2	13	16	3	-	2
e_2	14	15	4	t4	6
⋮		⋮			⋮
e_3	29	30	3	-	11

pid	pathString
0	A#
1	A##A#
2	B##A##A#
3	C##A##A#
4	D##C##A##A#
5	E##C##A##A#
6	E##B##A##A#
7	D##A##A#
8	C##D##A##A#
9	C##B##A##A#
10	D##B##A##A#
11	E##A##A#

Figure 3: Two Different Storage Schemes

For twig query processing, we first propose three basic bitmap indexes that can be defined on different value domains: *bitTag*, *bitPath* and $bitTag^+$. The indexes are used to determine the row-ids of the elements, required for a given query, stored in a relational table.

The ***bitTag*** is a bitmap index whose bit-vectors correspond to distinct tag names. This index can be obtained by taking the `tagName` column in Figure 3 as a value domain. exactly one bit-vector will be accessed from a *bitTag* index per query node index during query evaluation.

The ***bitPath*** is a bitmap index whose bit-vectors correspond to distinct paths. This bitmap index can be obtained by taking the `pid` (*i.e.*, path id) column in Figure 3 as a value domain. A 1 in a bit-vector indicates a terminal node (*i.e.*, opposite to the root node) in a linear path corresponding to the bit-vector. From a *bitPath* index, multiple bit-vectors per query node may be accessed during query evaluation, since a query with `//`-axis may select multiple linear paths (*e.g.*`/A` and `/A/A` for a given path `//A`).

The $\boldsymbol{bitTag^+}$ is a bitmap index based on a tag+level partitioning, as suggested in *iTwigJoin* [2]. XML elements are first partitioned by their tags, and the elements in each partition is again partitioned by their levels. In other words, $\boldsymbol{bitTag^+}$ is a bitmap index whose bit-vectors correspond to distinct pairs of (`tag`, `level`).

Note that the *bitTag* consists of the smallest number of bit-vectors among the indexes. The $bitTag^+$ has more bit-vectors than *bitTag* due to further partitioning by levels, and the *bitPath* has the largest number of bit-vectors. Obviously, the sizes of the bitmap indexes become large as the cardinality of the column which the indexes are built over grows higher. However, since bit-vectors are usually compressed, the size of a bitmap index can be remarkably shrunk. Typically, bit-vectors tend to be sparser if the cardinality grows higher, which makes bit-vectors more compressible.

The number of bit-vectors required for processing a twig query will also be different depending on which index is used. For example, we have three bit-vectors in a *bitPath* index, all which correspond to a query node C in a twig pattern shown in Figure 2(a). Because the path for C is `//A//C` and path ids of corresponding path strings are 3, 8 and 9 in the path table in Figure 3. If a *bitTag* index is used instead, there will be only one bit-vector required for the query node C. (This input scheme is identical to the way of twigStack). In our algorithms, if more than one bit-vector is required per query node, the bit-vectors can be either merged into a bit-vector by bitwise-OR operations or all the required bit-vectors can be accessed at the same time by using their own cursors. (see Table 1 for summary)

3.2 More Indexes for A-D Relationship

The basic bitmap indexes presented above are primarily for representing XML elements having a certain tag, a certain path, or a pair of (`tag`, `level`). To help determine the A-D relationship between elements identified by a *bitPath* bit-vector, we propose two additional bitmap indexes.

The ***bitAnc*** index consists of a set of bit-vectors, one per each distinct path in an XML document. The 1's in a bit-vector represent a set of ancestor nodes of a terminal node in the corresponding path and the terminal node itself. That is, a bit-vector of a *bitAnc* index covers all nodes constituting a linear path that the bit-vector represents. For example, the 1 bit at position 2 in a *bitPath* bit-vector in Figure 4(a)) represents node b_1, a terminal node in a path `/A/A/B`. In contrast, for a *bitAnc* bit-vector, the first and the second bits are also set to 1 to represent b_1's ancestors, a_0 and a_1.

The ***bitDesc*** is the same as the *bitAnc* index except that the bit-vectors represent descendants rather than ancestors. In Figure 4(a), the bit-vector of a *bitDesc* index represents the descendants of all terminal nodes for the path `/A/A/B`. For example, the position 8 of the *bitDesc* bit-vector is set to 1 to represent b_2's descendant, e_2.

Figure 4(b) shows a subtree represented by the three bit-vectors shown in Figure 4(a). This illustrates that a subtree corresponding to a given path can be projected from an original XML document using the bit-vectors. We utilize this property for twig query processing.

3.3 Properties of Bitmap Indexes

Bitwise operators such as bitwise-AND and bitwise-OR (denoted by $\odot$ and $\oplus$, respectively) are associative and commutative. While many rules may be derived for the indexes, we summarize only the most useful properties for our bitmap indexes in the following lemma. We denote a bit-vector of a certain index type by $indexName(value)$, where *value* is the column value the bit-vector corresponds to. For example, $bitTag(a)$ denotes a bit-vector of a *bitTag* index corresponding to a tag name a, and $bitTag^+(a, l)$ denotes a bit-vector of a $bitTag^+$ index corresponding to a tag name a on level l.

LEMMA 1. *The proposed bitmap indexes satisfy the following properties.*

$$bitTag(a) \equiv \bigoplus_{level l=1}^{height(doc)} bitTag^+(a,l) \equiv \bigoplus_{i \in end(a)} bitPath(i) \quad (1)$$

$$\bigoplus_{i \in pids(s)} bitPath(i) \equiv \partial_{path=s}(bitTag(a)), leaf(s) = a \quad (2)$$

$$bitPath(i) \equiv bitAnc(i) \odot bitPath(i) \equiv bitDesc(i) \odot bitPath(i) \quad (3)$$

$$(\bigoplus_{i \in pids(s)} bitDesc(i)) \odot bitTag(a) \equiv \partial_{a \in \triangle_{pids(s)}}(bitTag(a)), leaf(s) = a \quad (4)$$

Here, $\triangle_i$ represents subtrees, which is pruned from a whole XML document, rooted at tags corresponding to a path id i. $end(a)$ returns a set of pids of path strings which end with a tag name a. $leaf(s)$ returns the tag name of the terminal of a path string s and $pids(s)$ returns a set of path ids corresponding to a path string s.

Index	$bitTag$	$bitPath$	$bitTag^+$
Inputs per query node	single bit-vector& labels	single bit-vector & labels	multiple bit-vectors & labels
Type of bit-vector	$bitTag(a)$	$\bigoplus_{i \in leaf(a)} bitPath(i)$	$bitTag^+(a, l)$
Optimization	$advance(k)$	$condense(q)$	$advance(k)$

Table 1: Summary of Basic Indexes

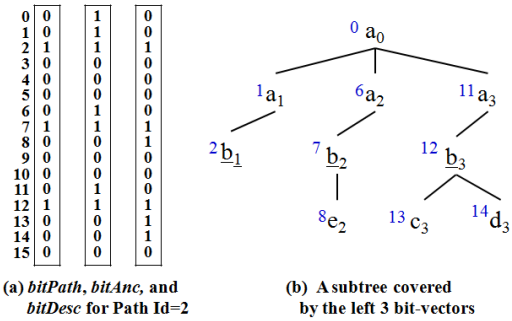


Figure 4: Example of A-D Indexes

Property (1) allows multiple bit-vectors to be merged into a single vector that constitutes a tag-based input for twig join algorithm. Property (2) states that a bit-vector generated by merging all the $bitPath$ bit-vectors satisfying a given path string s is identical to a $bitTag$ bit-vector corresponding to a tag $end(s)$. By property (3), bit-vectors in $bitAnc$ and $bitDesc$ indexes contain terminal nodes for a given path i as well, and they can be identified by virtue of $bitPath$. Finally, property (4) shows that with a given path s, we can get a bit-vector which covers nodes, whose tag name is a, in a subtree rooted $leaf(s)$ by merging $bitDesc$ and $bitTag$ with $\odot$.

4. CHECKING ELEMENT RELATIONSHIP WITH BIT-VECTORS ONLY

Twig query processing with the basic indexes (presented in Section 3.1) requires the use of additional information such as labels in order to determine the relationship between elements. This is because the bitmap indexes provide no more information than row-ids of elements corresponding to a certain value. In this section, we introduce a hybrid index called $bitTwig$, which consists of two different bitmaps. With the $bitTwig$ index, we can determine the relationship between elements without resorting to the labels. Since XML nodes are stored in a table in document order, we can check preceding and following relationship by simply inspecting two corresponding 1's positions in bit-vectors. A $bitAnc$ index covers all ancestors of terminal nodes specified by a $bitPath$ index. Thus we can identify A-D and P-C relationship with a pair of $bitPath$ and $bitAnc$, instead of interval-based labeling scheme.

We let $indexName(v)$ and $indexName(v)[k]$ denote a bit-vector of an $indexName$ bitmap for a given value v and the k-th bit of the bit-vector, respectively. The path identification corresponding to an element a is denoted by $pathid(a)$. In addition, we use $pos(a)$ to denote the document order of an element a.

THEOREM 1. *For two distinct elements a and d in the same document, a is an ancestor of d if and only if the following conditions are satisfied.*

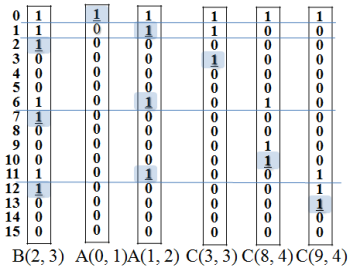


Figure 5: Identifying A-D relationship with bit-vectors

(1) $pathid(a) \neq pathid(d)$,

(2) $pos(a) < pos(d)$,

(3) $bitAnc(pathid(d))[pos(a)] = 1$,

(4) $bitPath(pathid(a))[k] = 0, \forall k : pos(a) < k < pos(d)$.

PROOF. ($\Longrightarrow$) To show that they are the necessary conditions for a to be an ancestor of d, we need to show that a is not an ancestor of d if any of the four conditions is not satisfied. If condition (1) is not satisfied, a cannot be an ancestor of d because their path strings are identical. If either condition (2) or condition (3) is not satisfied, a is obviously not an ancestor of d, by the document order restriction or the definition of the $bitAnc$ index. If condition (4) is not satisfied, there exists an element x such that $bitPath(pathid(a))[pos(x)] = 1$ and $pos(a) < pos(x) < pos(d)$. Since $bitPath(pathid(a))[pos(a)] = 1$, the path strings of a and x are identical, which makes x come after all the descendants of a in the document order. Since $pos(x) < pos(d)$, d cannot be a descendant of a either.

($\Longleftarrow$) We can show that they are the sufficient conditions for a to be an ancestor of d by contradiction. Suppose a is not an ancestor of d when all the four conditions are satisfied. Since a is not an ancestor of d, the relationship between a and d in the document hierarchy must be one of the three: (i) a is a descendant of d, (ii) a and d are not related and a comes after d in the document order, (iii) a and d are not related and a comes before d in the document order. The relationships (i) and (ii) both contradict the condition (2). Suppose a and d are in the relationship (iii). The condition (3) dictates that a is on a certain path (not necessarily as a terminal node) whose path string is identical to the path string of d. Since a and d have different path strings by the condition (1), the path string of a must be a prefix of the path string of d. This implies that there exists an element y such that y is an ancestor of d and the path string of y is identical to the path string of a. This in turn implies $bitPath(pathid(a))[pos(y)] = 1$, and contracts the condition (4), because $pos(a) < pos(y) < pos(d)$. Therefore, a must be an ancestor of d, if all the four conditions are satisfied. □

A sample use of the index pair for answering twig queries in Figure 2(a) is shown in Figure 5. In Figure 5, each label

under a bit-vector indicates tag name, path id and level the bit-vector represents. For better understanding, we illustrate a pair of bit-vectors as one by denoting terminal nodes as underlined 1's in the figure. We identify which bits in $bitAnc(i)$ represents terminal nodes by virtue of $bitPath(i)$ during query evaluation.

EXAMPLE 2. *With pairs of bit-vectors in Figure 5, we check A-D relationship between a_2 and c_1 and between a_2 and c_2 in Figure 1. Here, a_2, c_1 and c_2's bit positions are 1, 3 and 10, respectively as shown in Figure 1. We can decide that a_2 is an ancestor of c_1 exists, since they satisfy all the conditions in Theorem 1. However, a_2 is not an ancestor of c_2, since $bitAnc((pathid(c_2))[pos(a_2)]$ (*i.e.*, in the bit-vector labeled C(8,4)), which corresponds to c_2, is 0. It violates the condition (3) of Theorem 1.*

EXAMPLE 3. *With pairs of bit-vectors in Figure 5, we decide that a_2 and b_2 such that $pos(a_2) = 1$ and $pos(b_2) = 7$ have no A-D relationship, since $bitPath(1)[k] = 1, \exists k : pos(a_2) < k < pos(b_2)$.*

Note that if a_k is the k-th element which corresponds to a pathstring, and it has a descendant d, then $pos(a_k) < pos(d) < pos(a_{k+1})$. Thus, we can skip examining 1's in $bitPath(j)$ from $pos(a_k)$ to $pos(a_{k+1})$, if they do not satisfy other conditions. This gives us a performance merit while checking A-D relationship. We additionally use the level information of elements for identifying P-C relationship.

Algorithm 1: checkAD(a, b)

```
   Output: Boolean
1  if Curr_a.pid ≡ Curr_d.pid then
2  |  return false;
3  else if pos(a) < pos(d) then
4  |  if getValueFromBitAnc(Curr_a.pid, Curr_d.pos) then
   |  |  // Check any other node exists btw. them in the
   |  |     current ancestor's bitPath
5  |  |  if Next1_a[Curr_a.pid].pos < Curr_d.pos then return
   |  |  false;
6  |  |  else return true;
7  |  else return false ;
8  else
9  |  return false;
10 end
11 Function getValueFromBitAnc(pid, pos)
12 |  return a value from a bit-vector of bitAnc which represents
   |  pid in the position, pos;
13 end
```

Algorithm 1 describes a function which checks A-D relationship between two elements. In line 1, it checks whether two elements a and d have the same path id. If it is true, the elements are terminals of the different occurrences of a given pathstring. *e.g.*, a_2 and a_3 in Figure 1. Thus, they have no A-D relationship. However, if a and d have the same tag name, but different path ids, A-D relationship between them may exist. *e.g.*, two elements a_1 and a_2 in Figure 1 have the same tag name and also have A-D relationship. Note that we do not merge bit-vectors in Algorithm 1, so as to identify which bit-vector corresponds to the current nodes. In line 5, to examine the condition (4) of Theorem 1, we compare the next 1's position with the one of the current 1. To avoid moving a cursor in *bitAnc* from $pos(a_k)$ to $pos(a_{k+1})$ every time it checks the condition (4), we keep the position of the next 1 in advance by looking ahead (see Section 5.3). The `getValueFromBitAnc(pid, pos)` is a function for examining the condition (3) of Theorem 1.

5. TWIG QUERY PROCESSING

5.1 Algorithm Overview

Algorithm 2: Main procedure

```
1  condense a given twig Q to Q' if possible ;   // see section 5.4
2  select bit-vectors from index(es) and load them into each query
   node q ∈ Q' ;                                  // see section 5.2
3  while ¬end(root) do
4  |  q ← getNext(root);
5  |  if q ≠ root then
6  |  |  cleanstack(S_parent(q), Curr_q.pos);
7  |  if q ≡ root ∨ ¬empty(S_parent(q)) then
8  |  |  cleanstack(S_q, Curr_q.pos);
9  |  |  push Curr_q and Next1_q[Curr_q.pid] into stack S_q ;
   |  |  // see section 5.3
10 |  |  advance(q);
11 |  |  if isLeaf(q) then
12 |  |  |  showSolutionFromStacks(q)// see twigStack [1]
13 |  |  |  pop(S_q);
14 |  |  else advance(q, Curr_parent(q).pos);     // see algorithm 3
15 mergeAllPathSolutions();                       // see twigStack [1]
16 Function getNext(q)
17 |  if isLeaf(q) then return q;
18 |  foreach q_i ∈ children(q) do
19 |  |  n_i ← getNext(q_i);
20 |  |  if ¬exist(q_min) ∨ (Curr_n_i.pos < Curr_q_min.pos) then
   |  |  q_min ← n_i;
21 |  |  if ¬exist(q_max) ∨ (Curr_n_i.pos > Curr_q_max.pos) then
   |  |  q_max ← n_i;
22 |  |  if parent(n_i) ≠ q then return n_i;
23 |  while ¬checkAD(q, q_max) // see algorithm 1
24 |  do
25 |  |  advance(q);
26 |  if Curr_q.pos > Curr_q_min.pos then return q_min;
27 |  else return q
28 end
```

Our main algorithm for twig join is shown in Algorithm 2. The inputs vary according to the type of a selected index. Given a twig query Q, we first condense query nodes if possible. (This will be discussed in section 5.4.) This step reduces the number of query nodes to be processed. Second, we select bit-vectors for each query node from the index. Then, we iterate the procedure in line 3-14 until the root query node exhausts its input (*i.e.*, until all cursors in bit-vectors for *root* reach the end of bit-vector). The `getNext(q)` returns a query node n_i whose current position is the smallest among the current positions of query nodes that are descendants of the current element of a query node q. The `CleanStack(`S_q`, pos)` pops out nodes from the stack S_q, if they are not ancestors of a node referred to by *pos*. The two functions either read labels or call `checkAD(a,b)` (Algorithm 1) to determine the A-D relationship, depending on the type of index used.

The `advance()` is to move a cursor in bit-vector(s) assigned to each query node forward and set the current value of bit-vector(s) to *pos* the cursor points to. (This will be described in Section 5.3). The properties of the two hybrid indexes in this paper are summarized in Table 2.

5.2 Selecting Bit-vectors

The number of bit-vectors assigned to a query node vary according to the selected index type. With *bitTag*, only a single bit-vector per query node q is selected. With *bitPath*, multiple bit-vectors which satisfy a pathstring s for a query node q are merged into a single vector(Lemma 1.(2)), then it

Name	DescTag	bitTwig
Indexes used	$bitDesc$ & $bitTag$	$bitPath$ & $bitAnc$
Inputs per query node	single bit-vector & labels	multiple bit-vectors only
Type of bit-vector	$(\bigoplus_{i \in pids(q)} bitDesc(i)) \odot bitTag(a)$	$\bigcup_{i \in pids(s)}($ $bitPath(i), bitAnc(i))$
Optimization	$advance(k)$	$condense(q)$

Table 2: Summary of Two Hybrid Indexes

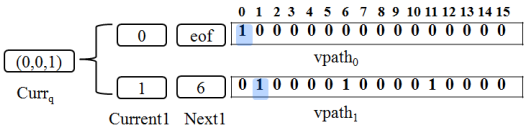


Figure 6: Current Values and Multiple Bit-vectors Per Query Node

associates with q. For a twig pattern with OR-predicate(in Figure 2(d)), we build a query node x instead of both C and D, then find bit-vectors which represent either of paths `//A/B/C` or `//A/B/D`. Next, we merge the bit-vectors into a single vector and assign it to x. With $DescTag$ index, we find all $bitDesc$ bit-vectors for pids corresponding to each query node q in Q, then merge them into a single vector for each q. Next, we AND this with $bitTag(a)$ for each query node a, according to Lemma 1.(4). $bitTag^+$ and $bitTwig$ allow multiple bit-vectors per query node, and selected bit-vectors in the indexes are not merged. Note that if any query node in Q does not have any corresponding bit-vector, no solution for Q exists in the XML document D. This property makes it possible to examine if twig solution for Q exists in D, without completing whole twig join.

5.3 Advancing Cursors

For the sake of twig joining, we augment our bitmap indexes with *cursors* so that the positions of 1's in a bit-vector can be checked individually by advancing a cursor associated with the bit-vector. The `advance()` moves a cursor forward to the next 1 in bit-vector(s) and fill $Curr_q$, *i.e.*, the current element information of q's input, with either of a bit position or a corresponding label. *i.e.*, $Curr_q$ retains a pair of (pos, $levelNum$) for $bitTwig$ and retains a label for other indexes. For a single bit-vector per query node, $Curr_q$ always retains pos a cursor indicates. For multiple bit-vectors per query node, only the minimum of the cursors' positions is selected as $Curr_q$ to guarantee that Algorithm 2 reads an element per query node at a time. Figure 6 illustrates multiple bit-vectors and their cursors on them associated with a query node `A` in a twig pattern (shown in Figure 2(a)). In the figure, a $Current1$ and a $Next1$ are associated with each bit-vector. The $Current1$ stores the position of the bit the cursor indicates and $Next1$ is set to the position of the next 1 in the bit-vector by looking ahead. Note that we avoid seeking the next 1 for checking A-D relationship in Algorithm 1 by keeping the $Next1$ in this way. Among $Current1$s, the minimum value is selected as $Curr_q$, with pid and $levelNum$ of the bit-vector.

When `advance()` is called, The $Next1$ of the bit-vector from which the current $Curr_q$ comes is copied to $Current1$ of the bit-vector. Next, the cursor in the bit-vector moves forward to the next 1, then the $Next1$ is newly set to the cursor's current position. *e.g.*, two bit-vectors' $current1$ indicate 0 and 1 in Figure 6. Thus, the minimum value, 0 is selected as $Curr_q$. After that, `advance()` moves the cursor of $bitPath(0)$, which corresponds to the current $Curr_r.pid$, forward and finally, it reaches the end of the bit-vector. At this time, 1 is chosen as $Curr_q.pos$.

With $bitTag^+$ index, We block off copying $current1$ to $Curr_q$ for irrelevant bit-vectors associated with q in runtime, by checking the current level value of inputs of $parent(q)$ or $ancestor(q)$. For example, we have a query node $ancestor(q)$ which associates with multiple bit-vectors for the input and let $Curr_{ancestor(q)}.level$ be the level of the current element of $ancestor(q)$. Here, we block off copying $Current1$ of the bit-vector which does not satisfy $level > Curr_{ancestor(q)}.level$. For P-C relationship, we ban copying $Current1$ of the bit-vector which does not satisfy $level = Curr_{parent(q)}.level+1$. As a result, we can skip reading elements of which levels do not satisfy the level condition. This improves I/O optimality of twig join by avoiding reading useless elements from the relational table.

5.4 Optimization

The CPU time of holistic twig join algorithm which allows multiple inputs for a single query node is $O(N \times |Q| \times |Input+Output|)$ for a twig query Q [2], where N is the total number of useful inputs for the twig query Q. Since $|Output|$ in all twig join algorithms should be always same, the only option for optimization is to reduce $|Q|$ and $|INPUT|$. We introduce our methods to reduce $|Q|$ and $|INPUT|$ in this section.

Condensing Query Nodes

With a bitmap index built over path domain, *i.e.bitPath* or *bitTwig*, query nodes which are neither leaf nodes nor branching nodes and which are not projected to be appeared in twig solution can be ignored during query processing. This reduces both $|Q|$ and $|INPUT|$. *e.g.*, in Figure 2(c), if a query node B is not projected, we do not read all b elements during query processing. Since, with our indexes built on path domain, bit-vectors associated with C represent $\exists c$ elements such that c corresponds to a linear path `//A/B/C`, rather than $\forall c \in C$, according to Lemma 1.(2). Thus, we can select useful c nodes without reading any b.

Bit-Vector Compression and Cursor Movement

The number of bit-vectors in a bitmap index is as same as the cardinality of a column the index is built over. Since a high-cardinality column makes index size large, compression techniques are useful to reduce the index size. A common technique for compressing bit-vectors is *run-length encoding*, where *run* is a consecutive sequence of the same bits. We use a hybrid run-length encoding [13] that groups bits into compressed words and uncompressed words. The major benefit of this approach is that bitwise operations are performed very efficiently, benefiting from the fact that bitwise operations are actually performed on word units. Figure 7 shows how a bit-vector of 8,000 bits is compressed into four 32-bit words by the compression scheme. The first bit of each word decides if the word is compressed. If the first bit is 1, the word is compressed with a sequence of bit values that the second bit(*i.e.*, a fill bit) stores. *e.g.*, if the fill bit is 0, then the word is filled with a sequence of 0's. In the figure, the first and the third word is uncompressed. The second word is compressed and 256*31 0's fill the word. The fourth word keeps the rest bits, which stores the last few bits that

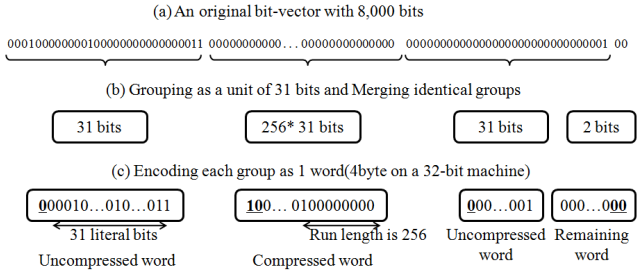


Figure 7: Compressing A Bit-vector on a 32-bit Machine

could not construct a single word by themselves. Another word with the value 2, not shown, is required to store the number of useful bits in the rest word.

Furthermore, to enable a cursor to move on a compressed bit-vector without decompressing the bit-vector, we implemented the cursor as a composite variable, which consists of a logical position and 3 physical position values.

C.position : the logical(integer) position of the current bit C is located at,
C.word : the current word C is located at,
C.bit : the current bit within *C.word* C is located at,
C.rest : The current bit within the rest word, if C is located at the word

We use two functions for moving a cursor on a compressed bit-vector (*i.e.*, `getPosOfNext` and `getValueFromBitAnc`). With the cursor C, we can skip examining a consecutive sequence of 0's by examining just one compressed word whose fill bit is 0. Thus, the cost of finding 1's in a bit-vector is remarkably reduced as the cursor meets the compressed words.

Example 4. *Let a cursor C indicate that the 31-th bit of the first word of a bit-vector at the bottom of Fig.7, then the current status of C is {31, 0, 31, 0}. When we find the position of the next 1, it begins examining the first bit of the second word, which is compressed. Since the word is filled with 0, we skip counting the position value bit-by-bit by simply increasing C by $run\_length * w$. C will be {7997, 1, 31, 0} right after reading the second word. Thus, the position of the next 1 will be 7,998(=31+256*31+31).*

In Algorithm 1, we also check the bit at a given position *pos* for identifying A-D relationship between two elements with a function (`getValueFromBitAnc`). For the function, we maintain two cursors in each bit-vector from *bitAnc*. We separate two-way cursor moves into two one-way cursor moves since a cursor's bidirectional moves may affect the distance C moves, *i.e.*, C moves forward longer distance if C went back once. In the function, if a given position *pos* is less than the forward cursor C, it moves another cursor B backward. Also, if the *pos* is behind the position of B, then it copies the position of C to B to set a starting line for B. By adding the backward cursor B, it prevents the cursor from strolling over a bit-vector by occasional backward moves. When the cursor C moves forward, it first computes the distance C should move and reduces the remaining distance whenever C advances. If the remaining distance is less than the number of bits the visited word represents, the word contains the bit to be checked. The cost for computing the remaining distance is also cheap when C meets compressed words, since the number of bits to be examined is reduced by the compression. In our algorithm, the two function calls never mingle with on a single bit-vector. The cursors in most of indexes are triggered by only `getPosOfnext` and only the cursors in *bitAnc* are triggered by `getValueFromBitAnc` calls.

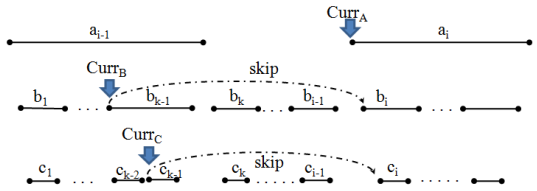


Figure 8: Skipping to Read Useless Nodes

Example 5. *Let a cursor C be at the 16-th bit of the first word in Figure 7,* i.e., *C={16, 0, 16, 0}. To get the value of 3,000-th bit, it first computes the remaining distance as 2,884(=3,000-16). Next, since the current word is not compressed and the distance is not smaller than the number of bits the word represents, The C moves forward by a word. Then, C.position is 31 and the remaining distance is 2,869(=3,000-31). The second word is compressed and the distance $< run\_length * 31$, thus it returns the fill bit of the word, 0 as a result.*

Skipping Obsolete Input Nodes

We skip reading many useless labels with a function `advance(k)`. The idea is illustrated in Figure 8. Assume that there is a linear path `//A//B//C`, and intervals of elements in each query node input are like those shown in Figure 8. Let the cursors for each input indicate a_i, b_{k-1}, and c_{k-1}. With no skipping, elements in the input for query node B will be read from b_{k-1} to b_i. Here, b_i which has A-D relationship with a_i satisfies a condition, $startPos(a_i) < startPos(b_i) \wedge endPos(b_i) < endPos(a_i)$. Thus, we can skip reading labels in the input of B from b_k to b_{i-1} by jumping the cursor in our index to b_i which satisfies $pos(a_i) < pos(b_i)$. Also, this action is performed recursively, so we also skip reading labels for the query node C from c_k to c_{i-1}.

Algorithm 3: advance(q, k) for a single bit-vector *bv*

```
1  pos ← C.position ;                          // A value for position
2  if k is not given then
3  |  pos← getPosOfNext1(bv);
4  else
5  |  repeat
6  |  |  pos← getPosOfNext1(bv);
7  |  until pos≤ k ;
8  end
9  if pos≠EOF then
10 |  read the record that pos value indicates and load a label;
11 end
```

In Algorithm 3, a input k is $pos(ancestor(q))$, *i.e.*, the current position of an ancestor of a query node q. Two elements which have A-D relationship satisfy a condition $pos(D) > pos(A)$. Therefore, we iteratively advance a descendant's cursor until it satisfies the condition (in line 5-7). Then, it loads a label from the record the cursor indicates. Note that *advance(k)* in Algorithm 3 is a version for a single input. If multiple inputs per query node are allowed, the function advances cursors of all bit-vectors until k and it sets the current position, by the rule described in Section 5.3.

5.5 Analysis of bitTwig

The size of input data when using one of basic indexes is $O(\sum_i^n(|bv_i|)+|tuples|)$, where n is the total number of useful bit-vectors, $|bv_i|$ is the size of a useful bit-vector bv_i and $|tuples|$ is the number of useful tuples in a node table(shown in Figure 3). Meanwhile, the size of input data when using *bitTwig* is at most $O(2 \times \sum_i^n |bv_i|)$ since it requires a pair of indexes only. The CPU time of our algorithm when using a basic index is simply the sum of the CPU time of twig joining and the cost of cursor moves on useful bit-vectors. The CPU time of *bitTwig* can be computed by summing the cost of total calls of each sub function in our algorithm. Algorithm 2 contains 3 sub functions reading inputs, *i.e.*, `advance()`, `getNext()` and `cleanStack()`. Total calls of `advance()` in Algorithm 2 read all 1's in useful bit-vectors. Moreover, in each `advance()` call, the minimum *pos* among bit-vectors' *current*1s is selected. Thus, the cost of total `advance()` calls in *bitTwig* is $O(n \times |Q| \times \sum_i^n |bitPath(i)|)$. Note that `advance()` moves a cursor in a $bitPath(i)$ forward, and `checkAD()` moves a cursor of $bitAnc(i)$ to a specific position. Thus, `checkAD()` does not affect the cost of `advance()` calls.

The `checkAD()` function is called in `cleanStack()` and in `getNext()` to identify A-D relationship between two elements. Since `getNext()` calls `checkAD()` with the position of ancestor's cursor, which is always advancing forward by `advance()`, it makes the cursor of $bitAnc(i)$ move forward only. Thus, the cost of total calls of `getNext()` is $O(n \times |Q| \times \sum_i^n |bitAnc(i)|)$. In contrast, `cleanStack()` involves the cursor to move backward. While `cleanStack(`a_i`,` d_j`)`, items in a stack S_A are compared with the current cursor position of $bitAnc(j)$ for checking A-D relationship according to the condition (3) of Theorem 1. Thus, items in S_A are visited from top to bottom and this involves that the cursor in $bitAnc(j)$ moves backward. However, even in this case, the cursor never goes back over the same 1's repeatedly. Because, if the topmost stack item has A-D relationship with d_j, other items in S_A are not compared with d_j and they do not trigger the backward cursor moves. On the contrary, if the topmost stack item has no A-D relationship, the item will be popped out from S_A, and the cursor does not move toward the position the popped item indicates any more. Thus, with our separated two-way cursors described in Section 5.4, the cost of total calls of `cleanStack()` is $O(n \times |Q| \times \sum_i^n |bitAnc(i)|)$. Finally, the total CPU time of *bitTwig* is $O(n \times |Q| \times (\sum_i^n(|bitPath(i)| + |bitAnc(i)|) + |Output|))$ when added in the output size, meaning that *bitTwig* is linear in the sum of the size of bit-vector pairs. Note that the size of a pair of the bitmap indexes is even less than the size of labels in most cases. (see Table 4).

6. EXPERIMENTS

6.1 Experimental Setup

We implemented our algorithms as a single threaded executable in GNU C++. All experiments were performed on a system having a Core 2 Duo 2GHz processor and a 5,400 RPM disk that ran Linux 2.6. We used the FastBit library [13] and augmented it with our cursor mechanism. We also used Xerces library to parse XML documents. The node table in Fig 3(b) and indexes were built on a file system. To check how indexes and different input data structure affected the performance, we also compared our algorithms with *twigStack* on tag-based streaming model. For more fairness, the outputs from all algorithms are were to be equal in size.

We used a synthetic XML and three real XML datasets for experiments. All datasets are first labeled and stored as rows in a table (shown in Fig 3(b)). XMark is the synthetic dataset that describes data dealt in a virtual auction site. Its structure is recursive, but not too much. DBLP and SWISSPROT(shortly, SPROT) are real-world datasets, which have rather shallow structure. TREEBANK(shortly, TBANK) is a real-world dataset, which has a deep recursive structure. A statistics of our datasets is shown in Table 3. Note that the number of distinct tags in all the datasets are not different much, but the number of distinct paths varies.(see TBANK in Table 3.) We selected various twig

	XMark	DBLP	TBANK	SPROT
Size(MBytes)	113	486	86	112
# of elements	1,666,315	11,692,273	2,437,666	2,977,031
# of attributes	381,878	2,305,980	1	2,189,859
# of tags	77	41	251	99
# of (tag, level)	119	47	2,237	100
# of paths	548	170	338,749	264
Max/Avg depth	12/5	6/2.9	36/7.8	5/3.5

Table 3: XML Datasets

queries which included (1) twigs with A-D axes only (2) twigs with P-C only (3) twigs which had one branching node (4) twigs which were mixed up with A-D and P-C axes.

Our indexes were built on the node table in Fig 3(b). We used a stack to generate reverse path strings(shortly *rps*) and label each node. Our index construction algorithm pushes an element into the stack when a *startElement* event of the element occurs, and pops out the element when an *endElement* occurs. In `push()`, we label *startPos* and generate *rps* by reading items in the stack from top to bottom and assign unique id(*i.e.*, pid) to the *rps*. In `pop()`, we label *endPos* to the element. Each bit-vector in our indexes stands for one distinct value in a single or dual columns. Thus, building the indexes are similar to the way to build a bitmap index in RDBMS. However, *bitAnc* and *bitDesc* indexes are different since each bit-vector in the indexes represents a set of values, and the sets are not disjoint. To build *bitAnc*, whenever `push()` is called, we copy the *pid* of an upper stack item into a lower items from top to bottom simultaneously. Thus, when parsing ends, all elements have a set of pids for their descendants, *e.g.*, a root a_1 in Fig 1 has all *pid*s in the XML, pid 0∼11. Then, we construct *bitAnc* by setting the i-th bit to 1 in each corresponding bit-vector for each in a set of path ids that the i-th XML element corresponds to. To build *bitDesc*, whenever `pop()` is called, we copy the *pid* of a lower stack item to an upper item from bottom to top simultaneously. Then, each element has a set of pids for its ancestors. *e.g.*, b_1 in Fig 1 has a set of pids, 0, 1, 2. We construct *bitDesc* with the pid sets.

6.2 Experimental Results

Statistics of Indexes We compared sizes of our indexes in Table 4. Compared with the interval-based labels, our indexes were comparably small. Also, even in a case of *bitTwig*, the sum of *bitPath* and *bitAnc* was smaller than the labeled values, except for TBANK dataset. Our observation

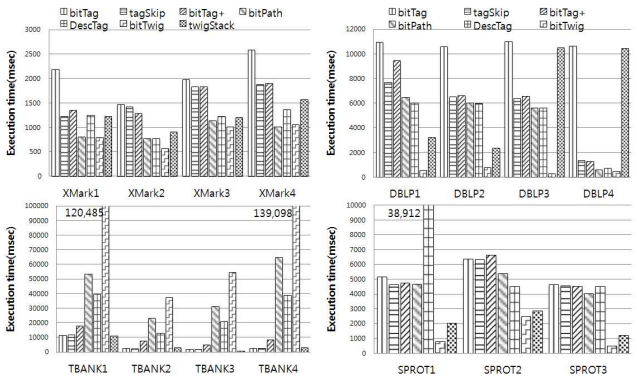


Figure 9: Query Execution Time

was that unless XML structure is severely deep and recursive, the size of bit-vectors built over path domain is small. Another observation was that index building time is mainly caused by column cardinality, rather than the number of tuples. *e.g.*, building indexes over path domain in TBANK took the longest time(in Table 5). On the contrary, building indexes over DBLP, which has about 14 million rows in a relation, took rather a little time.

(MBytes)	XMark	DBLP	TREEBANK	SWISSPROT
interval	23.44	160.20	27.90	59.13
bitTag	5.32	19.67	6.85	14.40
bitPath	6.18	20.00	23.53	19.03
bitAnc	6.74	20.08	30.84	26.50
bitDesc	6.00	18.34	24.18	18.92
$bitTag^+$	6.05	20.67	14.33	14.62

Table 4: Size of Encoded Values

(Seconds)	XMark	DBLP	TREEBANK	SWISSPROT
bitTag	17.36	98.51	38.61	52.34
bitPath	20.81	77.63	7,408.09	35.71
bitAnc	34.45	169.10	8,054.15	69.11
bitDesc	21.73	86.35	7,528.23	38.28
$bitTag^+$	29.00	180.95	206.75	88.30

Table 5: Index Build Time

Performance Measure We measured our query processing algorithm with various indexes with two criteria: (1) Execution time for answering twig queries (shown in Fig 9) and (2) The size of data read during twig join.(shown in Fig 10). In Fig 9, five indexes and *twigStack* were compared on four datasets. In the figure, *tagSkip* stands for the use of *bitTag* with *advance*(k) for skipping useless labels. The other indexes were optimized by either of *advance*(k) and *condense*(q). We assumed the bitmap indexes were not loaded into memory before query processing, and buffer size was set to 0 in our experiments. Thus, index loading time was always included in the execution time.

Performance Analysis In terms of the size of data read during query processing(in Figure 10), we saw that in all cases, *bitTwig* read the smallest size of data. This is because *bitTwig* reads only bit-vectors from two indexes for

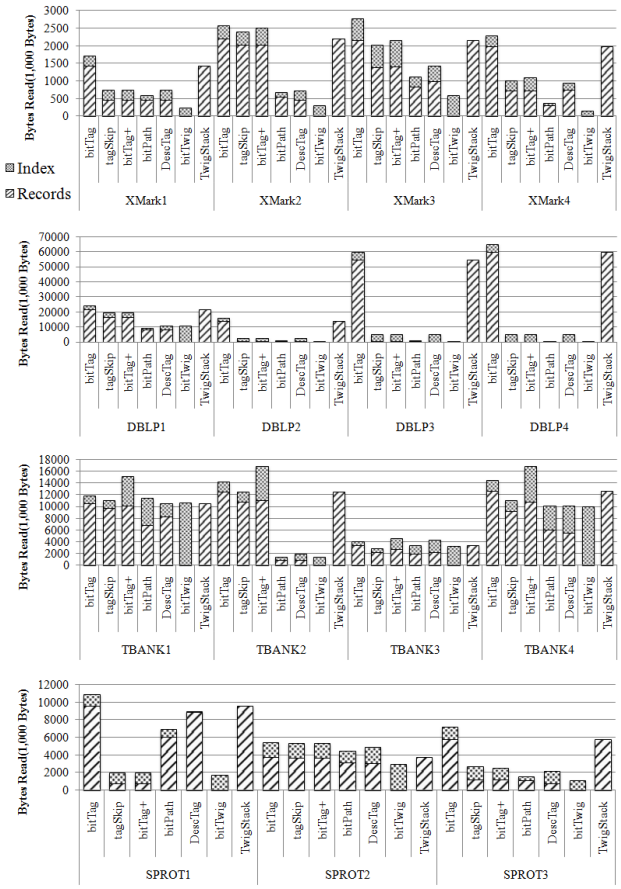


Figure 10: Bytes Read During Query Evaluation

answering twig queries. We also saw that in general, indexes built over path domain read less size of records. Usually, indexes built over tag domain read the most size of records, but *advance*(k) significantly reduced the size. The $bitTag^+$ index built over (`tag`, `level`) domain had a trade-off between the size of index read and data read, especially on TBANK dataset. The $bitTag^+$ read less records, however it required more bit-vectors for query processing. The $bitTag^+$ also did not outperform other strategies in terms of execution time in our experiments. The *DescTag*, which merges a path-based bit-vector and a tag-based bit-vector for each query node, was comparable to other strategies which used path-based indexes in some cases. However, when the structure was extremely flat and the number of leaf elements was large, merging bit-vectors affected the performance, *e.g.*, SPROT1 in Figure 9. In terms of execution time, *bitTwig* outperformed other indexes in most cases. Especially, it was best suitable for answering twigs towards XML data whose structures were not deep and not recursive, *e.g.*, DBLP and SPROT. However, it showed unendurable execution time for TBANK (in Figure9), since it needed to hold many bit-vectors per query node and managed cursor moves on them. In a case of recursive-structured XML, *bitTag* with *advance*(k) showed the best performance in our experiments. $bitTag^+$ was not the best in any case. However, it ran fast on TBANK dataset, next to *tagSkip* and *bitTag*. *DescTag* showed the execution time similar to the one of *bitPath*, but it rather showed better performance on

TBANK. We also observed that in cases of *bitPath* and *DescTag* on TBANK, most of execution time was spent to find and load bit-vectors. Finding corresponding bit-vectors with a given path which includes A-D axes involves a subsequence matching problem, which is known as difficult to be supported efficiently. Similar to the concept of pre-computed join, pre-merged bit-vectors may be useful for the path selections. Once, we merge bit-vectors into a single bit-vector for a given pathstring, we can find pathstrings with an exact matching query next time. In addition, the number of bit-vectors to be loaded decreases by one.

7. RELATED WORK

The twigStack is a multi-way, sort-merge join algorithm that avoids generating obsolete path solutions during twig pattern matching [1]. The I/O optimality is proven for twig patterns that have no /-axis. Much research has been focused on guaranteeing I/O optimality for a certain class of XPath [9, 2]. The iTwigJoin work discussed optimality for twig patterns with A-D or P-C only or one branch-node only by partitioning input data as a unit of tag+level, and prefix path-based [2]. Variants of a B^+-treeindex have been suggested to skip unnecessary input data such that twig queries can be processed more efficiently [1, 6]. TJFast adopted a different approach to reducing the size of input data [10]. It encoded root-to-leaf path of each leaf node in extended Dewey order, so internal nodes could be derived from the encoded values of leaf nodes.

Various XML storage schemes have been suggested for relational database systems [8]. A labeling scheme or a path materialization is among the common techniques adopted by an RDBMS for XML query processing. It has been shown that the conventional index structures of an RDBMS can be tailored to index XML data as well [12]. Recently, Grust *et al.* show that node approach can give better performance than native XML storage system with partitioned B^+-tree [4].

Bitmap index is known to be effective for indexing a column from a small domain and for performing logical operations [11]. For columns with high cardinality, bitmap indexes can be compressed to reduce the sizes, or a hybrid encoding scheme (such as WAH [13]) can be adopted so that bitwise operations can be performed efficiently using word units. BitCube [14] adopts bitmap indexes for XML query processing. Similar to the data cubes for OLAP, it consists of 3-dimensional bitmaps, where the dimensions represent documents, paths, and values. However, it only supports querying a linear path with value predicates and its size can be huge since its structure is hard to be well compressed.

8. CONCLUSIONS

We propose various bitmap indexes for querying XML data stored in relational tables. We support twig query processing by an RDBMS with cursor-enabled bitmap indexes. Our cursor mechanism and query processing algorithms work on compressed bit-vectors without uncompressing them. The performances of proposed algorithms are evaluated in our experiments. The *bitTwig* outperforms other strategies for shallow XML documents by computing twig solution with only bit-vectors. For XML documents with a deep and recursive structure, tag-based bitmap indexes with $advance(k)$ show the best performance.

9. REFERENCES

[1] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the ACM SIGMOD Conference*, pages 310–321, 2002.

[2] T. Chen, J. Lu, and T.W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proceedings of the ACM SIGMOD Conference*, pages 455–466, 2005.

[3] G. Gou and R. Chirkova. Efficiently querying large xml data repositories: a survey. *IEEE TKDE*, 19(10):1381–1403, 2007.

[4] T. Grust and et al. Why off-the-shelf RDBMSs are better at XPath than you might expect. In *Proceedings of the ACM SIGMOD Conference*, pages 949–958, 2007.

[5] P. J. Harding, Q. Li, and B. Moon. XISS/R: XML indexing and storage system using RDBMS. In *Proceedings of the 29th VLDB Conference*, pages 1073–1076, 2003.

[6] H. Jiang and et al. Holistic twig joins on indexed XML documents. In *Proceedings of the 29th VLDB Conference*, pages 273–284, 2003.

[7] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th VLDB Conference*, pages 361–370, 2001.

[8] H. Lu and et al. What makes the differences: benchmarking XML database implementations. *ACM Transactions on Internet Technology*, 5(1):154–194, 2005.

[9] J. Lu, T. Chen, and T.W. Ling. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In *Proceedings of the 13th ACM CIKM Conference*, pages 533–542, 2004.

[10] J. Lu and et al. From region encoding to extended dewey: on efficient processing of XML twig pattern matching. In *Proceedings of the 31st VLDB Conference*, pages 193–204, 2005.

[11] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of the ACM SIGMOD Conference*, pages 38–49, 1997.

[12] S. Pal and et al. Indexing XML data stored in a relational database. In *Proceedings of the 30th VLDB Conference*, pages 1146–1157, 2004.

[13] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *Proceedings of the 30th VLDB Conference*, pages 24–35, 2004.

[14] J.P. Yoon, V. Raghavan, V. Chakilam, and L. Kerschberg. BitCube: a three-dimensional Bitmap indexing for XML documents. *Journal of Intelligent Information Systems*, 17(2):241–254, 2001.

[15] M. Yoshikawa and et al. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1):110–141, 2001.

[16] C. Zhang and et al. On supporting containment queries in relational database management systems. In *Proceedings of the ACM SIGMOD Conference*, pages 425–436, 2001.