Holistic Join for Generalized Tree Patterns

Prakash Ramanan
Department of Computer Science
Wichita State University
Wichita, KS 67260–0083
ramanan@cs.wichita.edu

Abstract

We consider the problem of evaluating an XQuery query $Q$ (involving only child and descendant axes) on an XML document $D$. $D$ is stored on a disk and is read from there, in document order. Chen et al. [From Tree Patterns to Generalized Tree Patterns, Intl. Conf. on Very Large Databases, 2003, pp. 237-248] presented an algorithm to convert $Q$ (from a large fragment of XQuery) into a Generalized Tree Pattern $GTP(Q)$, and a set $J(Q)$ of value join conditions on its vertices. Evaluating $Q$ on $D$ reduces to finding the matches for $GTP(Q)$ in $D$. We present an efficient algorithm for finding these matches. Excluding the computation of the value joins $J(Q)$, our algorithm performs two linear passes over the data, and runs in $O(d|Q|)$ memory space, where $d$ denotes the depth of $D$; runtime and disk I/O are $O(|Q||D|)$. If separate input streams of document nodes for the individual vertices in $GTP(Q)$ are available, our runtime and disk I/O are linear in the input size; this runtime and disk I/O are trivially optimal.

Keywords. XML, XPath, XQuery, generalized tree patterns, query evaluation

1 Introduction

We consider the problem of evaluating an XQuery query [5] $Q$, involving only child and descendant axes, on an XML document $D$. We assume that $D$ is too large to be stored in the memory. It is either stored on a disk and is read from there, or it is input as a steam; in either case, it is input in document order. Towards the end, we present modifications to our algorithm to exploit some XML storage strategies that provide separate input streams of document nodes for the individual query vertices [2, 19, 22].

A large class of XPath queries [4] can be modeled as Tree Patterns [3, 24]. Chen et al. [8] presented an algorithm to convert an entire XQuery query $Q$ into a single Generalized Tree Pattern
$GTP(Q)$, and a set $J(Q)$ of value join conditions on its vertices. Their algorithm is applicable to a large fragment of XQuery (FLWOR expressions), possibly involving value joins, quantifiers, if-then-else, grouping, aggregation and nesting. $GTP(Q)$ summarizes the structural and atomic value constraints in $Q$; $J(Q)$ represents the value join constraints. Let $output(Q, D)$ denote the result (XML document) of evaluating $Q$ on $D$. Computing $output(Q, D)$ reduces to finding those matches for $GTP(Q)$ in $D$ that satisfy $J(Q)$. Chen et al. presented an algorithm to find these matches, but their algorithm involves several structural joins, requiring multiple passes over the data. This algorithm has been implemented in the TIMBER native XML database system [13].

To minimize confusion, we will use the terms vertices and arcs while referring to the components of $GTP(Q)$; nodes and edges refer to the corresponding components of $D$. The distinguished vertices in $GTP(Q)$ are those vertices whose matches in $D$ determine $output(Q, D)$ (see Section 2); for instance, an XPath query without union has exactly one distinguished vertex, corresponding to the node test in the last location step. Let $dv(Q)$ denote the ordered list of distinguished vertices of $GTP(Q)$; the order is determined by XQuery semantics. Let $bindings(dv(Q))$ denote the list of distinct node tuples of matches for $dv(Q)$ in $D$. Each such node tuple represents one or more matches for $GTP(Q)$ in $D$ (satisfying $J(Q)$) that agree on the matches for each of the distinguished vertices, but might differ on the matches for the other vertices. $Output(Q, D)$ can be easily obtained from $bindings(dv(Q))$.

We present a more efficient algorithm for finding $bindings(dv(Q))$. Our algorithm performs only two linear passes over the data, excluding the pass(es) required to compute the value joins. It takes $GTP(Q)$, $J(Q)$ and $D$ as input, and consists of four steps:

1. Pass 1 (Bottom-Up Pass): Process the document $D$ in bottom-up order (while reading it in document order). For each document node $e$, determine which structural constraints (subtrees) in $GTP(Q)$ are satisfied by the subtree (i.e., element) rooted at $e$. The nodes of $D$, along with this additional information, are written to the disk, in document $D'$, in postorder. Also, appropriate (node, string-value) pairs that are eligible to participate in the computation of
1. (Bottom-Up Pass): Process $D$ in bottom-up order. Construct document $D'$ and write it to disk, in postorder. Also, write appropriate (node, string-value) pairs for computing $J(Q)$, to the disk.

2. Perform the value joins $J(Q)$ on the (node, string-value) pairs from Step 1.


4. Obtain $\text{bindings}(dv(Q))$ from $D''$.

Figure 1: Four Steps in Obtaining $\text{Bindings}(dv(Q))$

---

A concise pseudocode is given in Figure 1. The output of Step 3 is $D''$, in right-to-left preorder. For each node in $D''$, there is information as to which distinguished vertices bind to it (over different embeddings). $D''$ also contains information as to which bindings of different vertices to different nodes can co-occur (in a single embedding). $D''$ preserves the ancestor-descendant relationship in $D$.

Almost every node in $D''$ contributes to $\text{bindings}(dv(Q))$ and is essential. We are guaranteed that $|D''| \leq |\text{bindings}(dv(Q))|$; $D''$ could be much smaller than $\text{bindings}(dv(Q))$. $D''$ is the most compact representation of $\text{bindings}(dv(Q))$ possible: For each node, it uses $O(1)$ space for each vertex in $dv(Q)$ that binds to it; this is irrespective of how many node tuples of bindings it might appear in. $\text{Bindings}(dv(Q))$, in document order, can be computed from $D''$ in $O(|\text{bindings}(dv(Q))|)$ time. If $D''$ fits in memory, this can be done in memory; else one more pass is needed (see Section
As a special case, let Q be an XPath or an XQuery query with only one distinguished vertex (ex. Q_2 in Section 2). In this case, D'' = bindings(dv(Q)). Our algorithm can be easily modified (Section 6) to output D'' in reverse document order (instead of the usual right-to-left preorder). If D'' is small, it can be stored in memory, reversed, and output in document order. Else, we need a third pass: D'' is written to the disk, read back in reverse, and output in document order.

**Resource Requirements:** Excluding Step 2 (value joins), our algorithm for computing D'' performs two linear passes over the data, and runs in O(d|Q|) memory space, where d denotes the depth of D; runtime and disk I/O are O(|Q||D|). If the vertices in GTP(Q) have low selectivity, the resources can be further reduced (by using linked lists in place of arrays). Some XML storage strategies [2, 19, 22] provide, for each vertex in GTP(Q), a stream of document nodes that are possible matches for that vertex. In this case our runtime and disk I/O are linear in the *input size*: sum of sizes of the streams returned by the strategy; this runtime and disk I/O are trivially optimal.

When Q involves value joins, if the structural constraints in Q are very selective, Step 1 would find only few elements eligible to participate in the joins; then Step 2 can be performed in memory.

Now, consider the computation of bindings(dv(Q)) from D''. If D'' fits in memory, this can be done in-memory, in O(|D''|) space and O(|bindings(dv(Q))|) time. Else, it takes one more pass, with O(d|Q|) memory space; runtime and disk I/O are linear in |bindings(dv(Q))|. This runtime and disk I/O are trivially optimal.

**Related Work:** First, consider XPath queries. Gottlob et al. [11] and Ramanan [24] presented in-memory algorithms for evaluating XPath queries that run in O(|D|) memory space and O(|Q||D|) time. Ramanan’s algorithm [24] consists of a bottom-up simulation, followed by a top-down simulation, of Q by D. Koch [17] presented an automata-based algorithm for evaluating XPath queries that makes two linear passes over D. Their automata need to be stored in the memory. In the worst case, the size of the automata could be exponential in |Q|. Their first pass (bottom-up) requires that
the input XML document be stored as a binary tree, and that its nodes be available in bottom-up order; additional passes are required to convert an XML document into such a binary tree, before their algorithm can be applied. Also, their algorithm is not applicable to XQuery queries which (unlike XPath queries) have multiple distinguished vertices: We need to obtain node tuples of bindings for the vertices in $dv(Q)$. It is not clear how to determine which bindings of different query vertices to different document nodes can co-occur (in a single embedding), when there are dependencies between the vertices in $dv(Q)$ (ex. FOR $y$ IN $x/...$).

Now, consider XQuery queries. Several in-memory algorithms for evaluating XQuery queries have been presented in the literature [23, 25, 10]. Marian and Simeon [21] pointed out that these algorithms can only handle small documents. They presented an algorithm to project a document $D$, so that only parts that are likely to contain the result for $Q$ are loaded into memory; then, an in-memory algorithm is used to evaluate $Q$. This increases the size of documents that can be handled. Their pruning of nodes is based only on the path from the document root to the node. It is oblivious to whether the subtree rooted at the node is a match for some subtree in $GTP(Q)$. In contrast, our Steps 1 and 3 prune nodes based first on the subtree rooted at the node, and then the path from the root (which now contains lot more information, from Step 1), respectively; our $D''$ contains only nodes that are absolutely essential to compute the output. Marian and Simeon [21] experimentally showed that their projection reduces the document size by a factor of 20, on the average. Our Steps 1 and 3 would reduce the document size by a much larger factor; hence our expectation that $D''$ would fit in memory, and that $bindings(dv(Q))$ can be computed in-memory, for many $(Q, D)$.

Ludascher et al. [20] presented an algorithm for evaluating $Q$ on an acyclic document $D$: No element in $D$ contains a subelement with the same label. Our algorithm is more general, as it can handle cyclic documents, bindings of the same query vertex to a document node and its descendants, as well as bindings of multiple vertices to the same node (over different embeddings).

Various algorithms for structural join have been described in the literature [1, 28, 30]. These algorithms can be used to determine $bindings(Q)$ for a twig $Q$; but doing so would require multiple
passes over $D$; also, intermediate result sizes could well exceed $|\text{bindings}(Q)|$. Chen et al. [8] presented such a structural join based algorithm for GTPs. The holistic twig join algorithm [6, 15, 14, 18, 7, 19] finds $\text{bindings}(Q)$, while avoiding large intermediate results. \textit{It is our only direct competitor.} In Section 3, we first compare our algorithm with theirs, for twigs, and then consider problems in extending their algorithm to GTPs.

Several one-pass algorithms have been proposed for evaluating XPath and XQuery queries on streaming XML documents. We would like to point out that \textit{all} such one-pass algorithms must use $O(|D|)$ memory space in the worst case. For example, for the query $/b[c]/f$, all $f$ nodes in the stream must be stored, because the root $b$ node might or might not have a $c$ child later in the stream. For the query $//b[c]$, the order in which the $b$ nodes are found to belong to the output (i.e., have a $c$ child later in the stream) might not match the document order; all such $b$ nodes need to be stored until they can be output in document order.

Since our algorithm uses two passes, it is not really a “stream processing algorithm”. But we do use a SAX parser to generate a stream of SAX events [26], as $D$ is read from the input (disk or stream) in document order. Our first pass operates on these SAX events.

**Outline of our Paper:** In Section 2, we describe GTPs. In Section 3, we compare our join algorithm for GTPs with the twig join algorithms of [6, 15, 14, 18, 7, 19]. In Section 4, we describe the SAX events pertaining to an XML document. Then, we first consider queries that do not involve value joins. The first pass of our algorithm is presented in Sections 5. The second pass is presented in Section 6 for XPath queries, and in Sections 7 and 8 for XQuery queries. In Sections 9, we describe the modifications needed in our algorithm to handle value joins. In Section 10, we explain how to obtain $\text{bindings}(dv(Q))$ from $D''$. In Section 11, we present modifications to our algorithm to handle separate input streams of nodes for the individual query vertices [2, 19, 22]. Section 12 concludes.
2 Generalized Tree Patterns

Chen et al. [8] presented an algorithm to convert an entire XQuery query $Q$ into a single Generalized Tree Pattern $GTP(Q)$, and a set $J(Q)$ of value join conditions on its vertices. Their algorithm is applicable to a large fragment of XQuery, possibly involving value joins, quantifiers, if-then-else, grouping, aggregation and nesting.

Before giving a formal description of GTPs, we first introduce them through a few examples. Our example queries $Q_1$, $Q_2$ and $Q_3$ below are extensions of the examples in [8]. The corresponding GTPs are shown in Figure 2. All these queries pertain to the auction.xml XMark document [27].

First consider the simple query $Q_1$ (Figure 2a):

```
FOR $p$ IN //people/person, $l$ IN $p$/profile
  WHERE $p$/address/state $\neq$ MI AND $l$/age > 25
  RETURN <result> {$p$/watches/watch[make $\neq$ "Seiko"]} {$l$/interest}</result>
```

For each binding of ($p$, $l$) in $D$, the query should return exactly one result element, irrespective of the number of bindings (including 0 bindings) for watch and interest; all bindings for watch and for interest (if any) should be listed inside this result element.

In our figures, thin and thick lines denote child axes (c-arcs) and descendant axes (d-
arcs), respectively. As in Chen et al. [8], solid and dashed arcs (whether thin or thick) correspond to \textit{mandatory} and \textit{optional} relationships, respectively. In general, the root vertex of $GTP(Q)$ has the tag $/$. For $Q_1$, the XPath expressions for $\$p$ and $\$l$ (in the \texttt{FOR} clause) correspond to the paths 1–2–3 and 3–9, respectively, in $GTP(Q_1)$. The two conditions in the \texttt{WHERE} clause correspond to the paths 3–4–5 and 9–10. The two output fragments in the \texttt{RETURN} clause correspond to 3–6–7–8 and 9–11. The \textit{mandatory} c-arc from $\$p$ to $\$l$ indicates that any document node $\$p$ binds to must have a \textit{profile} subelement (which in turn must have an \textit{age} subelement with value > 25). The \textit{optional} arc from $\$l$ to \textit{interest} indicates that a document node that $\$l$ binds to need not have an \textit{interest} subelement; but we still need to find all the \textit{interest} subelements (if any) of such a node, so that they can be output inside the corresponding \texttt{<result>} element.

An embedding $\beta$ of $GTP(Q_1)$ in $D$ is a \textit{partial} mapping from the vertices of $GTP(Q_1)$ to the nodes of $D$, such that:

1. $\beta(\text{root}(GTP(Q)))$ is defined.

2. If $\beta(u)$ is defined, then $\beta(v)$ is defined for all descendants $v$ of $u$ for which the path from $u$ to $v$ consists only of solid arcs.

3. If $\beta(v)$ is defined, then $\beta(u)$ is defined for all ancestors $u$ of $v$.

4. If $\beta(v)$ is defined:
   
   \begin{itemize}
   \item If the tag of $v$ is not $\ast$, then $v$ and $\beta(v)$ have the same element tag.
   \item $\beta(v)$ satisfies any conditions associated with $v$ (ex. “\texttt{\$\not=$\ MI}” at vertex 5 in Figure 2a).
   \end{itemize}

5. For arc $(u, v)$, if both $\beta(u)$ and $\beta(v)$ are defined: If $(u, v)$ is a \texttt{c-arc}, $\beta(v)$ is a child of $\beta(u)$.

   If $(u, v)$ is a \texttt{d-arc}, $\beta(v)$ is a descendant of $\beta(u)$.

By 1) and 2) above, $\beta(v)$ is defined for $v = 1, 2, 3, 4, 5, 9, 10$. Also, either all or none of \{$\beta(6), \beta(7), \beta(8)$\} are defined.

Now, consider queries containing more complex features. \textit{Value} aggregation is specified as a condition on a vertex in $GTP(Q)$. For instance, in $GTP(Q_1)$, we can attach the condition
\( \text{count(phones/phone)} \geq 15 \) to vertex 3. \textit{Structural} aggregation, whereby nodes are grouped together, is handled via nested queries, as discussed later.

Now, consider queries containing quantifiers. A query can be rewritten to eliminate the \textit{SOME} quantifier. The following query \( Q_2 \) contains the \textit{EVERY} quantifier (Figure 2b).

\[
\text{FOR } \$o \text{ IN } //\text{open_auctions/open_auction} \\
\text{WHERE EVERY } \$b \text{ IN } \$o//\text{bidder[sex = “male”]} \text{ SATISFIES } \$b/\text{increase} > 100 \\
\text{RETURN } <\text{result}> \{\$o\} </\text{result}>
\]

In \( GTP(Q_2) \), instead of the default arc label \textit{SOME}, the arc from \( \$o \) to \( \$b \) is marked \textit{EVERY}. It means that, for each document node \( \$o \) binds to, each of its \textit{bidder} descendants must satisfy the following: \textit{IF} it has a \textit{sex} subelement with value \textit{“male”}, \textit{THEN} it must have an \textit{increase} subelement with value \textit{> 100}. The if-then-else clause of XQuery is handled similarly, using \textit{IF}, \textit{THEN} and \textit{ELSE} labels on the arcs.

The following query \( Q_3 \) involves grouping, nesting, structural aggregation and value joins. It is modeled by \( GTP(Q_3) \), and the set \( J(Q_3) \) of value join conditions (Figure 2c).

\[
\text{FOR } \$p \text{ IN } //\text{person} \\
\text{LET } \$a := \text{FOR } \$t \text{ IN } //\text{closed_auction} \\
\text{WHERE } \$p/@id = \$t/buyer/@person \\
\text{RETURN } <\text{item}> \{\text{FOR } \$t2 \text{ IN } //\text{europe/item} \\
\text{WHERE } \$t/itemref/@item = \$t2/@id \\
\text{RETURN}\{\$t2/name\}\} </\text{item}> \\
\text{WHERE } \$p/\text{age} > 25 \text{ RETURN } <\text{person}> \{\$p/name\} \{\$a\} </\text{person}>
\]

The outermost FWR expression in \( Q_3 \) corresponds to the vertices 1, 2, 4 and 5, in \( GTP(Q_3) \). In the \textit{LET} clause, the outer FWR expression corresponds to vertices 3, 6, 7 and 8; the inner FWR expression corresponds to vertices 9 through 14.

In general, an XQuery query \( Q \) could contain the boolean operators \textit{and}, \textit{or} and \textit{not}. Ramanan [24] showed how to represent \textit{or} and \textit{not} in tree patterns. \( Q \) can be represented by a tree
\(GTP(Q) = (V, A)\) (along with \(J(Q)\)), where \(V\) is a set of vertices, and \(A\) is a set of arcs. Each vertex \(v \in V\) has a tag \(\tau(v)\), and a boolean operator \(bool(v)\) associated with it. \(\tau(v) \in \Sigma \cup \{\ast\}\) is the element type of \(v\); \(\ast\) denotes ‘any’ type. \(Bool(v) \in \{\text{and, or, not}\}\); the default is \text{and}. Each arc \(r \in A\) has an axis \(axis(r)\) and a label \(label(r)\) associated with it (in addition to mandatory/optional). \(Axis(r) \in \{\text{self, child, descendant}\}\); \text{self} axis is used while representing or and \text{not}. \(Label(r)\) can be \text{SOME, EVERY, IF, THEN or ELSE}; the default is \text{SOME}. In addition, there could be a “\(<\text{aggop}> <\text{relop}> \text{const}\)” condition associated with a vertex \(v\) (ex., \(\neq "MI"\) at vertex 5, or \(\text{count(phones/phone)} \geq 15\) possibly at vertex 3, in Figure 2a). \(<\text{RelOp}>\) denotes a relational (comparison) operator, and \(<\text{aggop}>\) denotes a possible aggregation operator.

For example, consider the XPath query \(Q_4 = //a[b>2] \text{ and not } c]//a \text{ or } /*//b\); \(GTP(Q_4)\) is shown in Figure 3. For each vertex \(v\), the pair \((\tau(v), bool(v))\) is shown next to \(v\); if \(bool(v)\) is not specified, default is \text{and}. For each arc \(r\), \(axis(r)\) is shown next to it: \(s, c\) and \(d\) stand for \text{self, child and descendant}, respectively.

In general, \(|GTP(Q)|\) is linear in \(|Q|\). The \textit{distinguished vertices} in \(GTP(Q)\) are those vertices whose images in \(D\) (under embeddings) determine \(output(Q, D)\). Let \(dv(Q)\) denote the ordered list of distinguished vertices of \(GTP(Q)\); the order is determined by XQuery semantics. An XPath query \(Q\) (without union) has exactly one distinguished vertex: It corresponds to the node test in
the last location step (ex. vertex 2 in Figure 3). For an XQuery query $Q$, $dv(Q)$ consists of the
vertices corresponding to: the variables defined in the $\text{FOR}$ clause, and the output fragments in the
$\text{RETURN}$ clause. For $Q_1, Q_2$ and $Q_3$ (Figure 2), we have $dv(Q_1) = (3, 9, 7, 11)$, $dv(Q_2) = (3)$ and
$dv(Q_3) = (2, 5, 6, 12, 14)$. Output($Q, D$) is determined by $\text{bindings}(dv(Q))$. By XQuery syntax,
we have

**FACT 2.1.**  
\begin{enumerate}
  \item A non-distinguished query vertex can have distinguished descendants along only one
  of its outgoing branches.  
  \item The path from the root to a distinguished vertex contains only and
  vertices and $\text{SOME}$ arcs.
\end{enumerate}

### 3 Comparison with Holistic Twig Join

A twig [6] is a tree pattern that represents a simple XPath query not involving the operators $\text{or}$ and
$\text{not}$, and library functions; for example, see Figure 4a. For generality, we let a twig have
several distinguished vertices, subject to Fact 2.1. Bruno et al. [6] presented a holistic twig join
($HTJ$) algorithm for finding all the matches for a twig $Q$ in a document $D$; as for our approach,
output($Q, D$) needs to be obtained from these matches. We first compare our algorithm with HTJ,
for twigs, and then consider problems in extending HTJ to GTPs.

Bruno et al. [6] assumed that $D$ has been stored according to certain XML storage strategies
[2]. For each vertex $j \in Q$, these strategies output a stream, $\text{stream}(j)$, of nodes that are possible
matches for $j$ (oblivious to the other vertices in $Q$). Our algorithm can also be modified to take these
streams as input; see Section 11. Then our runtime and disk I/O for finding $D''$ are linear in the
input size $|I|$; sum of sizes of the streams, for the relevant vertices in $GTP(Q)$. This runtime and
disk I/O are trivially optimal.

individual root-to-leaf query paths are output. These document paths are output in sorted root-to-
leaf document order; this requires the “blocking” of the paths from output, and writing them to the
disk, until all such paths involving ancestors of the nodes on the path have been computed. So, this
part requires two passes. In Part II, the document paths are merge joined to produce the node
tuples of bindings for the vertices in $Q$. In Part I, only those document paths that are likely (but not
guaranteed) to merge join with at least one match for each root-to-leaf query path are output; if $Q$
contains only $d$-arcs, this merge joinability of the paths is guaranteed. If the output of Part I fits in
the memory, then Part II can be done in memory. Else, Part II would need an extra pass.

HTJ algorithm uses $O(d|Q|)$ memory space, same as ours. HTJ constructs node tuples of bind-
ings for all the vertices in $Q$; i.e., it outputs $bindings(Q)$ instead of $bindings(dv(Q))$. When $Q$
contains only $d$-arcs, Part I can be modified to output path matches only for the distinguished ver-
tices. Then, Part II constructs tuples of bindings only for $dv(Q)$, as our algorithm does. For this
case, their runtime and disk I/O are linear in the input and output sizes; this is same as ours.

Now, consider the general case when $Q$ contains both $c$-arcs and $d$-arcs. Choi et al. [9] consid-
ered a large class of algorithms that includes HTJ. They showed that no algorithm in this class can
find the matches for $Q$ efficiently. When $Q$ contains $c$-arcs, Part I of HTJ could output many docu-
ment path matches that do not contribute to the result of Part II; such paths are said to be useless.
Lu et al. [18] considered the evaluation of certain queries on the TreeBank dataset [29]. For the
query $Q = S[/JJ]/NP$, only 10 of the 70988 paths output by Part I were useful.

A vertex in $Q$ is a branching vertex if it has at least two outgoing arcs. A branching vertex
is a bad branching vertex (BBV) if it has at least one outgoing $c$-arc. Lu et al. [18] presented a
modification of Part I; it periodically looks ahead and reads upto $d = \text{depth}(D)$ elements from each
input stream, and caches them in memory. From now onwards, “Part I” will refer to this new version.
If $Q$ contains no BBV, Part I outputs only useful paths. In their experiments, for queries containing
a BBV, Part I still outputted 2 to 5 times as many paths as useful paths.

When $Q$ contains $c$-arcs, there is a second problem with HTJ. As per XQuery semantics, we
want distinct node tuples of bindings for $dv(Q)$, in document order. HTJ outputs tuples of bindings
for all the vertices in $Q$; several such tuples could correspond to the same binding for $dv(Q)$, thereby
artificially boosting the output size. From now onwards, this will be referred to as the elongation
problem. We need to project the output of HTJ onto \( dv(Q) \), remove duplicates, and sort in document order. This is not trivial, because the projected tuples might not be in document order and duplicates might not be adjacent. The elongation problem can be partly alleviated by suppressing (i.e., not outputting) matches for some nondistinguished vertices, in Part I; suppressing matches for the other nondistinguished vertices could result in the output of spurious tuples in Part II.

When \( Q \) contains c-arcs, due to the two problems with HTJ discussed above, its runtime and disk I/O could be much worse than ours. Orthogonally, Jiang et al. [15] and Chen et al. [7] showed how to speed up Part I by using certain indexes. These indexes allow one to skip some entries in each of the input streams. Lu et al. [19] presented a holistic twig join algorithm that uses an extended Dewey representation for the nodes. It uses streams of nodes only for the leaf vertices in the twig. All these speed ups can also be applied to our algorithm.

Let us compare the size of our intermediate result \( D'' \), with that of HTJ (output of Part I). For each node \( e \) in \( D'' \), \( O(1) \) space is needed for each distinguished vertex of \( Q \) that binds to \( e \) in some embedding. In contrast, in the output of Part I, for each node, \( O(1) \) space is needed for each path match containing that node. Since the same node-vertex binding could appear in several path matches (when \( D \) is recursive), for each of several root-to-leaf query paths containing that vertex, \( D'' \) would be a lot smaller. So, getting \( bindings(dv(Q)) \) in-memory, is more likely for \( D'' \), than for the output of Part I.

Finally, consider the problems in extending HTJ to GTPs. Jiang et al. [14] extended Part I to handle twigs containing the \( or \) operator. It suffers from the same problems as the original algorithm, when \( Q \) contains c-arcs. Also, the elongation problem is worse here, because \( or \) appears only inside the predicates of \( Q \), and predicates do not contribute distinguished vertices.

Jiao et al. [16] presented an algorithm for finding the matches for a path query (unlike a twig, it has no branches) containing \( not \), in a document. There seems to be no natural way to extend HTJ to twigs containing \( not \). This is because HTJ is monotonic, whereas \( not \) is nonmonotonic. Let \( D_2 \) be a document obtained by adding some elements to \( D_1 \). Then Part I, on input \( D_2 \), would
output all the paths it output on $D_1$, and may be some more. But, for a query $Q$ containing \texttt{not}, $\text{output}(Q, D_2)$ could be a proper subset of $\text{output}(Q, D_1)$. Specifically, consider the query $Q = //a[\texttt{not} .//b[c \texttt{and} .//d]]//e$. Here, $b$ is a bad branching vertex (BBV, defined above). So, we cannot truncate (in Part I) the document path matches for a query root-to-leaf path anywhere at or above $b$; if we did, Part II would output spurious tuples. It is not clear what should be the query root-to-leaf paths in Part I, and how Part II should path-merge-join document path matches for them. Similar problems apply to \texttt{EVERY} and \texttt{if-then-else} clauses of XQuery, and to aggregation operators used in conjunction with the comparison operators $<$ and $\leq$; they can make the query nonmonotonic.

Also, consider the query $Q = //a[.//b]//c$. In $\text{stream}(a)$ of input document nodes for the query vertex $a$, let $a_1$ be an ancestor of $a_2$; $a_1$ precedes $a_2$. In Part I of HTJ, $a_1$ will be found to be a match for query vertex $a$, before $a_2$ is found to be a match; this satisfies an important \textit{stack property} (last-in-first-out) used in Part I. Next, consider the query $Q' = //a[\texttt{not} .//b]//c$. Now, $a_1$ can be found to be a match for $a$, only after $a_2$ is found to be a match; this violates the stack property. Also, what does “document path match” mean for a query path that contains \texttt{not}, such as $//a[\texttt{not} .//b]$?

Finally, HTJ has no special provision for handling value joins. The only approach is to declare the vertices involved in the value joins (ex. 3, 8, 10 and 13 in Figure 2c) also as distinguished vertices. After obtaining the tuples of bindings for this elongated $dv(Q)$, \textit{we need to find the ones that satisfy $J(Q)$}; then the usual elongation problem discussed earlier (projection, duplicate elimination, resorting) applies. Steps 1 and 3 of our algorithm can be used to do the same. But our Step 2 is a special provision for handling value joins that avoids this elongation problem. Our Step 2 requires less resources compared to handling elongated $dv(Q)$. If the former cannot be done in-memory, neither can the latter.
4 SAX Events

Consider an XML document $D$. As in [12], we use a SAX parser to read $D$ and generate a stream of events of five types: `startDocument()`, `startElement(a, event#, level#)`, `text(s, event#)`, `endElement(a, event#)`, and `endDocument(event#)`. Event# is the SAX event number, and level# is the depth of the element in $D$. We treat attributes similarly to elements; so, the label $a$ above might be an element label or an attribute label. $s$ is a data (string) value. For example, the document

```xml
<a b="101"><c>201</c></a>
```

leads to the following sequence of events:

- `startDocument()`
- `startElement(a, 1, 1)`
- `startElement(@b, 2, 2)`
- `text("101", 3)`
- `endElement(@b, 4)`
- `startElement(c, 5, 2)`
- `text("201", 6)`
- `endElement(c, 7)`
- `endElement(a, 8)`
- `endDocument(9)`

Our SAX events described above include the event number, and are slightly different from those in [26].

A document node (element or attribute) becomes open when its `startElement` is seen; it stays open until its `endElement` is seen, at which point it becomes closed. A node is current if it is open, but none of its descendants is open. Note that a node becomes current when it opens, and stays current until one of its children opens; it becomes current again when that child closes. All the open nodes lie on the unique path from the document root to the current node; this path is the current path.

5 First Pass: The Bottom-Up Transducer

Let the vertices in $GTP(Q)$ be numbered consecutively from 1 to $m$; $m = O(|Q|)$. For $1 \leq j \leq m$, let $Q_j$ denote the subtree rooted at vertex $j$. For an element $e$ in $D$, let $D_e$ denote the document subtree rooted at $e$. The Bottom-Up Transducer (BUT) $M_1$ does the following: For each element $e \in D$, it determines those vertices $j \in GTP(Q)$ such that there exists an embedding of $Q_j$ in $D_e$.
that maps \( j \) to \( e \). Note that this depends only on the subtrees rooted at \( e \) and \( j \); so, it can be determined by the time \( e \) closes. Nodes with no such \( j \) are deleted from \( D \). The “document” \( D' \) containing the remaining nodes, with their set of \( j \) values, is written to the disk in postorder: Descendants of a node are written before the node. \( M_1 \) uses \( O(d|Q|) \) (stack) memory space and \( O(|Q||D|) \) time.

The intuition behind BUT \( M_1 \) is as follows. It considers elements \( e \in D \), in bottom-up order. It determines what subtrees \( Q_j \) can be embedded in \( D_e \), using previously computed information on what subtrees \( Q_j' \) of \( Q_j \) can be embedded in the subtrees in \( D_e \).

In general, an XQuery query could contain the boolean operators \( \text{and} \), \( \text{or} \) and \( \text{not} \). As we saw in Section 2, such a query \( Q \) can be represented by a tree \( GTP(Q) = (V,A) \), where \( V \) is a set of vertices, and \( A \) is a set of arcs. The BUT must consider the labels \( \text{bool}(v) \) for each vertex \( v \in V \), and \( \text{axis}(r) \) and \( \text{label}(r) \) for each arc \( r \in A \). To avoid cluttering our description with these details, we first describe our BUT for queries that do not contain the operators \( \text{or} \) and \( \text{not} \), value aggregations, and arc labels \( \text{label}(r) \) (except the default label SOME). Extension to general queries is tedious but straightforward. Some pointers are given later for \( \text{not} \), \( \text{EVERY} \) and value aggregation. Then, \( \text{bool}(v) = \text{and} \) for each vertex; so it can be left out. Also, \( \text{axis}(r) \) is either \( \text{child} \) or \( \text{descendant} \) for each arc. Let \( r \) be the arc from vertex \( j \) to vertex \( j' \). If \( \text{axis}(r) = \text{child} \), we say that \( j' \) is a \( c \)-child of \( j \); if \( \text{axis}(r) = \text{descendant} \), \( j' \) is a \( d \)-child of \( j \).

Now, we explain how BUT \( M_1 \) processes \( D \). \( M_1 \) maintains a stack \( S \) of records. The records in \( S \), from bottom to top, correspond to the (open) elements on the current path in \( D \). Each element \( e \) is represented by the record

\[
\text{record}(e) = (\text{label}(e), \text{SEvent}\#(e), \text{level}(e), \text{self}_e, \text{child}_e, \text{desc}_e).
\]

\( \text{Label}(e) \) and \( \text{level}(e) \) are the element label and the depth of \( e \) in \( D \), respectively; \( \text{SEvent}\#(e) \) is the SAX event number for the \text{startElement} event for \( e \). \( \text{Self}_e, \text{child}_e \) and \( \text{desc}_e \) are boolean arrays indexed from 1 to \( m \). The arrays are defined as follows: For \( 1 \leq j \leq m \),

\begin{align*}
\text{S1} & \quad \text{self}_e[j] = 1 \text{ iff there is an embedding of } Q_j \text{ in the part of } D_e \text{ seen so far, with } j \text{ mapped to } e. \\
\text{C1} & \quad \text{child}_e[j] = 1 \text{ iff there is an embedding of } Q_j \text{ in } D, \text{ with } j \text{ mapped to a child of } e \text{ seen so far.}
\end{align*}
Figure 4: (a). \( Q = //a[b \text{ and } .//c]/ * [a \text{ and } .//b] \), and (b). a document fragment

D1 \( \text{desc}_e[j] = 1 \) iff there is an embedding of \( Q_j \) in \( D \), with \( j \) mapped to a descendant of \( e \) seen so far.

EXAMPLE 5.1. Consider the operation of \( M_1 \) for the query \( Q \) in Figure 4a, on the document fragment in Figure 4b. The vertices in \( GTP(Q) \) are numbered 1 to 7 (i.e., \( m = 7 \)). After document node 3 closes, we have \( \text{self}_1 = 0010010 \), \( \text{child}_1 = 0000110 \) and \( \text{desc}_1 = 0001111 \). For example, \( \text{self}_1[3] = 1 \) because there is an embedding of \( Q_3 \) that maps vertex 3 to node 1. \( \text{child}_1[5] = 1 \) because there is an embedding of \( Q_5 \) that maps vertex 5 to node 3 (a child of node 1). \( \text{desc}_1[7] = 1 \) because there is an embedding of \( Q_7 \) that maps vertex 7 to node 5 (a descendant of node 1).

Now, consider the operation of \( M_1 \) for the query \( Q \) in Figure 4a, on an arbitrary \( D \). For a node \( e \in D \), \( M_1 \) will set \( \text{self}_e[2] \) to 1 if and when: \( \text{Label}(e) = \tau(2) \) (which is \( a \)), \( \text{child}_e[3] = 1 \), \( \text{child}_e[4] = 1 \), and \( \text{desc}_e[5] = 1 \). \( M_1 \) operates on the SAX events for \( D \). The arrays \( \text{self}_e \), \( \text{child}_e \) and \( \text{desc}_e \) are initialized on the \text{startElement} event for \( e \). The three arrays are updated as each subtree rooted at a child of \( e \) is seen, in the bottom-up processing of \( D \); this updating is as described below, for the \text{endElement} event for that child. At the \text{endElement} event for \( e \), \( M_1 \) will have the final values for these three arrays. If \( \text{self}_e \neq \vec{0} \), \( M_1 \) will write information about \( e \) to \( D' \).

Let us see how \( M_1 \) operates on each of the five kinds of SAX events. After \( M_1 \) processes an event, the invariants \( S1, C1 \) and \( D1 \) stated above would hold at the current element.

\text{startDocument}: \( S \) is initialized to empty. Current element is ‘/’, with \( \text{record}(‘/’) = (‘/’, 0, 0, \vec{0}, \vec{0}, \vec{0}) \).

\text{startElement}: The record for the current element is pushed onto \( S \). The new element \( e \) becomes
the new current element, with \( \text{self}_e = \text{child}_e = \text{desc}_e = \vec{0} \). For each leaf \( j \in \text{GTP}(Q) \) such that \( \text{label}(e) \) matches \( \tau(j) \), and there is no “\( \text{relOp const} \)” condition associated with \( j \), set \( \text{self}_e[j] = 1 \).

text: Let \( e \) be the current element. Consider a leaf \( j \in \text{GTP}(Q) \) where \( \text{label}(e) \) matches \( \tau(j) \), and there is a “\( \text{relOp const} \)” condition associated with \( j \). If the string value in the text event satisfies this condition, set \( \text{self}_e[j] = 1 \).

eEndElement: Let \( e \) be the current element that is closing. Pop the top record, \( \text{record}(e') \), from \( S \); \( e' = \text{parent}(e) \) becomes the new current element. Update \( \text{child}_{e'} \), \( \text{desc}_{e'} \) and then \( \text{self}_{e'} \) as follows, to incorporate the effect of the subtree rooted at the child \( e \): For \( 1 \leq j \leq m \),

- \( \text{child}_{e'}[j] = \text{child}_{e'}[j] \lor \text{self}_e[j] \)
- \( \text{desc}_{e'}[j] = \text{desc}_{e'}[j] \lor \text{self}_e[j] \lor \text{desc}_e[j] \)
- If \( \text{self}_{e'}[j] = 0 \) then set it to 1 if:
  - \( \text{label}(e') \) matches \( \tau(j) \),
  - For each mandatory \( c \)-child \( j' \) of \( j \), \( \text{child}_{e'}[j'] = 1 \),
  - For each mandatory \( d \)-child \( j' \) of \( j \), \( \text{desc}_{e'}[j'] = 1 \).

If \( \text{self}_e \neq \vec{0} \), write \((\text{label}(e), \text{SEvent}\#, (e), \text{EEvent}\#, (e), \text{level}(e), \text{self}_e)\) to \( D' \). \( \text{EEvent}\#(e) \) is obtained from the \text{endElement} event for \( e \).

eEndDocument: Current element must be ‘/ ’, and \( S \) must be empty. Write \((’/’, 0, \text{EEvent}\#, 0, \text{self})\) to the disk. \( \text{EEvent}\# \) is obtained from the \text{endDocument} event.

Now, let us see how to handle the \text{not} operator. Consider the query \( Q_4 \) in Figure 3. For a document node \( e \), we have \( \text{self}_e[6] = 1 \) iff \( \text{self}_e[7] = 0 \); so, \( \text{self}_e[7] \) should be computed before \( \text{self}_e[6] \), because of the \text{self} arc from vertex 6 to vertex 7.

Now, let us see how to handle the \text{EVERY} arc label. Consider the query \( Q_2 \) in Figure 2b. Since \text{EVERY} is the opposite of the default arc label \text{SOME}, we need to redefine the arrays \text{self}, \text{child} and \text{desc} as follows: \( \text{self}_e[4] = 1 \) iff \( \text{label}(e) \neq \tau(4) \) (i.e., \( \text{bidder} \)), or \( \text{child}_e[5] = 0 \) when \( e \) closes, or \( \text{child}_e[6] = 1 \). \( \text{Child}_{e'}[4] = 1 \) iff, for every child \( e \) of \( e' \), we have \( \text{self}_e[4] = 1 \); \( \text{desc}_{e'}[4] = 1 \) iff,
for every descendant $e'$ of $e'$, we have $self_e[4] = 1$. $Child_{e'}[4]$ and $desc_{e'}[4]$ can be computed by initializing them to 1, and then setting them to 0 if some child $e$ of $e'$ is found to have $self_e[4] = 0$ or $desc_e[4] = 0$. Then $self_{e'}[3] = 1$ iff $label(e') = \tau(3)$ (i.e., open_auc) and $desc_{e'}[4] = 1$ when $e'$ closes.

Finally, let us see how to handle value aggregation. Consider the constraint $count(phones/phone) \geq 15$ added to vertex 3 in $GTP(Q_1)$ (Figure 2a). We will add the two vertices 12 (phones) and 13 (phone), and the c-arcs (3, 12) and (12, 13) to $GTP(Q_1)$. For each element $e \in D$, $self_e[3]$ will be a record that, in addition to a bit, stores $count(e/phones/phone)$. Also, $self_e[12]$ would store $count(e/phone)$. Then $M_1$ would proceed as follows:

- For each child $e''$ of $e'$ that is a match for vertex 13, when $e''$ closes, increment $self_{e'}[12].count$ by one.
- For each child $e'$ of $e$ that is a match for vertex 12, when $e'$ closes, add $self_{e'}[12].count$ to $self_e[3].count$.

Then, an additional condition for $e$ to be a match for vertex 3 is that $self_e[3].count \geq 15$.

Resource requirements of $M_1$: First, consider the memory space required. Stack $S$ contains exactly the open document nodes. Each record in $S$ needs $O(|Q|)$ space; total space needed is $O(d|Q|)$.

Now, consider runtime. In the operation of $M_1$, each event takes $O(|Q|)$ time. We show this for endElement events; trivial for others. $M_1$ spends $O(|Q|)$ time on the closing element $e$. On the new current element $e'$, updating child and desc takes $O(|Q|)$ time. Updating $self_{e'}[j]$ takes time proportional to $outdegree(j)$ in $GTP(Q)$; over all $j \in GTP(Q)$, this takes time $\sum_{j \in GTP(Q)} outdegree(j) = O(|Q|)$. So, $M_1$ spends $O(|Q|)$ time for each endElement event. Since there are totally $O(|D|)$ events, runtime is $O(|Q||D|)$. $|D'|$ and disk I/O are also $O(|Q||D|)$. If input streams of nodes for individual query vertices are available, only nodes in these streams need to be considered. Then runtime is $O(|Q||I|); |D'|$ and disk I/O are $O(|I|)$; here $|I|$ denotes the sum of sizes of the input streams.
THEOREM 5.1. For each element e in D, BUT M₁ correctly determines the set of vertices j ∈ GTP(Q) for which there exists an embedding of Q_j in D_e, with j mapped to e. M₁ writes this information to the disk in postorder, in document D'. M₁ uses O(d|Q|) memory space. Runtime, |D'| and disk I/O are O(|Q||D|). If input node streams for individual query vertices are available, runtime is O(|Q||I|); |D'| and disk I/O are O(|I|).

Proof. M₁ updates the arrays self_e, child_e and desc_e only when e is the current element. Using induction on time, we can prove that invariants S₁, C₁ and D₁ above hold for e, after M₁ processes each event. Correctness of M₁ follows from S₁. The resource analysis appears above.

There are some similarities between our BUT and the XPush machine of Gupta and Suciu [12]. But, unlike our BUT, the XPush machine is automaton based. The automaton has to be stored in the memory; in the worst case, this could take memory space and runtime exponential in |Q|. This can be somewhat reduced through lazy evaluation of the automaton states. The advantage of our approach is that a state of our BUT (represented by the boolean arrays self_e, child_e and desc_e) is constructed exactly when it is needed: No guess work is involved on what states might be needed in the future. Also, automaton-based approaches, such as the XPush machine, cannot be extended to queries that involve library functions like aggregation and position, because automata cannot count unbounded values. Our BUT can be extended to such queries (outlined above for aggregation count). We believe that our BUT could be of use in many XML applications.

6 Second Pass: Top-Down Transducer for XPath Queries

In this section, we describe the second (top-down) pass in a simple setting, namely, for XPath queries without union. Such a query Q has exactly one distinguished vertex, denoted by dv(Q); it corresponds to the node test in the last location step (Ex. vertex 3 in Figure 4a).

Recall, from Section 5, that BUT M₁ writes document D' to the disk in postorder. In postorder, a tree is traversed recursively in the order T_LT_Rr, where T_L and T_R denote the left and right subtrees.
of the root, and \( r \) denotes the root. In the second pass, the Top-Down Transducer (TDT) \( M_2 \) reads \( D' \) from the disk, backwards. The nodes of \( D' \) are read in right-to-left preorder: \( rT_R T_L \); each node is read before its descendants.

Reading \( D' \) from the disk backwards can be facilitated by reading a whole disk track at a time, and then processing it backwards in the memory. The radial disk head movement is the opposite of when \( D' \) was written.

For a vertex \( v \in GTP(Q) \), let

\[
\text{bindings}(v) = \{ n \in \text{nodes}(D) | \exists \text{ an embedding } \beta \text{ of } GTP(Q) \text{ in } D \text{ such that } \beta(v) = n \}.
\]

Since \( |dv(Q)| = 1 \), \( \text{bindings}(dv(Q)) = D'' \); \( M_2 \) outputs these nodes in document \( D'' \), in the order they are read in, namely, right-to-left preorder.

**EXAMPLE 6.1.** Consider the query \( Q \) in Figure 4a. Note that node \( e \in D \) is in \( \text{bindings}(3) \) iff:

- There exists an embedding of the subquery \( Q_3 \) in \( D_e \), with vertex 3 mapped to node \( e \). This is true iff \( self_e[3] = 1 \) (computed in Section 5).

- The parent of \( e \) in \( D \) is in \( \text{bindings}(2) \).

Similarly, \( e \in \text{bindings}(2) \) iff \( self_e[2] = 1 \), and \( e \) has an ancestor \( e' \) in \( \text{bindings}(1) \). Since \( M_2 \) proceeds top-down, when we reach a node \( e \), we can determine whether it meets the second condition if: we keep track of what bindings apply to the parent of \( e \) and collectively to all ancestors of \( e \). (see arrays \( SELF \) and \( ANC_e \) below).

While processing a node \( e \) in \( D' \), \( M_2 \) constructs two boolean arrays \( SELF_e \) and \( ANC_e \), indexed 1 to \( m \). For \( 1 \leq j \leq m \),

**S2** \( SELF_e[j] = 1 \) iff \( e \in \text{bindings}(j) \).

**A2** \( ANC_e[j] = 1 \) iff \( e \) has an ancestor in \( \text{bindings}(j) \).

\( M_2 \) outputs nodes \( e \) for which \( SELF_e[dv(Q)] = 1 \). Resulting document is \( D'' \). \( M_2 \) uses \( O(d|Q|) \) memory space and runs in \( O(|Q||D|) \) time.

\( M_2 \) will not compute \( SELF_e[j] \) or \( ANC_e[j] \), for vertices \( j \) that do not have a distinguished vertex at or below them; so, by Fact 2.1, it only encounters and vertices, and some arcs.

Now, let us see how $M_2$ processes $D'$. $M_2$ maintains a stack $S$ of records. The records in $S$, from bottom to top, correspond to the (open) elements on the current path in $D'$. For an open element $e$,

$$RECORD(e) = (label(e), SEvent\#(e), EE\#(e), level(e), SELF_e, ANC_e).$$

Let vertex $1$ be the root of $GTP(Q)$ (so, $\tau(1) = '/')$. First, $record('/') = ('/', 0, EE\#, 0, self)$ is read from $D'$. If $self[1] = 0$, then $bindings(dv(Q))$ is empty. Let $self[1] = 1$. ‘/ ’ becomes the current element, with $RECORD('/') = ('/', 0, EE\#, 0, SELF, ANC)$, where $SELF[1] = 1$, $SELF[j] = 0 (j > 1)$, and $ANC[j] = \emptyset$. If $dv(Q) = 1$, output $('/', EE\# = 0, EE\#, level = 0)$ to $D''$, and stop; else initialize stack $S$ to be empty.

At any instant, let $e'$ be the current element, and let $e$ be the next element whose record is read from $D'$. While $e'$ is not an ancestor of $e$ (i.e., because of right-to-left preorder, $SE\#(e') > SE\#(e)$) do: Discard $e'$, pop the top element of $S$ and make it the current element $e'$.

Now, the current element $e'$ is an ancestor of the new element $e$. Note that, by Section 5, $e'$ is the lowest ancestor of $e$ in $D$, such that $self_{e'} \neq \emptyset$. $ANC_e$ is computed as follows: For $1 < j \leq m$, $ANC_{e}[j] = SELF_{e'}[j] \lor ANC_{e'}[j]$. $SELF_{e'}[j]$ is computed as follows. Let $j$ be the parent of $j$ in $GTP(Q)$. Consider two cases.

**Case 1.** The arc from $\hat{j}$ to $j$ is a $c$-arc. $SELF_e[j] = self_e[j] \land (e' = \text{parent}(e)) \land SELF_{e'}[j]$

**Case 2.** The arc from $\hat{j}$ to $j$ is a $c$-arc. $SELF_e[j] = self_e[j] \land ANC_e[j]$

Push $RECORD(e')$ onto $S$; $e$ becomes the current element. If $SELF_{e}[dv(Q)]$ was set to $1$ above, output $(label(e), SE\#(e), EE\#(e), level(e))$ to $D''$.

EndOfFile($D'$): Current element must be ‘/’. Stop.

**THEOREM 6.1.** $TDT$ $M_2$ correctly determines $bindings(dv(Q))$: $e \in bindings(dv(Q))$ iff $e \in D''$ iff $SELF_e[dv(Q)] = 1$. $M_2$ outputs nodes $e$ in right-to-left preorder, to document $D''$. $M_2$ uses
Proof. By induction on time, we can prove that, when $\text{SELF}_e$ and $\text{ANC}_e$ are computed, the invariants $S2$ and $A2$ hold. Correctness of $M_2$ follows from $S2$.

The stack $S$ consists of the open nodes from $D'$. Each record in $S$ takes space $O(|Q|)$; total stack space is $O(d|Q|)$. For each node $e$ in $D'$, computing $\text{SELF}_e$ and $\text{ANC}_e$ takes $O(|Q|)$ time; total time is $O(|Q||D|)$. 

Our TDT $M_2$ described above outputs the nodes in $D'' = \text{bindings}(dv(Q))$ in right-to-left preorder. It can be easily modified to output these nodes in reverse document order, as follows: Instead of outputting a node $e$ (with $\text{SELF}_e[dv(Q)] = 1$) as soon as $\text{SELF}_e$ is computed, output it only when $e$ is discarded (after seeing all its descendants). Then $D''$ will contain the output nodes in reverse document order. If $D''$ is small, it can be stored in memory, reversed, and then output in document order. Else, we need a third pass: $D''$ is written to the disk, read back in reverse, and output in document order.

Koch [17] presented an automata based, two-pass algorithm for matching XPath queries. Compared to our algorithm, their algorithm has several disadvantages.

- Their automata need to be stored in the memory. In the worst case, their size could be exponential in $|Q|$. Excluding this, both our algorithms use the same resources.

- Automaton-based algorithms cannot be extended to queries that involve library functions like aggregation and position, because automata cannot count unbounded values.

- Their algorithm ignores the order of output nodes.

- Their first (bottom-up) pass needs a new storage model: XML documents should be stored as binary trees, and the nodes should be available in bottom-up order. Additional passes are required to convert an input XML document into such a binary tree, before their algorithm can be applied.
Their algorithm cannot be extended to XQuery queries that, unlike XPath queries, have several distinguished vertices:

- Their algorithm can be easily extended to finding matches for each of the distinguished vertices. But we still need a mechanism to determine which bindings of different query vertices to different document nodes can co-occur (in a single embedding), especially when one of the variables is defined in terms of another (ex. `FOR $y IN $x / ...`). We present such a mechanism, for our approach, in Sections 7 and 8.

- As for the HTJ algorithm (see Section 3), their algorithm has no special provision for handling value joins, similar to our Step 2 discussed in Section 9.

### 7 Concepts Behind the Second Pass for XQuery Queries

In this section, we discuss some concepts behind extending our second pass to XQuery queries. XQuery queries $Q$ typically have several distinguished vertices. For convenience, we also designate $\text{root}(GTP(Q))$ (with tag = `'/`) a distinguished vertex. For some $Q$, we would designate some more distinguished vertices, as explained later.

The output of the second pass is a new “document” $D''$ that consists of those nodes in $D$ that the distinguished vertices bind to; these nodes are output in right-to-left preorder. For each node in $D''$, there is a list of distinguished vertices that bind to it (over different embeddings). $D''$ also contains information to easily determine which bindings of different vertices to different nodes can co-occur (in a single embedding). Also, in $D''$, the ancestor–descendant relationship between the nodes of $D$ is preserved. Almost every node in $D''$ contributes to $\text{bindings}(dv(Q))$, and is essential; exceptions are discussed below, under “Other distinguished vertices”. In Section 10, we show how to obtain $\text{bindings}(dv(Q))$ from $D''$.

Let us consider our example queries from Section 2. For $Q_1$ (Figure 2a), $dv(Q_1) = (1, 3, 9, 7, 11)$. $D''$ will consist of nodes that these vertices bind to. $D''$ will also contain information as to which pairs of nodes that the vertex pair (3, 7) can bind to in a single embedding; same for (3, 9) and
Note that the node pairings for (3, 7), (3, 9) and (9, 11) are independent. In XQuery, ignoring value joins (see Section 9), we need to consider only vertex pairs; this is because, each variable or output fragment is dependent on at most one other variable. Our algorithm is very efficient because it finds all the bindings for vertex 7 exactly once, and provides information as to which of them can co-occur with each binding of vertex 3; compare this to finding those bindings of 7 for each binding of 3 (which will include repetitions), or worse for each binding of (3, 9). Our algorithm will work correctly in all cases: multiple bindings for 9 for each binding of 3; bindings for 3 underneath bindings for 9, etc.

For $Q_2$ (Figure 2b), $dv(Q_2) = (1, 3)$. Now consider $Q_3$ (Figure 2c) that contains value joins. One intermediate approach is to designate the vertices of $GTP(Q_3)$ that participate in the value joins also as distinguished vertices. Towards the end of Section 3, we saw the problems associated with this approach; Section 9 provides a better approach.

The rest of this section develops the mechanism needed to compactly specify which pairs of bindings of different query vertices to different document nodes can co-occur.

**Other distinguished vertices.** Let $y$ be a FOR clause variable (or an output fragment). Suppose that $y$ is defined using an extended XPath expression $P$ starting with another variable $x$. Consider the case where $P$ contains a descendant axis (outside the predicates), but the first axis in $P$ is the child axis. This is the only case where we designate some more (already existing) vertices of $GTP(Q)$ as distinguished vertices, as follows. Let $P_1$ be the longest prefix of $P$ consisting only of child axes (outside the predicates), and let $P_2$ be the rest of $P$. So, the first axis in $P_2$ is the descendant axis. We introduce a new variable $z$ as follows: $z \in x/P_1, y \in z//P_2$. For example, $y \in x/a/b/c//d...$ is split as $z \in x/a/b/c$, and $y \in z//d...$. The vertex of $GTP(Q)$ that corresponds to $z$ becomes a distinguished vertex. The reason for this will be clear from the lemmas below. In $D''$, nodes that $z$ binds to will not contribute to $bindings(dv(Q))$; matches for $z$ will be suppressed while constructing the node tuples (to avoid the elongation prob-
lem). But each such node $Z$ has at least one descendant $Y$ that $y$ binds to, and that does contribute to $\text{bindings}(\text{dv}(Q))$.

From now onwards, we assume that additional vertices in $GTP(Q)$ have been designated as distinguished vertices, as discussed above. So, in any (extended) XPath expression in $GTP(Q)$, either the first axis is the descendant axis, or all the axes are child axes.

For a vertex $v \in GTP(Q)$, $\text{bindings}(v)$ was defined in Section 6. If $v$ corresponds to $y$, $\text{bindings}(y) = \text{bindings}(v)$. For vertices $v_1, v_2$, let $\text{bindings}(v_1, v_2) = \{(n_1, n_2) \mid \exists \text{ an embedding } \beta \text{ of } GTP(Q) \text{ in } D \text{ such that } (\beta(v_1), \beta(v_2)) = (n_1, n_2)\}.$

The following two lemmas help us deduce $\text{bindings}(v_1, v_2)$, from $\text{bindings}(v_1), \text{bindings}(v_2)$ and some additional information stored with the nodes in $\text{bindings}(v_2)$; they will be used in the next section.

**LEMMA 7.1.** Let $y$ be a variable defined as: $y \text{ in } z//\cdots$ (i.e., the first axis is the descendant axis). Let $Z_1, Z_2 \in \text{bindings}(z)$, where $Z_1$ is an ancestor of $Z_2$. Let $Y \in \text{bindings}(y)$ be a descendant of $Z_2$. If $(Z_2, Y) \in \text{bindings}(z, y)$ then $(Z_1, Y) \in \text{bindings}(z, y)$.

In $D''$: For each $Y \in \text{bindings}(y)$, we will have a pointer $\text{lowestAnc}[z]$ that points to the lowest ancestor $Z$ such that $(Z, Y) \in \text{bindings}(z, y)$. Then $(Z', Y) \in \text{bindings}(z, y)$ for all ancestors $Z' \in \text{bindings}(z)$ of $Y$ that are also ancestors of $Z$ (i.e., $Z'.SEvent# < Y.\text{lowestAnc}[z].SEvent#$). The $\text{lowestAnc}$ pointer is labeled “all” to indicate this. In contrast, if the XPath expression defining $y$ in terms of $z$ involved only child axes steps then, for each $Y \in \text{bindings}(y)$, there is exactly one $Z \in \text{bindings}(z)$ such that $(Z, Y) \in \text{bindings}(z, y)$; in this case, the $\text{lowestAnc}$ pointer from $Y$ to $Z$ is labeled “only”.

**LEMMA 7.2.** Let $y$ a variable be defined as: $y \text{ in } z//\cdots$ (i.e., the first axis is the descendant axis). Let $Y_1, Y_2 \in \text{bindings}(y)$, where $Y_1$ is an ancestor of $Y_2$. Let $Z \in \text{bindings}(z)$ be an ancestor of $Y_1$. If $(Z, Y_1) \in \text{bindings}(z, y)$ then $(Z, Y_2) \in \text{bindings}(z, y)$.

**EXAMPLE 7.1.** Consider the vertex pair $(3, 7)$ in $GTP(Q_1)$ (Figure 2a). In the XPath expression
defining vertex 7 in terms of vertex 3, the first axis is the descendant axis. In an XML document
$D$, for each node $n_7 \in \text{bindings}(7)$, there could be many ancestor nodes $n_3 \in \text{bindings}(3)$, such
that $(n_3, n_7) \in \text{bindings}(3, 7)$. With $n_7$, we will store the field $\text{lowestAnc}[3]$ whose value is the
SEvent# of the lowest ancestor $n_3$ such that $(n_3, n_7) \in \text{bindings}(3, 7)$. By Lemma 7.1, for any an-
ccestor $n'_3 \in \text{bindings}(3)$ of $n_7$, $(n'_3, n_7) \in \text{bindings}(3, 7)$ iff $\text{SEvent#}(n'_3) \leq n_7.\text{lowestAnc}[3]$.
To indicate this, the $\text{lowestAnc}[3]$ field is labeled “all”.

By Lemma 7.2, if $(n_3, n_7) \in \text{bindings}(3, 7)$, then $(n_3, n'_7) \in \text{bindings}(3, 7)$ for all descendants
$n'_7 \in \text{bindings}(7)$ of $n_7$.

8 Second Pass: Top-Down Transducer for XQuery Queries

In this section, we extend the second pass to XQuery queries. As in Section 6, TDT $M_2$ processes $D'$
in right-to-left preorder. While processing a node $e$ in $D'$, it constructs two arrays $\text{SELF}_e$ and $\text{ANC}_e$
(indexed from 1 to $m$) of records. Each record has three fields: $\text{binds}$ (boolean), $\text{lowestAnc}$ (SAX event#), and $\text{category}$ (“all” or “only”). The $\text{binds}$ field serves the same purpose as the boolean
array entry of Section 6. The use for the latter two fields was explained in Section 7 (following
Lemma 7.1); they are used to indicate which bindings of different vertices to different nodes can
coop-occur. To compute these two fields, we need an additional fourth field, $\text{lowest}$ (a SAX event#),
in $\text{ANC}_e$.

The arrays are defined as follows: For $1 \leq j \leq m$,

S2 $\text{SELF}_e[j] = (\text{binds}, \text{lowestAnc}, \text{category})$ is defined as follows. $\text{binds} = 1$ iff $e \in \text{bindings}(j)$.

If $\text{binds} = 0$, the other two fields are undefined; let $\text{binds} = 1$. Let $j'$ be the lowest dis-
tinguished ancestor of $j$ in $GTP(Q)$. Let $e'$ be the lowest ancestor of $e$ in $D'$ such that
$(e', e) \in \text{bindings}(j', j)$. Then $\text{lowestAnc} = \text{SEvent#}(e')$. If the first (iff any) arc on the
path from $j'$ to $j$ is a c-arc, $\text{category} =$“all”, else it is “only”.

A2 $\text{ANC}_e[j] = (\text{binds}, \text{lowest}, \text{lowestAnc}, \text{category})$ is defined as follows. $\text{binds} = 1$ iff $e$ has
an ancestor in $\text{bindings}(j)$; let $f$ be the lowest such ancestor. If $\text{binds} = 0$, the other three
fields are undefined; let $\text{binds} = 1$. Then $\text{lowest} = \text{SEvent}\#(f)$. Let $j'$ be the lowest distinguished ancestor of $j$ in $GTP(Q)$. Let $f'$ be the lowest ancestor of $f$ in $D'$ such that $(f', f) \in \text{bindings}(j', j)$. Then $\text{lowestAnc} = \text{SEvent}\#(f')$. If the first (iff any) arc on the path from $j'$ to $j$ is a $d$-arc, $\text{category} = \text{“all”}$, else it is “only”.

EXAMPLE 8.1. Continuation of Example 7.1. Consider vertex $j = 7$ in $GTP(Q_1)$ (Figure 2a). Its lowest distinguished ancestor is $j' = 3$. For a node $e \in D'$, $\text{SELF}_e[7] = (\text{binds}, \text{lowestAnc}, \text{category})$ is defined as follows. $\text{Bind} = 1$ iff $e \in \text{bindings}(7)$. If $\text{binds} = 1$, $\text{lowestAnc}$ contains the $\text{SEvent}\#$ of the lowest ancestor $e'$ of $e$, such that $(e', e) \in \text{bindings}(3, 7)$. Since the first arc on the path from 3 to 7 is a $d$-arc, $\text{category} = \text{“all”}$. If the first arc on the path from 3 to 7 is a $d$-arc, $\text{category} = \text{“all”}$. $\circ$

$\text{ANC}_e[3] = (\text{binds}, \text{lowest}, \text{lowestAnc}, \text{category})$ is defined as follows. $\text{Bind} = 1$ iff $e$ has an ancestor in $\text{bindings}(3)$. Let $\text{binds} = 1$, and let $f$ be the lowest such ancestor; then $\text{lowest} = \text{SEvent}\#(f)$. The lowest distinguished ancestor of $j = 3$ is $j' = 1$. Let $f'$ be the lowest ancestor of $f$ in $D'$ such that $(f', f) \in \text{bindings}(1, 3)$. Then $\text{lowestAnc} = \text{SEvent}\#(f')$. Since the first arc on the path from 1 to 3 is a $d$-arc, $\text{category} = \text{“all”}$. $\circ$

TDT $M_2$ outputs nodes $e$ and those records $\text{SELF}_e[j]$ such that $j \in dv(Q)$, and $\text{SELF}_e[j].\text{binds} = 1$. Nodes $e$ with no such $j$ are not output. The resulting document is $D''$. $M_2$ uses $O(d|Q|)$ memory space; runtime and disk I/O are $|D'| = O(|Q||D|)$.

The operation of $M_2$ at the next element $e$ differs from that in Section 6, only in the computation of $\text{SELF}_e$ and $\text{ANC}_e$. Recall that $e'$ is the lowest ancestor of $e$ in $D'$. For $1 < j \leq m$,

$\text{ANC}_e[j] = (\text{binds}, \text{lowest}, \text{lowestAnc}, \text{category})$ is computed as follows:

If $\text{SELF}_{e'}[j].\text{binds} = 0$, then $\text{ANC}_e[j] = \text{ANC}_{e'}[j]$.

Else $\text{ANC}_e[j] = (1, \text{SEvent}\#(e'), \text{SELF}_{e'}[j].\text{lowestAnc}, \text{SELF}_{e'}[j].\text{category})$.

$\text{SELF}_e[j] = (\text{binds}, \text{lowestAnc}, \text{category})$ is computed as follows. Let $\hat{j}$ be the parent of $j$ in $GTP(Q)$. Consider two cases.

**Case 1.** The arc from $\hat{j}$ to $j$ is a $c$-arc. $\text{binds} = \text{self}_e[j] \land (e' = \text{parent}(e)) \land \text{SELF}_{e'}[\hat{j}].\text{binds}$
lowestAnc = if \( \hat{j} \) is distinguished then \( \text{SEvent}^#(e') \) else \( \text{SELF}_{v'[\hat{j}]} \).lowestAnc

category = if \( \hat{j} \) is distinguished then “only” else \( \text{SELF}_{v'[\hat{j}]} \).category

**Case 2.** The arc from \( \hat{j} \) to \( j \) is a \( c \)-arc. 

\[ \text{binds} = v[e'][\hat{j}] \land \text{ANC}_{v'[\hat{j}]} \text{.binds} \]

\( \text{lowestAnc} = \) if \( \hat{j} \) is distinguished then \( \text{ANC}_{v'[\hat{j}]} \).lowest else \( \text{ANC}_{v'[\hat{j}]} \).lowestAnc

\( \text{category} = \) “all”

**EXAMPLE 8.2.** Continuation of Example 8.1. Consider the computation of

\[ \text{SELF}_{v}[7] = (\text{binds}, \text{lowestAnc}, \text{category}) \]

Parent of \( j = 7 \) is \( \hat{j} = 6 \), and \( (6, 7) \) is a \( c \)-arc. So Case 1 above applies.

\[ \text{binds} = v[e][7] \land (e' = \text{parent}(e)) \land \text{SELF}_{v'[6]} \text{.binds} \]

Vertex 6 is not distinguished; vertex 3 is its lowest distinguished ancestor.

So, \( \text{lowestAnc} = \text{SELF}_{v'[6]} \).lowestAnc \) would point to the lowest ancestor of \( e' \) in \( \text{bindings}(3) \).

\( \text{Category} = \text{SELF}_{v'[6]} \).category; since \( (3, 6) \) is a \( c \)-arc, this would be “all”. ⊓⊔

**THEOREM 8.1.** For each element \( e \) in \( D \), TDT \( M_2 \) correctly determines the set of distinguished vertices \( j \in \text{GTP}(Q) \) such that \( e \in \text{bindings}(j) \) : \( e \in \text{bindings}(j) \) iff \( \text{SELF}_{v'[j]} \).binds = 1.

Let \( j \) be a distinguished vertex such that \( \text{SELF}_{v'[j]} \).binds = 1; let \( j \) be defined in terms of another distinguished vertex \( j' \).

- If the first (iff all) arc on the path from \( j' \) to \( j \) in \( \text{GTP}(Q) \) is a \( c \)-arc, there is exactly one element \( e' \in \text{bindings}(j') \) such that \( (e', e) \in \text{bindings}(j', j) \). \( \text{SELF}_{v'[j]} \).lowestAnc gives \( \text{SEvent}^#(e') \).

- If the first (iff any) arc on the path from \( j' \) to \( j \) in \( \text{GTP}(Q) \) is a \( c \)-arc then, for an ancestor \( e' \in \text{bindings}(j') \) of \( e, (e', e) \in \text{bindings}(j', j) \) iff \( \text{SEvent}^#(e') \leq \text{SELF}_{v'[j]} \).lowestAnc.

\( M_2 \) outputs elements \( e \) in right-to-left preorder, to document \( D'' \). \( |D''| = O(\sum_j |\text{bindings}(j)|) \), where the summation is over \( j \in \text{dv}(Q) \). \( M_2 \) uses \( O(d|Q|) \) memory space, \( O(|Q||D|) \) runtime, and \( O(|D'|) \) disk I/O. ⊓⊔

Proof is same as that of Theorem 6.1. In Section 10, we show how to obtain \( \text{bindings}(\text{dv}(Q)) \) from \( D'' \), in \( O(|\text{bindings}(\text{dv}(Q))|) \) time; this could be much larger than \( |D''| \). Since \( |D''| =
\(O(\sum_j |\text{bindings}(j)|)\), \(D''\) is the most compact representation possible, for \(\text{bindings}(dv(Q))\).

BUT and TDT can be speeded up by not computing/storing some entries in \(self_e, SELF_e\) and \(ANC_e\). Details are omitted due to lack of space.

9 Handling Value Joins

As we saw in Section 3, the HTJ algorithm \([6, 15, 14, 18, 7, 19]\) has no special provision for handling value joins. The only approach is to declare the vertices involved in value joins (ex. 3, 8, 10 and 13 in Figure 2c) also as distinguished vertices. After obtaining tuples of bindings for this elongated \(dv(Q)\), we need to find the ones that satisfy the value joins; then the usual elongation problem (projection, duplicate elimination, resorting) applies. Steps 1 and 3 of our algorithm can be used to accomplish the same. But our Step 2 (value join step), described in this section, is a special provision for handling value joins that avoids this elongation problem. It requires less resources compared to handling the elongated \(dv(Q)\). If our Step 2 cannot be done in-memory, the problem of handling the elongated \(dv(Q)\) cannot be solved in-memory. This is because our Step 2 is performed on minimum sets of values. The elongated tuple approach considers all tuples containing these values, and there could be too many such tuples.

Our Step 2 is performed in between the top-down and bottom-up passes. Our algorithm of sections 5 through 8 is modified as follows. In Step 1, BUT \(M_1\) collects appropriate \((node, string-value)\) pairs that are eligible to participate in the value joins; they are written to the disk, outside \(D'\). In Step 2, we perform the value joins \(J(Q)\) on these pairs. In Step 3, TDT \(M_2\) is modified to use the result of Step 2.

The value joins (Step 2) should be performed at the lowest possible level in \(D\). If this is below the root level, then the result of the value joins should be incorporated into Step 1; this amounts to integrating Steps 1 and 2. For example, in query \(Q_3\) (Figure 2c), consider adding the value join \(buyer.@person = itemref.@item\) (please ignore the absurdity). This value join should be performed at each node \(e\) that could be a match for vertex 6 (lowest common ancestor of vertices 8
and 10). If the result of the value join is empty, then $self_e[6]$ should be set to 0 in Step 1.

Now, we explain the modifications to our algorithm, using the original example query $Q_3$ (Figure 2c). We have $dv(Q_3) = (1, 2, 5, 6, 12, 14)$. Note that, vertices that participate in the value joins (i.e., vertices 3, 8, 10, 13) are not distinguished vertices now. $D''$ will contain bindings for the distinguished vertices, as well as information on which bindings can co-occur. For vertex pairs (2, 5) and (12, 14), information on which bindings can co-occur will be given as explained in Section 8.

What is new is how the corresponding information is specified for vertex pairs (2, 6) and (6, 12). For (2, 6), this information is specified as follows: For each node $e \in \text{bindings}(2)$, we will output a set $set_e(2, 6)$ of bindings for vertex 6 that can co-occur with $e$; similarly for (6, 12).

Our algorithm is modified as follows. With vertex 2 of $GTP(Q_3)$, associate a relational table $TableA(SEvent\#A, valueA)$. In BUT $M_1$, at each person node $e$ with $self_e[2] = 1$, collect the values for all its @id children (for generality, ignore the constraint that a person node can have at most one @id child). In TableA, we will place one tuple for each @id child: $SEvent\#A$ is the start event# for the person node (not the @id node), and $valueA$ is the value of the @id child. Note that, since $self_e[2] = 1$, $e$ has a child node labeled age, with value > 25.

$TableA$ is constructed as follows. It is stored on the disk outside of $D'$. When $M_1$ sees a person node $e$ (a possible match for vertex 2), $M_1$ will carry $SEvent\#(e)$ down to $e$’s children. At each @id child, it will write a tuple ($SEvent\#(e)$, @id_value) to $TableA$. When $e$ closes, if $self_e[2] = 1$, then these tuples will remain in $TableA$. Instead, if $self_e[2] = 0$ (for example, because $e$ closed without an age > 25 child), all these tuples (with first component $SEvent\#(e)$) will be erased from $TableA$. Note that these tuples are at the top of $TableA$; so, for writing and erasing, $TableA$ behaves like a stack. This would hold even if, in $GTP(Q_3)$, vertex 3 is a c-child of vertex 2.

With vertex 6 of $GTP(Q_3)$, associate two tables: $TableB(SEvent\#B, valueA)$ and $TableC(SEvent\#B, valueB)$. In BUT $M_1$, at each closed_auction node $e$ with $self_e[6] = 1$, collect the values for all buyer.@person and itemref.@item nodes below it. In $TableB$, place one tuple for each such buyer.@person node: $SEvent\#B = SEvent\#(e)$, and $valueA$ is the value
of the buyer.@person node. Similarly for TableC.

With vertex 12 of \( GTP(Q_3) \), associate the table \( TableD(SEvent\#C,valueB) \). In BUT \( M_1 \), at each item node \( e \) (with \( self_e[12] = 1 \)) whose parent \( e' \) is labeled europe, \( (e' \) is the top element in stack \( S \) of \( M_1 \), collect the values for all its @id children. In TableD, place one tuple for each such @id child: \( SEvent\#C = SEvent\#(e) \), and \( valueB \) is the value of the @id child.

In Step 2, first compute the value join \( TableA \bowtie_L TableB \bowtie_L TableC \bowtie_L TableD \), where \( \bowtie_L \) denotes the left outerjoin operator; this reflects the mandatory/optional labels of the arcs in \( GTP(Q_3) \). Then compute its projection \( J \) onto the schema \( (SEvent\#A,SEvent\#B,SEvent\#C) \).

LEMMA 9.1. The value of \( J \) computed above is final: It contains all and only the node tuples that bind to the vertex triple \((2,6,12)\) in \( GTP(Q_3) \) (thereby satisfying both the value join constraints and the structural constraints).

Proof. The proof follows from the fact that each tuple in \( TableA \) has an \( SEvent\#A \) that corresponds to an element \( e \) with \( self_e[2] = 1 \); similarly for \( TableB, TableC \) and \( TableD \). □

This is in contrast to the value join performed on all //person/@id, //closed_auction/buyer/@person, //closed_auction/itemref/@item, and //europe/item/@id values from \( D \). Some tuple from this join result could drop out when structurally joined, for example because it involved the @id of a person with \( age \leq 25 \).

Continuing with our algorithm, consider the modifications to our second pass, to get \( D'' \). Let \( J_{AB} \) and \( J_{BC} \) be the projections of \( J \) onto the schemas \( (SEvent\#A,SEvent\#B) \) and \( (SEvent\#B,SEvent\#C) \), respectively. Let \( J_A, J_B \) and \( J_C \) be the projections onto \( SEvent\#A, SEvent\#B \) and \( SEvent\#C \), respectively. In TDT \( M_2 \), the computation of \( SELF_e[j] \) (Section 8) needs to be modified, only for \( j = 2,6,12 \). For \( SELF_e[j] \) to be set to 1, the following additional condition must be satisfied: for \( j = 2,6 \) or 12, \( SEvent\#(e) \) must be in \( J_A, J_B \) or \( J_C \), respectively. Note that, for \( j = 2 \), this condition is vacuous due to \( \bowtie_L \). If \( SELF_e[2] \) is set to 1, then \( M_2 \) associates with \( e \) the set of bindings for vertex 6 that can co-occur with the binding of vertex 2 to \( e \), as follows:
\[ set_e(2, 6) = \Pi_{SE\text{vent}\#B}(\sigma_{SE\text{vent}\#A=SE\text{vent}\#(e)}(JAB)). \]

Similarly, for each node \( e \) such that \( SELF_e[6] \) is set to 1, the \( set_e(6, 12) \) (similar to \( set_e(2, 6) \) above) is associated with it.

**Resource Requirements of the Modified Algorithm:** Steps 1 and 3 still use only \( O(d|Q|) \) memory space and \( O(|Q||D|) \) time. We still have \( |D''| < |\text{bindings}(dv(Q))| \).

Also, \( \text{bindings}(dv(Q)) \) can be computed from \( D'' \) in \( O(|\text{bindings}(dv(Q))|) \) time (see Section 10).

Consider the resources for computing the value joins (Step 2). Our value joins only involve elements that satisfy the structural constraints in \( GTP(Q) \), and all the tuples in the join result are absolutely essential. This is because our BUT \( M_1 \) removes useless tuples from the input to the value join computations. If the structural constraints in \( GTP(Q) \) are very selective, \( M_1 \) would find only few elements that are eligible to participate in the computation of the value joins. Then, BUT \( M_1 \) can output the tables (ex., Tables A through D above) to the memory, instead of disk; Step 2 can be performed in memory.

### 10 Obtaining Node Tuples of Bindings from \( D'' \)

The output of our Step 3 is a new “document” \( D'' \) that consists only of those nodes in \( D \) that the distinguished vertices bind to. These nodes are output in right-to-left preorder. For each node in \( D'' \), there is a list of distinguished vertices that bind to it (over different embeddings). \( D'' \) also contains information to easily determine which bindings of different vertices to different nodes can co-occur (in a single embedding).

In this section, we show how to obtain \( \text{bindings}(dv(Q)) \) from \( D'' \), in \( O(|\text{bindings}(dv(Q))|) \) time. We do this through our example queries \( Q_1, Q_2 \) and \( Q_3 \) (Section 2). First, consider \( Q_2 \) (Figure 2b). Since \( |dv(Q_2)| = 1 \), \( \text{bindings}(dv(Q_2)) = D'' \). \( D'' \) contains these nodes in right-to-left preorder. They can be output in document order, as explained in Section 6.

Next, consider query \( Q_1 \) (Figure 2a). \( D'' \) consists only of those nodes that the distinguished vertices 3, 7, 9 and 11 bind to. On \( D'' \), a straightforward translation (using Theorem 8.1) of the
FWR expression given in Section 2 reduces to the following:

\[
\text{FOR } p \text{ such that } SELF_p[3].binds = 1, \\
\quad \text{descendant } l \text{ of } p \text{ such that } SELF_l[9].binds = 1 \\
\quad \quad \text{and } SELF_l[9].\text{lowestAnc} = SEvent\#(p) \\
\quad \text{return } \langle \text{result} \rangle \\
\quad \quad \text{descendant } w \text{ of } p \text{ such that } SELF_w[7].binds = 1 \\
\quad \quad \quad \text{and } SELF_w[7].\text{lowestAnc} \geq SEvent\#(p) \\
\quad \quad \quad \quad \{\text{descendant } i \text{ of } l \text{ such that } SELF_i[11].binds = 1 \\
\quad \quad \quad \quad \quad \text{and } SELF_i[11].\text{lowestAnc} = SEvent\#(l)\rangle \quad \langle /\text{result} \rangle
\]

\( p, l, w \) and \( i \) denote document nodes that match the distinguished vertices 3, 9, 7 and 11, respectively. For each node \( e \in D'' \), whether it is a match for one of these vertices can be determined in constant time, by just testing whether \( SELF_e[j] = 1 \), for \( j = 3, 9, 7, 11 \). Note that the reduced query above has no XPath expressions, and all the non-distinguished vertices in \( GTP(Q_1) \) (as well as their matches in \( D \)) are now irrelevant. Since the paths 3–9 and 9–11 consists of a single \( c \)-arc, the above expression can be further simplified as follows.

\[
\text{FOR } p \text{ with } SELF_p[3].binds = 1, \text{ child } l \text{ of } p \text{ with } SELF_l[9].binds = 1 \\
\text{return } \langle \text{result} \rangle \\
\quad \{\text{descendant } w \text{ of } p \text{ such that } SELF_w[7].binds = 1 \\
\quad \quad \text{and } SELF_w[7].\text{lowestAnc} \geq SEvent\#(p)\} \\
\quad \quad \{\text{child } i \text{ of } l \text{ such that } SELF_i[11].binds = 1 \rangle \quad \langle /\text{result} \rangle
\]

Finally, consider \( Q_3 \) which involves value joins (Figure 2c). \( D'' \) consists of those nodes that the distinguished vertices 2, 5, 6, 12 and 14 bind to. On \( D'' \), the FLWR expression given in Section 2 reduces to the following:

\[
\text{FOR } p \text{ such that } SELF_p[2].binds = 1 \\
\text{let } a := \text{FOR } t \text{ such that } SEvent\#(t) \in set_p(2, 6) \\
\text{return } \langle \text{item} \rangle \\
\quad \{\text{FOR } t2 \text{ such that } SEvent\#(t2) \in set_t(6, 12) \}
\]

34
RETURN \{\text{child n2 of t2 with } SELF_{n2}[14] = 1\} \}<item>

RETURN <person> \{child n1 of p with SELF_{n1}[5] = 1\} \{$a$} </person>

$p$, $n_1$, $t$, $t_2$ and $n_2$ denote document nodes that match the distinguished vertices 2, 5, 6, 12 and 14, respectively. Note that the sets $\text{set}_{p}(2, 6)$ and $\text{set}_{t}(6, 12)$ are stored with the nodes $p$ and $t$, respectively, in $D''$. So, the conditions in the above query can be tested efficiently.

As we showed in Section 8, $D''$ is the most compact representation of $\text{bindings}(dv(Q))$ possible. If $D''$ fits in memory, $\text{bindings}(dv(Q))$ can be computed and output in document order, in-memory. If $D''$ does not fit in memory, then we need to do the following. In the top-down pass (Step 3 of our algorithm), output document path matches for individual root-to-leaf query paths (as in Part I of the HTJ algorithm [6]). Unlike in HTJ, our path matches are output only for the distinguished vertices on the query paths. In the top-down pass, we know whether a node $e$ is a match for a vertex $j$, simply by testing whether $SELF_{e}[j] = 1$. Unlike in Part I of HTJ, each node path output is guaranteed to be merge-joinable with some node path for each query root-to-leaf path, even if $Q$ has $c$-arcs, or and not. As in HTJ [6], these node paths need to be output with “blocking”, so that they are output in sorted root-to-leaf node order. Then, as in Part II of HTJ [6], these paths need to be merge joined, to produce the node tuples of bindings.

As we pointed out in Section 3, our intermediate result $D''$ would typically be much smaller than the output of Part I of HTJ. So, getting $\text{bindings}(dv(Q))$ in-memory is more likely for $D''$, than for the output of Part I. Also, Marian and Simeon [21] experimentally showed that their projection reduces the document size by a factor of 20, on the average. Our Steps 1 and 3 would reduce the document size by a much larger factor; so, we believe that $D''$ would fit in memory for many $(Q, D)$.

11 Handling Separate Node Streams

Suppose that, as mentioned in Section 3, $D$ has been stored according to some XML storage strategies [2]. For each vertex $j \in GTP(Q)$ whose incoming arc is not labeled $self$ (see Figure 3), let $\text{stream}(j)$ be the sequence of document nodes that are possible matches for $j$. Each node $e$ in
stream\(j\) is represented by a tuple \((\text{label}(e), S\text{Event}\#(e), E\text{Event}\#(e), \text{level}(e))\). Stream\(j\) is output in increasing order of \(S\text{Event}\#\).

Our algorithm for finding \(D''\) is modified as follows, to take these streams as input. First, merge all the streams into one stream (eliminating duplicate elements), sorted in increasing order of \(S\text{Event}\#\). Since each stream is sorted, this can be done using an in-memory priority queue consisting of the next element in each stream. The priority queue has \(O(|Q|)\) elements at any time (one element from each stream), so getting the next element of the merged stream takes \(O(\log |Q|)\) time; total merge time is \(O(|I| \log |Q|)\). Here, \(|I|\) denotes the sum of sizes of the input streams. Then, in the first pass (Section 5), before processing a new element \(e\), \(M_1\) would check if the current element \(e'\) should be closed; i.e., if \(E\text{Event}\#(e') < S\text{Event}\#(e)\). Also, it would check whether the new element is a child or it is a descendant of the current element, and accordingly set the arrays \(\text{child}\) or \(\text{desc}\), as done in the second pass (Sections 6 and 8).

Note that, in Section 5, \(|D'| = O|I|\). So, our runtime and disk I/O for finding \(D''\) are linear in \(|I|\). The algorithm for finding \(\text{bindings}(dv(Q))\) from \(D''\) is same as before (Section 10); its runtime and disk I/O are linear in \(|\text{bindings}(dv(Q))|\). This is trivially optimal.

Lu et al. \([19]\) presented a node labeling scheme that extends the well-known Dewey representation: From the label of a node, we can deduce the element names of all the ancestors of that node. Using this representation, they presented another holistic twig join algorithm: It uses streams of nodes only for the leaf vertices in the twig. Our algorithm can be modified to use those same streams, with the same runtime and disk I/O as theirs; but our algorithm can handle more general queries, namely GTPs. O’Neil et al. \([22]\) presented another extension of the Dewey representation that supports insertion of new nodes in \(D\); it is used in Microsoft SQL Server 2005. The labeling schemes used in \([22]\) and \([19]\) can be combined; the resulting dynamic scheme can be used in the algorithm of \([19]\), and ours.
12 Conclusions

We presented an efficient and novel algorithm for finding all the matches for a Generalized Tree Pattern \( Q \) in an XML document \( D \). It outputs a summary document \( D'' \) that is the most compact representation possible for \( \text{bindings}(dv(Q)) \). \( D'' \) could be much smaller than \( \text{bindings}(dv(Q)) \); \( \text{bindings}(dv(Q)) \) can be easily determined from \( D'' \) in \( O(|\text{bindings}(dv(Q))|) \) time.

Excluding the computation of the value joins, our algorithm makes two linear passes over the data, and runs in \( O(d|Q|) \) memory space and \( O(|Q||D|) \) time; \( d \) denotes the depth of \( D \). Its disk I/O is linear in the input and output sizes.

The only competitor to our algorithm is the Holistic Twig Join (HTJ) algorithm [6, 15, 14, 18, 7, 19]. Our algorithm is more general, as it can be applied to GTPs that involve not, quantifiers, if-then-else, aggregation and value joins. We believe that HTJ cannot be extended to such GTPs.

Finally, our algorithm can be easily extended to queries containing the preceding and preceding-sibling axes (in addition to child and descendant axes), as follows (for representing such queries as tree patterns, see [24]). For BUT \( M_1 \) in Section 5, we only need that \( self_e \) be known by the time \( e \) closes; this requirement is met for queries containing preceding and preceding-sibling axes. For each element \( e \), we define two more boolean arrays \( prec_e \) and \( precsib_e \) (in addition to \( self_e, child_e \) and \( desc_e \)) indexed from 1 to \( m \), as follows: For \( 1 \leq j \leq m \),

\[ \text{P1} \quad prec_e[j] = 1 \quad \text{iff there is an embedding of } Q_j \text{ in } D, \text{ with } j \text{ mapped to an element preceding } e. \]

\[ \text{PS1} \quad precsib_e[j] = 1 \quad \text{iff there is an embedding of } Q_j \text{ in } D, \text{ with } j \text{ mapped to a preceding sibling of } e. \]

Let \( e' \) be the parent of node \( e \) in \( D \). Note that, at any instant in the computation of \( M_1 \) while \( e \) is open, we have \( prec_e[j] = prec_e'[j] \lor desc_e'[j] \), and \( precsib_e[j] = child_e'[j] \); this is because, when \( e \) is open, the effect of the subtree rooted at \( e \) is not reflected in \( child_e' \) and \( desc_e' \).

In TDT \( M_2 \), because nodes are processed in right-to-left preorder, nodes that precede \( e \) in \( D \) are processed after \( e \). This too facilitates the handling of the preceding and preceding-sibling
axes. Similar to the ANC array for handling the descendant axis, we introduce the boolean arrays FOLL and FOLLSIB, for handling the preceding and preceding-sibling axes. Details are omitted.

XQuery evaluation is an active area of research. We believe that our algorithm provides a new approach in this area.
References


