The VLDB Journal
DOI 10.1007/s00778-012-0295-5

REGULAR PAPER

Optimal and efficient generalized twig pattern processing: a combination of preorder and postorder filterings

Radim Bača · Michal Krátký · Tok Wang Ling · Jiaheng Lu

Received: 26 October 2011 / Revised: 6 July 2012 / Accepted: 10 September 2012

Abstract Searching for occurrences of a twig pattern query (TPQ) in an XML document is a core task of all XML database query languages. The generalized twig pattern (GTP) extends the TPQ model to include semantics related to output nodes, optional nodes, and boolean expressions which are part of the XQuery language. *Preorder filtering* holistic algorithms such as TwigStack represent a significant class of TPQ processing approaches with a linear worst-case I/O complexity with respect to the sum of the input and output sizes for some query classes. Another important class of holistic approaches is represented by *postorder filtering* holistic algorithms such as Twig^{2}Stack which introduced a linear output enumeration time with respect to the result size. In this article, we introduce a holistic algorithm called GTPStack which is the first approach capable of processing a GTP with a linear worst-case I/O complexity with respect to the GTP result size. This is achieved by using a combination of the preorder and postorder filterings before storing nodes in an intermediate storage. Additionally, another contribution of this article is an introduction of a new perspective of holistic algorithm optimality. We show that the optimality depends not only on a query class but also on XML document characteristics. This new view on the optimality extends the general knowledge about the type of queries for which the holistic algorithms are optimal. Moreover, it allows us to determine that GTPStack is optimal for any GTP when a specific XML document is considered. We present a comprehensive experimental study of the state-of-the-art holistic algorithms showing under which conditions GTPStack outperforms the other holistic approaches.

Keywords XML · Query processing · Generalized twig pattern · Holistic algorithms

R. Bača (✉) · M. Krátký
Department of Computer Science, VŠB—Technical University of Ostrava, Ostrava, Czech Republic
e-mail: radim.baca@vsb.cz

M. Krátký
e-mail: michal.kratky@vsb.cz

T. W. Ling
Department of Computer Science, National University of Singapore, Singapore, Singapore
e-mail: lingtw@comp.nus.edu.sg

J. Lu
DEKE, MOE and School of Information, Renmin University of China, Beijing, China
e-mail: jiahenglu@ruc.edu.cn

1 Introduction

Query model There exist several query models which express the main functionality of query languages such as XPath or XQuery [37] that are de facto standards among XML query languages. Twig pattern query (TPQ) is the most simple model used by many approaches [1,6,24,38,46]. A TPQ is represented by a rooted labeled tree (see the Q1 query in Fig. 1a). We denote each query node by a # prefix and query nodes with the same tag are determined by apostrophes. As a result of a twig query matching, we obtain a set of occurrences (query matches) of a TPQ in an XML document. A query match is a tuple of XML nodes, where every XML node corresponds to exactly one TPQ's query node, and the relationships between XML nodes correspond to the relationships between query nodes. By processing the TPQ Q1 in the XML tree in Fig. 2a we obtain sixteen tuples (query matches), where each tuple contains five nodes, for example: $(a_1, c_2, f_1, b_1, a_2)$, $(a_1, c_2, f_1, b_2, a_2)$, and so on.

TPQ is a query model which does not reflect some important XQuery constructs. For example, it lacks the information

Published online: 10 October 2012

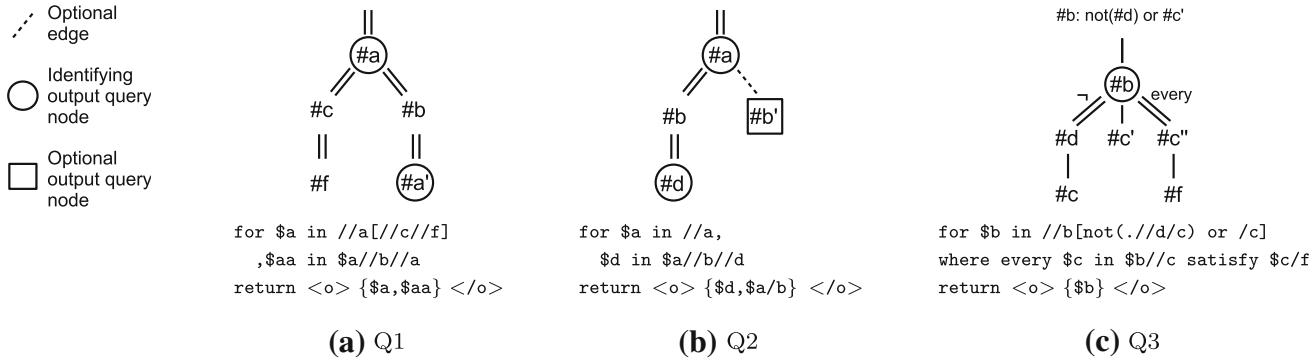


Fig. 1 XQuery queries and their corresponding GTP models

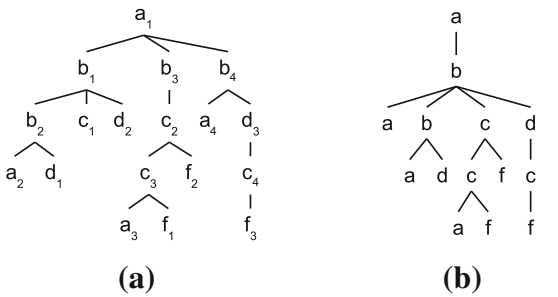


Fig. 2 **a** XML tree. **b** DataGuide of the XML tree

about output nodes in the XML query. The XQuery query Q1 from Fig. 1a contains two query nodes identifying the output tuples (we call them *identifying output query nodes*) and the Q1's result set consists only of three tuples (a_1, a_2), (a_1, a_3), and (a_1, a_4) although a TPQ processor returns sixteen query matches. The information about the output query nodes can significantly reduce the intermediate storage size since we can avoid storing and processing unnecessary nodes; therefore, it can speed up the query processing. Semantics related to the identifying query nodes is included in the generalized twig pattern (GTP) [10]. Every GTP contains also other semantic aspects of XQuery queries such as optional edges, optional output query nodes, and quantified expressions which can also be processed holistically as is shown in the Twig^{2}Stack algorithm [8]. Another challenge is represented by Boolean expressions (AND, OR, and NOT operators) which are a part of the GTP model as well; however, they are not handled by the existing GTP algorithms. We address all above mentioned constructs of GTP in this article.

Node filtering Every holistic algorithm has a filtering mechanism skipping irrelevant input data nodes which are not a part of any query match before these nodes are stored in an intermediate storage. Holistic algorithms use stacks during the filtering. If the filtering skips irrelevant nodes so that they are not stored on stacks at all, we speak about the *preorder filtering*. The *postorder filtering* skips irrelevant nodes (i.e., they are not stored in the intermediate storage) when they are popped out from their stacks. The simplest preorder filtering is represented by PathStack [6] which skips an irrelevant node n corresponding to $\#q$ when there is no occurrence of a path from $\#root$ to $\#q$ containing n. Advanced preorder filtering algorithms represent another type of algorithms which use a recursive function such as `getNext` [6] or `getPart` [15] and they skip irrelevant nodes which are not a part of a whole TPQ occurrence. On the other hand, a postorder filtering algorithm (e.g., Twig^{2}Stack [8] or TwigList [33]) skips an irrelevant node n corresponding to $\#q$ if there is no occurrence of a subtree rooted at $\#q$ containing n.

We say that a filtering is optimal for a query Q if it skips all irrelevant nodes during the sequential scan of the input, which means that an algorithm with such filtering has a linear worst-case I/O complexity for Q with respect to the TPQ result size. For example, the TwigStack [6] and TJStrictPre [15] algorithms are optimal for TPQs having only ancestor-descendant relationships. The postorder filtering algorithms introduce intermediate storages with a linear output enumeration time with respect to the result size and they are capable of a straightforward GTP output enumeration. However, the main disadvantage of the postorder filtering is that it pushes many irrelevant nodes into an intermediate storage and no optimality has been proved for it. In order to partially solve this problem, it is possible to perform the preorder filtering by PathStack and then to apply the postorder filtering as is shown in [15]. The TJStrictPre algorithm uses a combination of the advanced preorder filtering and the postorder filterings; however, it applies the postorder filtering after storing nodes in an intermediate storage. Therefore, it uses the postorder filtering to enable a linear output enumeration time, but its I/O complexity is the same as for the advanced preorder filtering algorithms such as TwigStack.

Table 1 shows the number of nodes stored in an intermediate storage by various filtering algorithms. In this table, we use three TreeBank queries specified in Sect. 8. The last column shows the number of nodes which are a part of a GTP result tuple. Evidently, the combination of PathStack and the

Table 1 Numbers of nodes inserted into an intermediate storage and numbers of nodes relevant to the GTP result

Query	Postorder	PathStack+Postorder	Advanced preorder	Nodes in GTP result
	Twig^{2}Stack	Twig^{2}Stack+PathStack	TwigStack	
TB1	172,851	92,972	32,012	804
TB2	170,874	49,765	24,834	158
TB3	404,582	187,961	12,126	63

Table 2 Number of the preorder filtering function calls corresponding to the inner query nodes

Query	`getNext`		`getPart`	
	Calls (10^3)	Unnecessary calls (10^3)	Calls (10^3)	Unnecessary calls (10^3)
TB1	8,32	184	268	53
TB2	1,658	337	145	15
TB3	3,192	1,954	515	239

postorder filtering can store an enormous number of irrelevant nodes; however, it stores less nodes than the postorder filtering itself. An advanced preorder filtering algorithm such as TwigStack stores significantly less nodes, but it still typically stores large number of irrelevant nodes due to the fact that it filters only according to the TPQ model and no postorder filtering is included.

In this article, we introduce the GTPStack algorithm combining the advanced preorder filtering and postorder filtering (let us call it a *combined filtering*) before storing a node in an intermediate storage and to our best knowledge it is the first such correct algorithm. The combined approach enables optimal filtering according to GTP; therefore, only the nodes relevant to the GTP result are stored in an intermediate storage if the algorithm is optimal. In other words, if GTPStack is optimal, then it has a linear worst-case I/O complexity with respect to the GTP result size. The combined approach significantly reduces number of nodes in an intermediate storage even if GTPStack is not optimal for a query. Moreover, in order to speed up the query processing time, we use the following two improvements in the filtering mechanism: (1) we introduce a novel advanced preorder filtering function called `getMatch` which always outperforms the `getPart` function, and (2) we avoid storing predicate nodes on stacks.

GTPStack processing time improvements Let us briefly describe our ideas behind the above improvements on examples. An advanced preorder filtering function such as `getNext` or `getPart` is typically called many times as is shown in Table 2. This table gives numbers of the function calls corresponding to the inner query nodes and numbers of unnecessary calls for three TreeBank queries specified in Sect. 8. An unnecessary call of the preorder filtering function works with exactly the same data nodes as the last function call; therefore, it returns the same query node. As observed on the query TB3, there can be almost half of function calls unnecessary. Our novel `getMatch` function avoids all these unnecessary calls. Additionally, as is shown in Sect. 8.1.2, the efficiency of the advanced preorder filtering function is significantly dependent on the ability to skip irrelevant nodes. The `getPart` and `getNext` functions sometimes return irrelevant nodes which are not subsequently stored on stacks. Another advantage of `getMatch` is that it skips all these irrelevant nodes.

We use the term *main branch query node* to name the query nodes which are on a query path from the root to an output node. The rest of the query nodes are called *predicate query nodes*. GTPStack separates the node filtering and the output enumeration which yields the following optimization. It allows us to avoid storing the nodes corresponding to the predicate query nodes on stacks. Considering the GTP Q1 and the XML tree in Fig. 2a, the nodes corresponding to $\#a$, $\#c$, and $\#f$ are processed by a preorder algorithm in the following order: a_1, c_2, c_3, f_1, f_2, c_4, f_3. Since the nodes corresponding to $\#c$ and $\#f$ are processed only to decide whether $\#a$'s nodes are relevant or not (it is because they are the predicate query nodes), then the processing of the nodes c_3, f_2, c_4, and f_3 is unnecessary because the relevance of a_1 has already been clarified by c_2 and f_1. Moreover, as is shown in Sect. 6, we can avoid storing the nodes corresponding to the predicate query nodes on stacks at all.

Optimality An important feature of holistic algorithms using the advanced preorder filtering is that they have a linear worst-case I/O complexity with respect to the TPQ result size (i.e., they are optimal) for some query classes. Different holistic approaches define their optimality conditions in a different way; however, all of them specify only the query requirements. Surprisingly, none of the existing holistic algorithms describes its optimality with respect to the XML document. By writing this article we fill this gap and describe the holistic algorithm optimality conditions which take account of the XML document characteristics as well. Moreover, we show that there is an XML document such that the optimality of the holistic algorithm is guaranteed for any twig query such as TPQ or GTP.

To our best knowledge, GTPStack is the first algorithm that is optimal for some query classes with respect to the GTP result size. GTPStack optimality is defined only by XPath axes and XML document characteristics. In other words, semantics related to the output nodes, Boolean expressions, and quantifiers do not influence its optimality.

Contributions The contributions of this article are as follows: (1) the introduction of the GTPStack algorithm with two main features: (a) it has a linear worst-case I/O complexity with respect to the GTP result size, and (b) it improves the existing filtering techniques which makes GTPStack query processing faster than other state-of-the-art holistic algorithms for a large number of queries, (2) the introduction of a new perspective of a holistic algorithm optimality since we show that the optimality depends not only on a query type but also on an XML document; we show that the knowledge about the holistic algorithm optimality can be used to speed up the query processing, (3) thorough experiments comparing state-of-the-art holistic algorithms with GTPStack; our experiments show under which conditions GTPStack outperforms the other holistic approaches and they also show that the intermediate storage size is reduced by one order of magnitude when GTPStack is used.

The paper is organized as follows. In Sect. 2, we depict a model of an XML document and the supported query constructs. Section 3 analyzes the related work. Section 4 describes issues of an algorithm combining a preorder and a postorder filtering and our solution of them. Section 5 introduces the `getMatch` advanced preorder filtering function. Section 6 introduces the GTPStack algorithm and its optimizations. In Sect. 7, we state our new optimality condition. Section 8 experimentally verifies the advantages of the GTPStack algorithm and contains results of the analysis of common XML collections in order to show the scope of our optimality condition.

2 Preliminaries

2.1 Data model

An XML document can be modeled as a rooted, ordered, labeled tree, where every tree node corresponds to an element or an attribute of the document and edges connect elements, or elements and attributes, having the parent–child relationship. We call such a representation of an XML document an *XML tree*. In what follows, we simply write 'node' instead of the correct 'tree node' or 'data node'. An example of the XML tree is shown in Fig. 2a. The nodes with the same tag name in this XML tree are preorder numbered for easy referencing in our examples.

Labeled path ℓp is a sequence of tag names $tag_{root}/\ldots/tag_n$ from the root to a node n in the XML document. In such case, we say that the node n corresponds to the labeled path ℓp. The XML tree in Fig. 2a includes several labeled paths, e.g., $a/b/a$, $a/b/b$, $a/b/b/a$, etc. The DataGuide tree [14] is a labeled tree which contains all labeled paths from the XML tree and every labeled path occurs only once there. Figure 1b depicts an example of a DataGuide for the XML document in Fig. 1a.

We assign a label to every node of an XML tree. Node labels allow us to resolve basic XPath relationships between two nodes during the query processing. There are two types of labeling schemes: (1) *element scheme* (e.g., containment scheme [46] or Dietz's scheme [12]), and (2) *path scheme* (e.g., Dewey order [35]). The main feature of labels using a path labeling scheme is that we can extract all ancestor labels. Note that the GTPStack algorithm described in Sect. 6 can be utilized with any labeling scheme which is capable of resolving ancestor-descendant (AD) and parent–child (PC) relationships between two nodes.

2.2 Query language

A *twig pattern query* (TPQ) can be modeled as an ordered rooted tree. Single and double edges between two query nodes represent PC and AD relationships, respectively. By $\#q$ we denote a query node q. For the sake of simplicity, by writing $n_{\#q}$ we mean an XML document (or data) node corresponding to $\#q$. In our algorithm, we use the following notation: `subtree`($\#q$) is a set of query nodes in the TPQ subtree rooted at $\#q$; `parent`($\#q$) is a query node being a parent of $\#q$ in the TPQ; `parentRel`($\#q$) stands for the $\langle \#q, \texttt{parent}(\#q)\rangle$ relationship; and `children`($\#q$) is a set of query nodes which are children of $\#q$ in the TPQ.

Given a query node $\#q$, a *query match of* $\#q$ is a tuple qm of nodes, for which the following conditions hold: (1) the size of qm is equal to the size of `subtree`(#q), (2) each node of qm corresponds exactly to one query node, and (3) each node $n_{\#q'} \in qm$ satisfies the relationship $\langle \#q'_{child}, \#q' \rangle$ (if there is any) with all nodes $n_{\#q'_{child}} \in qm$. A *full query match* is a query match of the root query node. Note that a node can be a part of a query match of a non-root query node, but not a part of a full query match at the same time. We say that a node $n_{\#q}$ *is matched* if it belongs to a query match of $\#q$.

Note that the query match is defined only with respect to TPQs without NOT and OR operators since it is sufficient for this article. We kindly refer to articles [18,45] which contain a precise definition of a query match for a TPQ with these operators.

Example 1 Consider the query `//a//b[./a]//c` and the XML tree in Fig. 2a. Then the tuple (b_4, a_4, c_4) is a query match of $\#b$, the tuple (a_1, b_4, a_4, c_4) is a full query match.

A *generalized tree pattern* (GTP) is an extension of TPQ, which enables expressing the XQuery semantics more accurately [10]. Edges of GTP can be mandatory and optional and they are denoted by solid and dashed lines in a tree, respectively. A mandatory edge connects a subexpression corresponding to the FOR and WHERE clauses with the rest of the query. An optional edge connects a subexpression corresponding to the LET and RETURN clauses with the rest of the query. The GTP model contains two types of output query nodes: (1) identifying output query nodes, and (2) optional output query nodes. A GTP query node is an identifying query node if it corresponds to the last query node in the FOR path expression. The nodes corresponding to the identifying query nodes form a so-called *GTP result tuple identifier*. An output of an XQuery query contains only result tuples with unique result tuple identifiers. In other words, query matches with the same identifier are represented by one GTP result tuple. An output query node is optional if it corresponds to the last query node of the LET or RETURN path expression. Optional nodes are grouped together for every GTP result tuple identifier; therefore, optional nodes of query matches corresponding to the same tuple identifier are stored in one GTP result tuple. GTP enables to model logical expressions in a query. Any query node having more than one child has a corresponding logical expression. The AND logical operator is an implicit connector for the twig query branches; therefore, we explicitly write only expressions with the NOT and OR operators as in the case of #*b* in the GTP Q3 in Fig. 1c.

Example 2 Consider the GTP Q2 in Fig. 1b and the XML tree in Fig. 2a. The node pairs corresponding to the identifying query nodes a and d form three result tuple identifiers: (a_1, d_1), (a_1, d_2), and (a_1, d_3). We can find query matches corresponding to one GTP result tuple identifier, e.g., the query matches (a_1, b_1, b_4, d_3), (a_1, b_3, b_4, d_3), and (a_1, b_4, b_4, d_3) correspond to the (a_1, d_3) result identifier. These query matches form one result tuple $(a_1, d_3, b_1, b_3, b_4)$. We can observe that the nodes corresponding to the optional output query node #*b* are grouped together in this result tuple. The GTP result of Q2 is a set of three tuples since we have three tuple identifiers. On the other hand, the TPQ result of Q2 is a set of twelve query matches.

2.3 Holistic algorithms

Holistic approaches use an abstract data type (ADT) called a *stream* which is an ordered set of node labels with the same *schema node label*. There are many options how to create the schema node labels (also known as *streaming schemes* [9] or partitions of data nodes [19]). A cursor pointing to the first node label is assigned to each stream at the beginning of a query processing. Given a stream T, let us define the following operations:

- NodeLabel `H`(T)—returns the node label at the cursor's position,
- Bool `eof`(T)—returns true if the cursor is at the end of T,
- `advance`(T)—moves the cursor to the next node label in T.

Two important assumptions ensuring the linear I/O complexity are that we can access only a single node from each stream in every step of a holistic algorithm and cursors can be only forwarded. Note that a stream ADT can be easily implemented by an inverted list.

In the following text, we use methods x.`preceding`(`y`), x.`following`(`y`), x.`ancestor`(`y`), and x.`descendant`(`y`) returning true if the corresponding relationship between the nodes x and y is satisfied. We write $x < y$ if the node x has a lower document order than the node y.

2.4 DataGuide search

The TwigStack algorithm [6] utilizes the *Tag* streaming scheme. In [9], the authors propose an extension of this algorithm by using various streaming schemes and introduce *Tag+level* and *Prefix path* streaming schemes. The Prefix path streaming scheme is basically a utilization of labeled paths [21,44]. For the sake of simplicity, let us use the abbreviation LPS for 'Labeled path streaming scheme' or 'Prefix path streaming scheme' and T+LS instead of 'Tag+Level streaming scheme'. Moreover, we use the terms 'labeled path' and 'tag+level' instead of labeled path and tag+level scheme node labels, respectively.

The content of a stream depends on the streaming scheme used. For example, in the case of LPS, the stream $T_{a/b/c}$ includes all nodes corresponding to the labeled path `a/b/c`.

Example 3 Consider the XML document in Fig. 2a. Then, for instance, $T_{a/b} = \{b_1, b_3, b_4\}$ and $T_{a/b/c/c/f} = \{f_1\}$ in the case of LPS, $T_{b,2} = \{b_1, b_3, b_4\}$ and $T_{b,3} = \{b_2\}$ in the case of T+LS.

In any algorithm using LPS or T+LS, we must first find labeled paths and tag+levels, respectively, matching the query in the DataGuide tree [3]. Searching for the matching labeled paths or tag+levels is called a *stream pruning* in [9]. By $PRU_{\#q}$ we denote a set of streams corresponding to #q. In other words, $PRU_{\#q}$ is a set of streams which can contain nodes that are a part of a full query match.

Example 4 Consider the XML document in Fig. 2a and the query `//b[./c]//d`. Under LPS, $PRU_{\#b} = \{T_{a/b}\}$, $PRU_{\#c} = \{T_{a/b/c}\}$, and $PRU_{\#d} = \{T_{a/b/b/d}, T_{a/b/d}\}$. Note that $T_{a/b/b}$ is not a part of $PRU_{\#b}$ since the necessary `a/b/b/c` labeled path is not in the DataGuide; therefore, `a/b/b` cannot contain a fully matched node.

The `isEnd`(#`q`) function returns true if all streams from $PRU_{\#q}$ are completely read (i.e., `eof`(T) returns true for

all $T \in PRU_{\#q}$). A holistic algorithm needs only a stream $T \in PRU_{\#q}$ with the smallest $\mathtt{H}(T)$ label. Therefore, we use the $\mathtt{H}(\#q)$ and $\mathtt{advance}(\#q)$ operations as follows: (1) we simply keep the set of streams $T \in PRU_{\#q}$ sorted according to their $\mathtt{H}(T)$ and $\mathtt{H}(\#q)$ then returns the head node of the first stream in the sorted set, and (2) $\mathtt{advance}(\#q)$ calls $\mathtt{advance}(T)$ on the first stream T and sorts the $PRU_{\#q}$ set. This approach was proposed in [2] and it allows us to easily process any streaming scheme together with any holistic algorithm. Otherwise said, this approach makes the usage of streaming schemes orthogonal to holistic algorithms.

3 Related work

Many different TPQ and GTP processing approaches with various features have been developed recently [1,6,8,9,24,26,38,46].

Two major types of approaches are represented by holistic joins [6,8,9,24,26] and binary structural joins [1,44,46,16]. Structural joins can be easily integrated in a full-fledged XQuery processor in order to support all XPath axes [5,10,32]; however, they can produce a large intermediate result when compared to the query result. Holistic joins can significantly reduce the intermediate result size without sophisticated query optimizations which are necessary in the case of structural joins [31]. Holistic joins can be integrated into XQuery algebra as well [30] and we can even combine them with structural joins in the same XQuery processing plan [40].

An appropriate join algorithm for a query should be picked by a cost-based optimizer as is proposed in Weiner and Härder [39]. The authors of Weiner and Härder [39] consider the selectivity of an XPath location step to be a major parameter for this cost-based optimization. However, the selection of an appropriate join is dependent on many other parameters as well. For example, we should consider the characteristics of available indices, labeling scheme, streaming scheme, granularity and accuracy of the result size estimation used, and many other details specific for each implementation. However, this is beyond the scope of this article since a meaningful cost-based comparison of structural and holistic joins should be discussed in a separate article.

In this article, we focus on holistic algorithms and push forward their theoretical frontiers and processing time capabilities. Our major motivation is the existence of the worst-case I/O complexity of holistic algorithms which is further improved by this article.

The TPQ or GTP holistic query processing consists of three phases: (1) reading nodes of an XML document from an index, (2) node filtering, and (3) storing nodes in an intermediate storage and an output enumeration. The existing optimization of the first phase includes: (a) indexed streams such as XB-tree [6] allowing a quick skipping of irrelevant input nodes, (b) compression of input data [4,17], (c) utilization of a path labeling scheme [25,26,28,43], and (d) utilization of a refined streaming scheme [9,11,20,44]. These first phase optimization techniques are special in the sense that they are orthogonal to the techniques used in the second and the third phase. Therefore, all these techniques are compatible with GTPStack as well. Moreover, we show that the holistic algorithm optimality is dependent on the XML document characteristics if a refined streaming scheme is used.

The GTPStack algorithm improves the node filtering phase (i.e., the second phase) which is described in detail in Sects. 5 and 6. The choice of an intermediate storage and an output enumeration algorithm is also important. Many holistic algorithms use the intermediate storage and the output enumeration proposed by TwigStack [6]. However, the TwigStack intermediate storage can contain many duplicate and irrelevant nodes and the whole intermediate storage is sequentially read regardless of the result size. Twig^{2}Stack [8] proposes a DOM-like intermediate storage which overcomes most of the TwigStack intermediate storage problems and enables a fast GTP enumeration in the linear time with respect to the GTP result size even though the intermediate storage contains many irrelevant nodes. The main disadvantage is that Twig^{2}Stack cannot be combined with an advanced preorder filtering function. The TwigList algorithm [33] proposes another fast intermediate storage with the linear time complexity of the output enumeration with respect to the TPQ result size which stores nodes in simple arrays minimizing the intermediate storage maintenance. TJStrictPre's and TJStrictPost's [15] intermediate storage (called the level split vectors) improves the TwigList intermediate storage, so that it can handle PC relationships more efficiently. We compared the above described features of the intermediate storages in Sect. 8.1.1 and decided to use the level split vectors intermediate storage since it has a comparable performance with the Twig^{2}Stack intermediate storage, but the level split vectors can be easily combined with an advanced preorder filtering whereas the Twig^{2}Stack intermediate storage cannot.

There are also algorithms dealing with other aspects of the XML queries such as processing content queries [41,42] or ordered queries [25,27]. All these approaches can be combined with GTPStack as well; however, it is beyond the scope of this article.

4 Analysis of node processing order

The usage of an advanced preorder filtering function causes nodes to be pushed and popped out in an order not corresponding to the document order. This is a problem if we intend to use a combined approach (i.e., a combination of the preorder and postorder filterings) since all postorder filtering

Table 3 Classification of the holistic approaches according to their push order

	Holistic algorithm
Global push order	Twig^{2}Stack [8], TwigList [33], TJStrictPost [15], PathStack [6,15]
Local push order	TwigStack [6], iTwigJoin [9], TJStrictPre [15]

Algorithm 1: Extension of the pop procedure

```
Procedure: popExtension(#q)
1 forall #child ∈ children(#q) do
2     while ¬S_#child.empty ∧
      S_#child.top.descendant(S_#q.top) do
3         pop(#child);
```

algorithms require the nodes to be popped out in the document postorder for their correct functionality [8,15,23]. Our solution of this problem consists of two parts: (1) modification of the order of the pop operation when an advanced preorder filtering is used (it is described in this section as a *relaxed pop order* and represented by the `popExtension` procedure), and (2) usage of a postorder filtering algorithm which correctly processes the query if the relaxed pop order is used (see Sect. 6.2).

Every holistic algorithm uses stacks during the query processing. There are two types of the push operation sequences used by holistic algorithms: (1) *global push order* – a node n_i is pushed onto a stack earlier than a node n_j if $n_i \leq n_j$, and (2) *local push order* – if $\#q$ is a parent of $\#q'$, then a node n_i corresponding to $\#q$ is pushed onto a stack earlier than any node n_j corresponding to $\#q'$ if $n_i \leq n_j$. Table 3 summarizes holistic approaches according to their push order. There are also other approaches such as TwigStackList [24], where nodes are not pushed on stacks in the document order even if they correspond to the same query node. In these cases, we cannot pop nodes in the order required by a postorder filtering; therefore, we omit them in this paper.

There are several types of the pop operation sequences as well. The node pop order is determined by the push order and by the pop algorithm since the nodes are popped out before we push the following nodes on stacks. The definition of a *global pop order* is analogous to the definition of the global push order. A *local pop order* pops nodes in postorder only in the case of the nodes corresponding to the same query node. The global and local pop orders correspond to algorithms having the global and local push orders, respectively. The most important thing is that if an algorithm pushes nodes in the local push order, then it cannot pop nodes in the global pop order. Moreover, if we use the local push order and try to pop nodes from stacks in the global pop order, then an intermediate storage can be in an incorrect state and some results can be lost as is shown in [15].

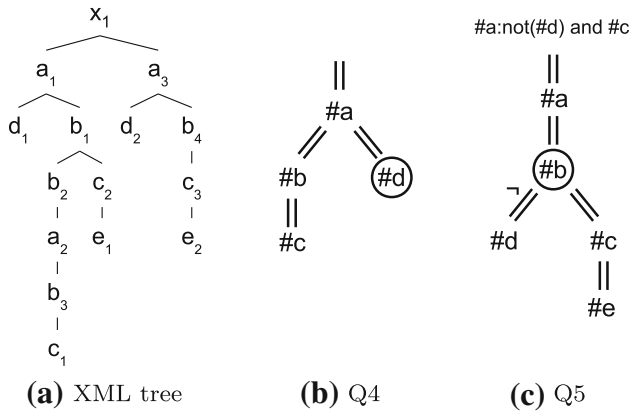
Therefore, as a part of our solution, we introduce a *relaxed pop order* which can be a pop order of a holistic algorithm having the local push order if we slightly modify the pop procedure of this algorithm. Nodes are popped out in the relaxed pop order if a node n_i corresponding to $\#q$ is removed from the stack only after we pop all descendant-or-self nodes being in a query match of $\#q$ with n_i. Even though nodes are not popped out in the global pop order, the relaxed pop order is sufficient when deciding whether a node is matched or not.

Let us recall that there is a `pop`($\#q$) procedure used by all holistic algorithms which removes a node from the stack $S_{\#q}$. In Algorithm 1, we introduce a procedure called `popExtension` which pops nodes in the relaxed pop order even though nodes are pushed in the local push order. The `popExtension` procedure has to be called at the beginning of the `pop` procedure. The `popExtension` procedure checks all stacks of child query nodes of $\#q$ and pops all nodes which are the descendants of the node $n_{\#q}$ that we want to pop out. The usage of `popExtension` allows us to combine an advanced preorder filtering with our GTPStack postorder filtering.

Example 5 Consider the TPQ Q4 and the XML tree in Fig. 4. The sequence of the push and pop operations during the query processing using an advanced preorder filtering function is shown in Fig. 3. The last row displays the pop operation sequence with the relaxed pop order. Even though it is close to the global pop order, there are still situations when the global pop order is not followed. We can see that b_2 is popped out before a_2. Due to this behavior we cannot directly combine the advanced preorder filtering with the postorder filtering such as TwigList or Twig^{2}Stack even though `popExtension` is used. In Sect. 6.2, we describe a postorder filtering technique which works correctly when the relaxed pop order is used.

Fig. 3 Example of the `push` and `pop` operation sequences during the holistic processing of Q4

	1	2	3	4	5	6	7	8	9	
local push order	a_1	b_1	b_2	a_2	b_3	c_1	c_2	d_1	a_3	...
local pop order							b_3 b_2 c_1		a_2 a_1	...
relaxed pop order							c_1 b_3 b_2	a_2	c_2 b_1 d_1 a_1	...



(a) XML tree **(b)** Q4 **(c)** Q5

Fig. 4 XML tree and two GTPs

5 GetMatch advanced preorder filtering function

The existing `getNext` and `getPart` advanced preorder filtering functions have two shortcomings: (1) they often perform unnecessary recursive calls, and (2) they return a query node #q even if there is no ancestor of `H`(#q) on $S_{parent(\#q)}$. As a result, they both cause many unnecessary computations which are completely avoided by the `getMatch` function introduced in this section.

The `getMatch` function is given in Algorithm 2. If lines of code with underlined numbers are removed, we obtain the `getNext` function originally introduced in TwigStack [6]. In other words, the underlined lines represent our novel ideas when compared to the existing algorithms. We use this notation in all algorithms of this article.

The `getMatch` function introduces three improvements of the existing advanced preorder filtering functions: (1) a mapping `Matched`: Query node $\mapsto$ Bool, which is implemented by means of an array called `Matched` so that a flag `Matched`[#q] is true if the streams' cursors in `subtree`(#q) are not moved from the last call of `getMatch`(#q), (2) a `descendantForward` procedure which forwards the cursor according to the bottom node of the parent's stack (called `bottomItem` in Algorithm 2), and (3) a cycle for the inner nodes (Lines 4–25) which does not terminate the `getMatch` call until all streams are ended or a node $n_{\#q}$ being possibly the root of a query match of #q is found. The usage of the parent's stack and the above cycle cause a more progressive forward movement of the streams' cursors which is a major parameter influencing the processing time of an advanced preorder filtering function as is shown in Sect. 8.1.2.

A `fwdToAncOf` function depicted in Algorithm 2 correctly forwards `H`(#q) according to #q's child query nodes only if the AND operator is used. An extension of `fwdToAncOf` for logical expressions with the OR and NOT operators can be found in Algorithm 5 in "Appendix 1".

Algorithm 2: `getMatch` adv. preorder filtering function

```
Function : QueryNode getMatch(#q)
1  if isRoot(#q) ∨ #q.isEnded then
2      descendantForward(#q);
3  if isLeaf(#q) then return #q ;
4  while true do
5      foreach #child ∈ children(#q) do
6          if ¬Matched[#child] then
7              #node = getMatch(#child);
8              if #node ≠ #child then
9                  Matched[#q] = false;
10                 return #node ;
11     if all streams corresponding to non-optional query nodes
       returned by getMatch() are ended then
12         #q.isEnded = true;
13         return #q ;
14     #minChild = child of #q with the minimum head;
15     fwdToAncOf(#q);
16     if H(#q) < H(#minChild) then
17         Matched[#q] = true;
18         return #q ;
19     else
20         if S_#q.empty ∨
           H(#minChild).following(S_#q.bottomItem) then
21             Pop all nodes from S_#q ;
22             advance(#minChild);
23             Matched[#q] = false;
24         else
25             return #minChild ;

Procedure: descendantForward(#q)
1  #par = parent(#q);
2  if S_#par.empty then
3      if #par.isEnded then #q.isEnded = true;
4      fwdToDescOf(#q, H(#par));
5  else
6      fwdToDescOf(#q, S_#par.bottomItem);
7      if ¬S_#par.bottomItem.AncestorOrSelf(H(#q))
       then
8          if #par.isEnded then #q.isEnded = true;
9          fwdToDescOf(#q, H(#par));

Procedure: fwdToAncOf(#q)
1  #maxChild = child of #q with the maximum head;
2  while H(#q).preceding(H(#maxChild)) do
3      advance(#q);

Procedure: fwdToDescOf(#q, label)
1  while H(#q) < label do advance(#q);
```

An important issue of the evaluation of queries with the NOT operator is that the optimality of GTPStack is necessary for skipping irrelevant nodes.

Example 6 (`getMatch` Function) Consider the TPQ Q5 and the XML document in Fig. 4. The first and the second call of `getMatch` returns #d and #c, respectively, and then their cursors are forwarded. The third call of `getMatch`

returns the root query node $\#a$ and a_1, b_1, c_2, e_1, and d_2 are the head nodes of their streams $T_{\#a}$, $T_{\#b}$, $T_{\#c}$, $T_{\#e}$, and $T_{\#d}$, respectively. Clearly, c_2 is a descendant of b_1 and d_2 is not, which corresponds to $\#b$'s logical expression. The `Matched` flag is set to true for all query nodes since every recursive `getMatch` call ends at Line 18. The cursor of $T_{\#a}$ is advanced by a holistic algorithm and a_2 is now the head of $T_{\#a}$. The fourth call of `getMatch` skips all unnecessary recursive calls of `getMatch`($\#b$) since the `Matched`[$\#b$] flag is true (see Line 6 of the `getMatch` algorithm). In other words, all head nodes corresponding to query nodes in `subtree`($\#b$) remain unchanged since the last call of `getMatch`($\#b$); therefore, it is clear that b_1 is still the root of the same query match of $\#b$.

5.1 Optimality of preorder filtering with getMatch

An advanced preorder filtering function has the following property for a specific (see below) class of queries: it returns only a query node $\#q$ such that there is a query match of $\#q$ containing `H`($\#q$). Therefore, the preorder filtering removes all nodes irrelevant to the TPQ. The `getPart` and `getNext` procedures guarantee this property for queries having only AD relationships [6,15]. Since `getMatch` only avoids unnecessary function calls and skips irrelevant nodes having no ancestor on parent's stack, its optimality properties are the same as for `getPart` and `getNext`. This means that `getMatch` is optimal for queries having only AD relationships. Let us note that in Sect. 7 we prove that a holistic algorithm with an advanced preorder filtering function (e.g., `getMatch`) can be optimal even for a query containing any combination of PC and AD relationships depending on XML document characteristics.

An extension of `getMatch` for logical expressions does not change the preorder filtering optimality which is influenced only by query nodes relationships and the XML document characteristics.

6 GTPStack

In this section, we introduce the GTPStack algorithm containing our novel postorder filtering that can be combined with any advanced preorder filtering function. This combination of filterings subsequently enables an optimization of predicate nodes reducing the GTP processing time and allows an optimal GTP processing with a linear worst-case I/O complexity with respect to the GTP result size.

The GTPStack pseudocode in Algorithm 3 shows a common holistic algorithm loop and (analogously to `getMatch`) we underline the lines' numbers with novel ideas. This algorithm performs the following steps during each iteration: (1) it calls the `getMatch` advanced preorder filtering function (Lines 1 and 10 of GTPStack), (2) it pops useless nodes from stacks (Line 4 of GTPStack and Line 2 of `push`), (3) on stacks it stores nodes having an ancestor on the parent stack (Line 3 of `push`). We introduce an optimization of the nodes corresponding to a predicate query node (Line 8 of GTPStack) in the holistic loop. These nodes (i.e., the nodes processed by the `processPredicate` procedure) are pushed on stacks only if we do not know yet whether the nodes stored on their parent stack are matched or not. By avoiding storing nodes corresponding to predicate query nodes on stacks we speed up the query processing of GTPStack as is shown in Sect. 8.1.3.

Stacks represent a core data structure for the node filtering and also for the preparation for the output enumeration. An item on a stack is represented by a node, a pointer to a node on the parent's stack (referred as a *pointerToParent* in our

Algorithm 3: Main loop of GTPStack

```
Procedure: GTPStack()
1  #q = getMatch(#root);
2  while not all streams are ended do
3  |  if ¬isRoot(#q) then
4  |  |  popPreceding(parent(#q), H(#q));
5  |  if isMainBranch(#q) then
6  |  |  push(#q);
7  |  else
8  |  |  processPredicate(#q);
9  |  advance(#q);
10 |  #q = getMatch(#root);
11 Pop the remaining items from stacks;

Procedure: popPreceding(#q, label)
1  while ¬S_#q.top.ancestor(label) do
2  |  pop(#q);

Procedure: push(#q)
1  if hasPrefixMatch(#q, H(#q)) then
2  |  popPreceding(#q, H(#q));
3  |  S_#q.push(H(#q), S_parent(#q).top);

Function : Bool hasPrefixMatch(#q, label)
1  if isRoot(#q) then return true;
2  switch parentRel(#q) do
3  |  case AD: return ¬S_parent(#q).empty;
4  |  case PC: return ¬S_parent(#q).empty ∧
   |  S_parent(#q).top.parent(H(#q));

Procedure: processPredicate(#q)
1  if isSubtreeOptimal(#q) then
2  |  if isMainBranch(parent(#q)) ∨
   |  ¬isSubtreeOptimal(parent(#q)) then
3  |  |  propagateBits(#q, true);
4  else
5  |  if bit of the S_parent(#q).top blocking info corresponding to
   |  #q is false then
6  |  |  push(#q);
```

algorithms), a *blocking info*, a (prev:next) pair, and a set of (start:end) pairs. The blocking info is a bit array used during the postorder filtering (see Sect. 6.1), the (prev:next) pair prevents an intermediate storage sorting (see Sect. 6.2.1), and the (start:end) pairs are used during the output enumeration (see Sect. 6.2.2).

6.1 Postorder filtering

In this section, we depict the `pop` procedure which represents our novel postorder filtering used by the GTPStack algorithm. The postorder filtering represented by Algorithm 4 is mainly based on an array of bits called a blocking info assigned to every node on a stack.

The blocking info array of a node $n_{\#q}$ is the same size as the number of $\#q$'s child query nodes. The bit b_i is true iff there is a matched node $n_{\#child}$ satisfying the $\langle \#child, \#q \rangle$ relationship with $n_{\#q}$. When a node is pushed onto a stack, all bits of its blocking info are set to false. The only exception is a bit corresponding to the query node $\#q_{child}$, where the relationship between $\#q$ and $\#q_{child}$ corresponds to the `every` quantifier. The `testBits` function (Line 2 of `pop`) is a simple function evaluating the query node's logical expression, where the blocking info's bits determine values of the logical variables.

Bits of the node's blocking info are set by the `propagateBits` function when the holistic algorithm pops some of its matched children. Due to the fact that nodes are popped out in the relaxed pop order, the `testBits` function has the following property: when the `pop` procedure is prepared to pop a node n_q (Line 3 of `pop`), the `testBits` function returns true if n_q is matched, otherwise it returns false.

If we utilize the blocking info, we do not have to use an intermediate storage by the postorder filtering as it is done by the Twig^{2}Stack, TwigList, TJStrictPost, and TJStrictPre approaches. In other words, we do not have to read an intermediate storage content in order to decide whether a node is useless or not. As a result, we never store nodes corresponding to predicate query nodes in an intermediate storage (since they are useless for the enumeration).

Example 7 Figure 5 depicts operations of the postorder filtering operations during the query processing of the `//a[not(./a) and ./c]//b` XPath query. This example shows how the blocking infos of the nodes a_1 and a_2 are set by `propagateBits` during the `pop` operation (Algorithm 4). We observe that the nodes corresponding to $\#a$ and $\#c$ are not pushed into an intermediate storage since they are the nodes corresponding to the predicate query nodes. The blocking info of a_2 is (0,1,1) after the steps displayed in Fig. 5 which means that the boolean expression of $\#a$ is satisfied and a_2 is pushed into an intermediate storage. On the other hand, the blocking info of a_1 is (1,1,0); therefore, a_1 is skipped.

Algorithm 4: Postorder filtering

```
Procedure: pop(#q)
1  popExtension(#q);
2  Bool matched = isSubtreeOptimal(#q) ∨ isLeaf(#q)
     ∨ testBits(#q);
3  StackItem popItem = S_#q.pop ;
4  if parentRel(#q) is labeled by the every quantifier then
5      if ¬matched then
6          propagateBits(#q, false);
7  else
8      if ¬isRoot(#q) ∧ matched ∧
       ¬isSubtreeOptimal(parent(#q)) then
9          propagateBits(#q, true);
10 if isEnumerateQueryNode(#q) then
11     if matched then
12         Add popItem into its interm. storage list;
13         Setting of ancestor's (prev:next) (Sec. 6.2.1);
14         Setting of (start:end) (Sec. 6.2.2);
15     else
16         Setting of ancestor's (prev:next) (Sec. 6.2.1);

Procedure: propagateBits(#q, Bool bit)
1  StackItem parentItem = S_#q.top.pointerToParent ;
2  foreach item in S_parent(#q) where the relationship between item
   and parentItem corresponds to parentRel(#q) do
3      item.setBit(#q, bit);
```

6.1.1 GTP postorder filtering

Note that when GTPStack is ready to pop $n_{\#q}$, then it has already popped out all descendants of $n_{\#q}$ due to the relaxed pop order (see Sect. 4). Therefore, we can easily decide whether $n_{\#q}$ is relevant or not according to the GTP semantics of the query node. We consider logical expressions, mandatory and optional edges, and quantified expressions (Lines 2-9 of `pop`). That is possible regardless of whether the advanced preorder filtering is optimal or not.

6.2 Intermediate storage

GTPStack uses a modification of the level split vectors intermediate storage (LIS) proposed in [15] which is an enhanced variant of the TwigList's intermediate storage (TIS) [33]. A shortcoming of both LIS and TIS is that if we push nodes into the intermediate storage in postorder, the enumeration outputs the result tuples unordered (which represents a problem of the TwigList and TJStrictPost algorithms). We describe a solution of this problem in Sect. 6.2.1. Section 6.2.2 depicts a new technique which is one of

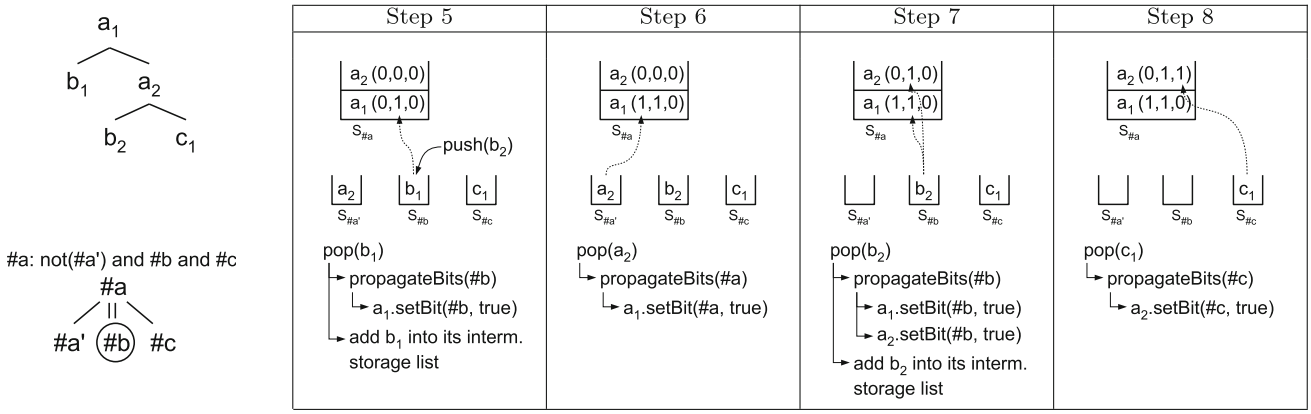


Fig. 5 Example of the postorder filtering steps which perform the bits propagation during the query processing

the GTPStack parts that enables a linear worst-case I/O complexity of GTPStack with respect to the GTP result size.

6.2.1 Sorting avoidance

Our improvement of LIS simply uses a list instead of an array as the main intermediate storage structure. The list data structure enables a concatenation of nodes according to the document order in a constant time and it avoids excessive sorting of the output.

For the purpose of the concatenation, we store a (prev:next) pair as a part of each stack item. The prev and next items are pointers to the list's node. Their values are set when we pop out a node and the detailed description of the (prev:next) pair settings is given in "Appendix 2". Note that this concatenation is necessary only when a query node has the AD relationship with its parent, because in the case of the PC relationship and LIS the nodes are pushed into the intermediate storage in the document order.

6.2.2 Preparation for enumeration

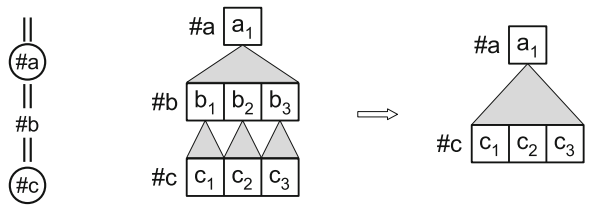
For the purpose of an output enumeration, we have a set of (start:end) pairs corresponding to a node $n_{\#q}$, where every ($\text{start}_{\#child}$:$\text{end}_{\#child}$) pair corresponds to exactly one child of $\#q$. If we consider a set S of nodes stored in an intermediate storage which correspond to $\#q$ and which are descendants of $n_{parent(\#q)}$, then the $\text{start}_{\#q}$ of $n_{parent(\#q)}$ points to $n'_{\#q} \in S$ having the lowest document order among all nodes in S and the $\text{end}_{\#q}$ of $n_{parent(\#q)}$ points to $n''_{\#q} \in S$ having the highest document order among all nodes in S.

The TwigList, TJStrictPost, and TJStrictPre algorithms read the last node stored in the intermediate storage in order to set the (start:end) pairs. However, as is described in Sect. 4, we can pop nodes in an order not corresponding to the global pop order in the case of GTPStack. Therefore, GTPStack needs to set the (start:end) pairs in a different way. We use a simple propagation technique analogous to the `propagateBits` procedure. Every time we pop a matched node $n_{\#q}$ from a stack, we set the ($\text{start}_{\#q}$:$\text{end}_{\#q}$) pairs of all nodes $n_{parent(\#q)}$ satisfying the `parentRel`($\#q$) relationship.

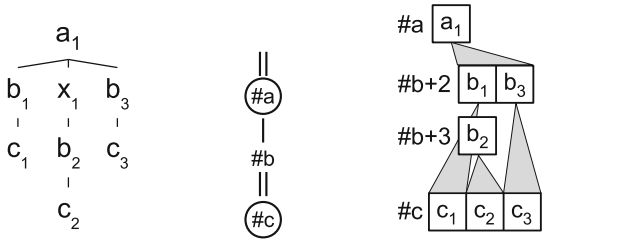
Moreover, we propose an optimization which allows us to store only the nodes corresponding to output nodes in the intermediate storage. If a main branch query node $\#q$ is not an output query node and the GTPStack algorithm is optimal for a subtree rooted at its parent query node, then nodes corresponding to $\#q$ do not have to be stored in the intermediate storage. In Example 8, we illustrate this idea and explain why the parent query node optimality is important when we want to avoid storing nodes corresponding to non-output main branch query nodes in the intermediate storage. Therefore, GTPStack does not store nodes corresponding to non-output query nodes if it is optimal for the whole query.

Example 8 Consider the XML tree in Fig. 6b, two GTPs Q5 and Q6 from Fig. 6a, c, respectively, and their intermediate storages. A gray triangle in the intermediate storage figures represents a (start:end) pair of its top corresponding node. Since the GTPStack preorder filtering is optimal for Q5, we easily avoid storing the nodes corresponding to $\#b$ since each node corresponding to $\#a$ can have a (start:end) pair pointing directly to the nodes corresponding to $\#c$. In this case, a_1 can have the (c_1:c_3) pair instead of (b_1:b_3). That is not possible in the case of Q6 for which the preorder filtering is not optimal. We see that the lists corresponding to $\#b$ can contain useless nodes and if we replace a_1's pair (b_1:b_3) by the (c_1:c_3) pair, the interval contains the irrelevant c_2 node.

Once the intermediate storage contains nodes and their corresponding (start:end) pairs, it is straightforward to enumerate the output. The new output enumeration algorithm is given in "Appendix 3".



(a) Q5 with its possible intermediate storages



(b) XML tree **(c)** Q6 with its intermediate storage

Fig. 6 Examples of GTPs and their corresponding intermediate storages

6.3 Analysis of GTPStack

The correctness of the filtering used in GTPStack follows from the correctness of both the relaxed pop order described in Sect. 4 and the postorder filtering discussed in Sect. 6.1. Furthermore, the GTPStack output enumeration is correct due to the correctness of the TwigList output enumeration algorithm. The GTPStack output enumeration only requires nodes to be in preorder in the intermediate storage, which is achieved by a list concatenation (see Sect. 6.2.1).

Although GTPStack is correct for any GTP with any combination of PC and AD relationships, we can prove the GTPStack optimality only for some types of GTPs (as in the case of any other holistic algorithm). The GTPStack optimality for a GTP corresponds to the optimality of the preorder filtering optimality for a TPQ which is discussed in Sect. 5.1. This comes from the fact that if the preorder filtering used by GTPStack is optimal, then the only irrelevant nodes stored on stacks are those corresponding to some query matches, but not corresponding to any output query nodes. As is described in Sects. 6.1 and 6.2.2, GTPStack's postorder filtering skips nodes not corresponding to any output query nodes, so that only the nodes relevant to the GTP result are stored in the intermediate storage. Therefore, GTPStack is optimal for any GTP having only the AD relationships or satisfying the conditions described in Sect. 7. When GTPStack is optimal, its worst-case I/O complexity is linear with respect to the sum of input list sizes and the size of the GTP result. The space complexity of GTPStack is linear with respect to the maximum depth of the XML tree. This follows from the utilization of stacks which store nodes with the AD relationship in each phase of the algorithm.

6.4 Summary of GTPStack

GTPStack is the first algorithm with a linear worst-case I/O complexity with respect to the sum of the input and GTP result sizes (in this case, GTPStack is optimal for the GTP). This is mainly achieved by the combination of the advanced preorder and postorder filterings and, to our best knowledge, GTPStack is the first correct holistic algorithm using a combined filtering before storing a node in an intermediate storage. The combined approach used in GTPStack has the following advantages: (1) it allows us to avoid storing nodes corresponding to predicate query nodes on stacks which speeds up the query processing, and (2) it significantly decreases the number of nodes in the intermediate storage even when GTPStack is not optimal for a query. GTPStack uses our novel advanced preorder filtering function called `getMatch`, which avoids unnecessary function calls and improves cursor forwarding which furthermore speeds up the query processing. All these features make GTPStack superior to the state-of-the-art holistic approaches as is shown experimentally in Sect. 8.

7 Preorder filtering optimality

Optimality of a preorder filtering algorithm for a query or at least subquery has several important impacts: (1) we can guarantee that we skip all nodes irrelevant to a TPQ during the preorder filtering, (2) the algorithm optimality is necessary for a more efficient preorder node filtering of a query with the NOT operator (Line 10 of `isUseless` in Algorithm 5), (3) we store only nodes corresponding to the output query nodes in the intermediate storage (see Sect. 6.2.2), and (4) we avoid storing all nodes corresponding to the predicate query nodes on stacks (Lines 1 and 2 of `processPredicate` in Algorithm 3).

Every preorder filtering algorithm has its specific query classes, for which the algorithm optimality is proved. The most common preorder algorithm optimality is the AD query (a query having only the AD relationships) optimality which is also GTPStack's optimality if the tag streaming scheme is used. However, we show that the preorder algorithm optimality can be significantly extended using the T+LS or LPS scheme.

We define the optimality of the algorithm for the TPQ model in Sect. 7.1, since the optimality of the algorithm is not influenced by the GTP model semantics. In other words, GTPStack is optimal for a GTP if it is optimal for the corresponding TPQ (see Sect. 6.3).

When we speak about a holistic algorithm, we mean a holistic algorithm using an advanced preorder filtering such as TwigStack, TJStrictPre, or GTPStack.

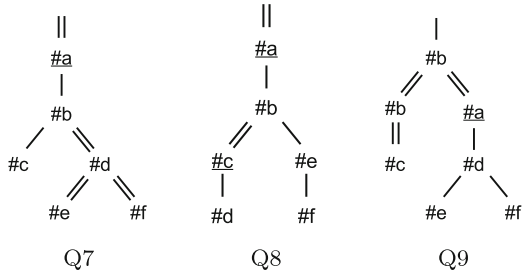


Fig. 7 Example of three TPQs and their corresponding checking query nodes (*underlined*)

7.1 Query optimality condition

We show that we can determine the algorithm optimality after the DataGuide search even for a query not belonging to any common holistic algorithm query class. Our new condition extends a known set of queries for which the current holistic algorithms are optimal. This optimality condition can be applied to any holistic algorithm using an advanced preorder filtering, since the usage of different streaming schemes is orthogonal to all holistic approaches as is mentioned in Sect. 2.4.

Let us define several terms related to the query nodes of a TPQ:

- *Query node with PC in its subtree* is a query node $\#q$ having the PC relationship between some two nodes from the $\texttt{subtree}(\#q)$ set.
- A *checking query node* $\#q$ is a query node with PC in its subtree and the AD relationship as $\texttt{parentRel}(\#q)$. It is an important query node type from the optimality point of view. $\texttt{Checkingnodes}(Q)$ is a set of checking query nodes in the query Q.
- A tag is called *single level* if all nodes in the XML tree with this tag are on the same level.

Example 9 (Checking Query Nodes) Fig. 7 shows three TPQs, where the checking query nodes are underlined. The query node $\#a$ is the single checking node of Q7 since the second query node $\#b$ with PC in its subtree has the PC relationship with its parent query node. Furthermore, for the TPQs Q8 and Q9 we have $\texttt{checkingnodes}(\text{Q8}) = \{\#a, \#c\}$ and $\texttt{checkingnodes}(\text{Q9}) = \{\#a\}$, respectively.

Definition 1 (*Query Node Optimality*) Let Q be a TPQ. We say that $\#q \in Q$ is an *optimal query node* if:

- when T+LS is used - $PRU_{\#q}$ contains one stream,
- when LPS is used - there is no labeled path $\ell p' \neq \ell p$ such that $T_{\ell p'} \in PRU_{\#q}$ and $\ell p'$ is an ancestor (i.e., prefix) of ℓp.

Our novel optimality condition for holistic algorithms is—roughly speaking—as follows: an algorithm is optimal for a TPQ if all checking query nodes are optimal after the DataGuide search. In the case of the T+LS scheme, every checking query node contains exactly one stream. In the case of the LPS scheme, no checking query node contains two labeled paths, where one is a prefix of the other. If this condition is satisfied, then a holistic algorithm using an advanced preorder filtering is optimal.

Now let us state these two lemmas:

Lemma 1 (Streams after DataGuide Search) *Let $\#q$ be a non-root node of a TPQ Q. Then, for every stream $T \in PRU_{\#q}$, there exists at least one stream T' corresponding to $\texttt{parent}(\#q)$, where the streams T and T' satisfy the $\texttt{parentRel}(\#q)$ relationship.*

Proof Since we search for the query matches of Q in a DataGuide in order to obtain the $PRU_{\#q}$ set, every $T \in PRU_{\#q}$ must be a part of a query match. Therefore, every stream T must have a parent stream T' in the query match. □

Lemma 2 (PC Relationship for Optimal Query Nodes) *Consider a TPQ Q and a set $S \subseteq Q$ of query nodes with PC in their subtrees. Moreover, assume that every $\#q \in S$ is optimal. Consider query nodes $\#q$, $\texttt{parent}(\#q) \in Q$ with the PC relationship. Then, any node n in a stream $T \in PRU_{\#q}$ having the AD relationship with a node n' from a stream $T \in PRU_{parent(\#q)}$ must also have the PC relationship with n'.*

Proof We prove this lemma by contradiction. Let us assume that there is a node n in a stream $T \in PRU_{\#q}$ having the AD relationship with a node n' in a stream $T' \in PRU_{parent(\#q)}$, where the query nodes $\#q$ and $\texttt{parent}(\#q)$ have the PC relationship and the node n does not have the PC relationship with the node n'. Consequently, by Lemma 1, there exists a stream $T'' \neq T'$ belonging to $PRU_{parent(\#q)}$. The query node $\texttt{parent}(\#q)$ contains only optimal streams since it has PC in its subtree. Obviously, in both streaming schemes the existence of the stream T'' is in contradiction with the optimality of $\texttt{parent}(\#q)$. □

In other words, Lemma 2 says that if $\#q$ is optimal, then we can replace the PC relationship between $\#q$ and $\texttt{parent}(\#q)$ by the AD relationship without modifying the query result.

If two query nodes $\#q$ and $\texttt{parent}(\#q)$ are connected with a PC relationship, then both have the same number of labeled paths after the DataGuide search. This is caused by the fact that these labeled paths in such query nodes have a one-to-one relationship. Due to this simple observation, it holds that $\#q$ is optimal iff $\texttt{parent}(\#q)$ is optimal and these query nodes have the PC relationship. As a result, we do not have to check query nodes with PC in a subtree if the parent

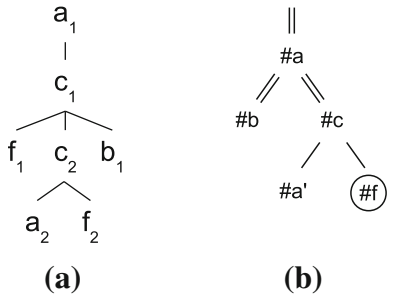


Fig. 8 **a** XML tree **b** TPQ Q10

relationship is PC and we only need to verify the optimality of streams corresponding to the `checkingnodes`(Q) set.

Definition 2 (*Optimality Condition*) We say that the *optimality condition is satisfied under an XS streaming scheme for a TPQ* Q *and an XML document if every* $\#q \in$ `checkingnodes`(Q) is optimal under XS.

Theorem 1 *(*Holistic Algorithm Optimality*) Consider an XML document and a TPQ* Q*, where the optimality condition is satisfied under an XS streaming scheme. Then, under the XS streaming scheme, the advanced preorder filtering algorithm removes all irrelevant nodes.*

Proof Our proof supposes that the advanced preorder filtering can remove all irrelevant nodes when Q has only the AD relationships. This was shown many times (see, e.g., [6,9]). If we replace all PC relationships in the query Q by AD relationships, we get a query Q'. We know that Q and Q' have the same output due to Lemma 2. The advanced preorder filtering is optimal for Q'; therefore, it is optimal for Q as well. □

We define the query optimality only for the T+LS and LPS streaming schemes, however, our optimality condition can be applied even to a new streaming scheme. We only have to specify the stream optimality satisfying Lemma 2.

Example 10 (**Stream Optimality Condition**) Consider the XML tree and the TPQ Q10 in Fig. 8. In the case of TwigStack, the PC relationships in the subquery $Q_{\#c}$ `//c[./a and ./f]` of the query Q10 cause the main problem. Let us have a configuration, where c_1, a_2, and f_1 are the head nodes of their streams $T_{\#c}$, $T_{\#a'}$, and $T_{\#f}$, respectively. The node a_2 is not in the query match of $\#c$ with the node c_1, but some nodes in the rest of the $T_{\#a'}$ stream can be in this query match. Furthermore, there is no query match of $\#c$ with c_1 and the nodes a_2 and f_1 cannot be skipped since we are not sure whether they are useless or not. To resolve this problem, c_1 is pushed onto the stack $S_{\#c}$ and the algorithm continues. However, c_1 is an irrelevant node as we can see in Fig. 2a.

In the case of LPS scheme, `checkingnodes`(Q10) = $\{\#a, \#c\}$, and, after the DataGuide search, we get $PRU_{\#a} = \{T_a\}$ and $PRU_{\#c} = \{T_{a/c/c}\}$. Here, all checking query nodes are optimal, therefore, a holistic algorithm using the LPS scheme is optimal for this query. Note that the query does not belong even to the most general iTwigJoin+PPS query class [9]. Consequently, the above described TwigStack issue does not occur during the query processing with an arbitrary LPS holistic algorithm. This is caused by the fact that the stream $T_{a/c}$ with the node c_1 is pruned during the DataGuide search.

7.2 XML document and algorithm optimality

There are characteristics of XML documents which lead to a situation that a holistic algorithm is optimal for any TPQ (or at least for a set of TPQs). In such case, we say that a holistic algorithm is optimal for the XML document. To our best knowledge, this article is the first one defining a holistic algorithm optimality for an XML document type.

Let us introduce a basic definition concerning the stream optimality of tree tags:

Definition 3 (*Optimal Tree Tag*)

- In the case of the T+LS scheme, a tree tag is called optimal if it is a single level tag or if it corresponds only to a leaf node.
- In the case of the LPS scheme, a tree tag tag_X is called optimal if every tag_X non-leaf node never has a tag_X node as an ancestor.

7.2.1 Optimality for XML document

Lemma 3 *Consider a TPQ* Q*. Then, for every* $\#q \in Q$ *with an optimal tree tag* $tag_{\#q}$*, it holds that* $\#q$ *is optimal.*

Proof The lemma follows easily from Definitions 1 and 3. □

Theorem 2 (Optimal Set of Tree Tags) *A holistic algorithm under an XS streaming scheme is optimal for any TPQ* Q*, for which it holds that every tag of* $\#q \in$ `checkingnodes`(Q) *corresponds to an optimal tree tag under XS.*

Proof Since every query node $\#q \in$ checkingnodes(Q) has the optimal tree tag under *XS*, every $\#q$ is optimal (see Lemma 3). Therefore, the stream optimality condition is satisfied and, by Theorem 1, a holistic algorithm using *XS* is optimal. □

We say that a holistic algorithm is optimal under an *XS* streaming scheme for an XML document if the XML document contains only optimal tree tags under *XS*. In such case, we guarantee the holistic algorithm optimality under *XS* for any TPQ and in the case of GTPStack for any GTP.

Example 11 (**Optimal XML Document**) The XML document in Fig. 9 serves as an example of an XML document

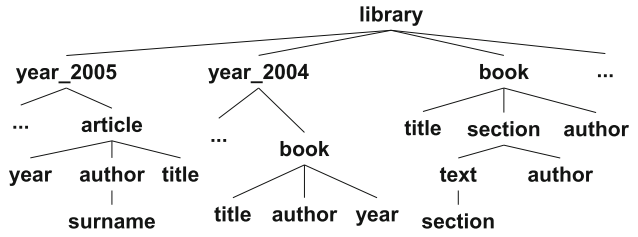


Fig. 9 Example of an optimal XML document under the LPS scheme

containing only optimal tree tags under the LPS scheme. Note that the definition of an optimal tree tag is specified only for inner nodes, and thus, the tree tag `section` is optimal.

Considering T+LS, the set of optimal tags is as follows: `year_2005`, `year_2004`, `article`, `year`, `surname`, `title`, and `text`. The tree tag `title` corresponds only to the leaf nodes; therefore, it is optimal under T+LS, even though it is not a single level tag.

7.3 Summary of optimality

Let us summarize the novel optimality conditions introduced in this section. Having an XML document D and a TPQ Q, a holistic algorithm is optimal for Q under XS if at least one of the following conditions is satisfied:

- The holistic algorithm is optimal for D under XS.
- Every tag of $\#q \in$ `checkingnodes`(Q) is an optimal tree tag under XS (Theorem 2).
- For every $\#q \in$ `checkingnodes`(Q) it holds that every stream T of class $\#q$ is optimal under XS after the DataGuide search (Theorem 1).

Note that the first condition implicates the second and the second condition implicates the third. All conditions are sufficient but not necessary.

8 Experimental results

We implemented five state-of-the-art holistic algorithms in C++: TwigStack [6], TwigList [33], TJStrictPre [15], TJStrictPost [15], and Twig^{2}Stack+PathStack [8] (abbreviated to T2PS). We do not include experimental results of the TwigList algorithm since both TJStrictPost and TJStrictPre use an improved version of TwigList. In our experiments, we use more than one version of GTPStack. We combine GTPStack with existing advanced preorder filtering functions; therefore, we use the following simple notation, where GTPStack+N, GTPStack+P, and GTPStack+M stand for GTPStack combined with `getNext`, `getPart`, and `getMatch`, respectively. By writing GTPStack we mean any of the above versions of GTPStack. Since GTPStack+N always outperforms TwigStack, we include the results for the TwigStack query processing only in Sect. 8.1.1. The main shortcoming of TwigStack is represented by its redundant intermediate storage and inefficient output enumeration.

Table 4 ZIPF query templates for XPath queries

	Query template	Number of generated queries
1.	$//\tau[/\upsilon$ and $/\omega]$	147
2.	$//\alpha/\beta[//\chi$ and $//\delta]$	26
3.	$//\alpha/\alpha[//\beta]/\chi//\chi[//\delta$ and $//\epsilon]$	82
4.	$//\alpha[/\tau$ and $//\upsilon$ and $//\omega]$	105
5.	$//\alpha/\beta[//\chi/\delta]$	81

We use one own synthetic XML document called ZIPF and three real-world XML collections. The ZIPF document contains seven different elements named from `a` to `g` spread randomly using the Zipfian distribution, where `a` has the highest occurrence ($\approx 50\,\%$) and `g` has the lowest occurrence ($\approx 1\,\%$). Every element of ZIPF has exactly two children and the depth of the collection is 24 which means that all paths in ZIPF have the same length. The real-world collections are XMark [34] with factor 10, INEX 1.9 [13], and TreeBank [36]. Note that the schema of every collection is significantly different. Whereas the XMark collection is shallow and data oriented, the INEX collection is document oriented and TreeBank has a very recursive structure with many different labeled paths. Basic statistics of these collections are shown in Table 9.

Queries for the XMark and TreeBank collections are selected from several existing articles on TPQ processing [8,24,22]. Queries for the INEX collection were selected in order to show differences between the algorithms. A list of the selected real-world collections' queries can be found in "Appendix 5".

The largest number of queries were generated for the ZIPF collection. The ZIPF queries are generated according to five query templates shown in Table 4. A template only specifies relationships between query nodes, output query node, and predicate query nodes. For each template, we generated all XPath queries such that $\alpha, \beta, \chi, \delta \in$ {`a,d,g`} and $\tau, \upsilon, \omega \in$ {`a,b,c,d,e,f,g`}. Table 4 gives the number of XPath queries corresponding to each template.

We run our experiments on a PC with Intel Xeon 2.93 GHz CPU, and Windows Server 2008 operating system.

When measuring the processing time, each query is processed fifteen times in the main memory, and then, we compute the average result omitting the two worst and the two best results. The raw data of query processing time for the real-world collections can be found in "Appendix 5". If we want to compare processing times T_x and T_y of two approaches x and y for a set of queries, we first compute a

Table 5 GTPStack+M compared to all tested approaches for all queries

Filtering approach	RPTI (%)	RFTI (%)	Number of queries	
			Faster	Slower
T2PS	77	172	401	65
TJStrictPost	88	222	461	9
TJStrictPre	12	43	279	0
GTPStack+N	68	150	340	73
GTPStack+P	13	46	346	0

geometric mean of ratios T_y / T_x of each query. Subsequently, since we want to have the value in percents, we simply subtract one from the calculated geometric mean and multiply it by 100. We call it a *relative processing time improvement (RPTI) of approach x compared to approach y for a set of queries*. For example, if we have the result of the geometric mean 1.68, we write that RPTI of x has a 68 % improvement compared to y.

Since GTPStack's improvements of processing time relate only to the filtering part of holistic algorithms, we also measured the filtering time of the algorithms (i.e., the processing time without the time spent on reading the input data), and therefore, we also compute the *relative filtering time improvement (RFTI) of approach x compared to approach y for a set of queries*. In order to minimize the processing time measurement error, we say that a method is faster than the other one for a query Q if its RPTI for Q is at least 2 % and their processing time difference for Q is at least 10 ms.

8.1 Processing time

Table 5 gives RPTI and RFTI of GTPStack+M compared to all tested approaches for all queries. Table 5 also contains the number of queries for which GTPStack+M is faster and slower.

Clearly, GTPStack+M outperforms both approaches TJStrictPre and GTPStack+P using the `getPart` function for all queries since the `getMatch` function is always faster or equally fast compared to `getPart`. This comes from the fact that `getMatch` improves `getPart` without any additional overhead. The remaining approaches (T2PS, TJStrictPost, and GTPStack+N) perform significantly worse in average than GTPStack+M (RFTI of GTPStack+M ranges from 150 to 222 % when compared to these approaches).

To better understand the differences among the algorithms and the advantages of GTPStack+M, we need to compare their corresponding parts separately. In Sect. 8.1.1, we compare only the result enumeration time; in Sect. 8.1.2, we compare the preorder filterings; in Sect. 8.1.3, we show how GTPStack optimizes its processing time for queries with many predicate nodes; and, in Sect. 8.1.4, we compare the postorder filterings.

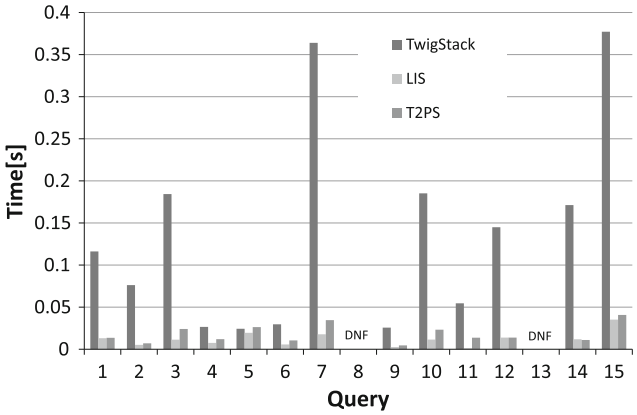


Fig. 10 Performance of the output enumeration

8.1.1 Test of intermediate storages for TPQs

Let us start with a test showing the properties of various intermediate storages and mainly the performance of the output enumeration. Note that the LIS intermediate storage is used by the TJStrictPost, TJStrictPre, and GTPStack algorithms. Results in this section serves as a hint for a selection of the most appropriate intermediate storage for our approach.

In this test, we present only the XMark queries since the results for other collections are similar. However, we ignore the GTP semantics (i.e., we consider all query nodes as output ones) since TwigStack cannot enumerate GTPs. As a result, queries 8 and 13 did not finish since their TPQ result sizes were over one billion and the available main memory was not sufficient.

Figure 10 shows the results of this experiment. As expected, TwigStack performs very poorly which corresponds to the results published in [8]. Inefficiency of the TwigStack intermediate storage comes from the duplicate work with nodes and the sequential scan of the whole intermediate storage during the output enumeration. If we compare the output enumeration times of T2PS and LIS, the difference is not significant. This result comes from the fact that they use the postorder filtering; therefore, their output enumeration time is linear with respect to the result size.

Finally, we decided to use the LIS storage since the Twig^{2}Stack intermediate storage requires global pop order for its correct functionality. LIS can work with our postorder filtering and its enumeration time performance is comparable to the Twig^{2}Stack intermediate storage.

8.1.2 Analysis of preorder filtering

We can find three major types of preorder filtering approaches: (1) the PathStack algorithm which filters only according to the node query path from the root query node and does not perform any cursor forward movement, (2) the `getNext`

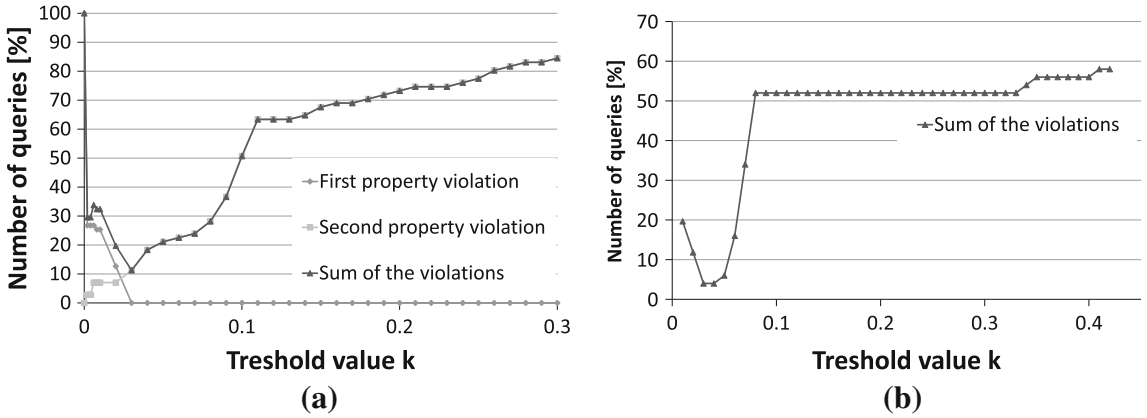


Fig. 11 Number of queries violating the properties for different k values **a** for all ZIPF queries **b** for queries corresponding to the first and to the fourth ZIPF template

function which performs a cursor forward movement according to the query node descendants (Line 15 in `getMatch` in Algorithm 2), and (3) `getPart` and `getMatch` which perform a cursor forward movement according to the query node descendants and ancestor (Line 2 in `getMatch` in Algorithm 2). We ignore `getPart` in the following text since `getMatch` always outperforms this function for our queries as is shown in Sect. 8.1.

The PathStack algorithm is very simple and fast and its processing time is linear with respect to the input size (the correlation coefficient between the PathStack processing time and the input size for the ZIPF queries is 0.98). Since PathStack does not use any advanced cursor forwarding, its processing time is not influenced by the query result. On the other hand, an advanced preorder filtering function (i.e., `getNext` or `getMatch`) can skip irrelevant nodes more quickly using the cursor forward movement. Therefore, these functions outperform PathStack if they *forward the cursor sufficiently often*; this is discussed further in this section.

The main attribute indicating the efficiency of an advanced preorder filtering is the *average cursor forward movement* (denoted as `AvgFwd`) during one `FwdToAncOf` or `FwdToDescOf` function call. Since the `getMatch` function uses both forwarding functions while `getNext` uses only the `FwdToDescOf` function, we also define: (1) an average cursor forward movement during one `FwdToAncOf` function call, and (2) an average cursor forward during one `FwdToDescOf` function call. Let us call them an *average ancestor forward movement* and an *average descendant forward movement* and denote them `AvgAncFwd` and `AvgDescFwd`, respectively.

We first compare the `getNext` and `getMatch` functions (i.e., we compare GTPStack+N and GTPStack+M). The `getNext` function uses only `FwdToAncOf`, and therefore, `getMatch` performs better if its `AvgDescFwd` is sufficiently large. Our goal is to find a threshold value k with the following two properties:

- if `AvgDescFwd` $> k$, then $T_{getNext} > T_{getMatch}$,
- if `AvgDescFwd` $< k$, then $T_{getNext} < T_{getMatch}$.

Figure 11a shows how many queries violate the above properties for all queries in our ZIPF query set for a different k value. As we can see, there is no optimal k value for which all queries satisfy both properties since the sum of violations never reaches zero. In other words, we cannot find any exact threshold value of `AvgDescFwd` which would say whether to use `getNext` or `getMatch`. The reason for this is that various query nodes in a query can have significantly different `AvgDescFwd` values; therefore, selection of the same advanced preorder filtering function for all query nodes is not always the best solution. Figure 11b shows the number of violations for the ZIPF queries generated by templates 1 and 4. These queries are very simple and the descendant forwarding is always performed in the relation to the same parent query node. We can observe that the threshold value $k = 0.03$ is a good selection for many of them since the sum of violations is less than 5 %. Based on the above heuristics, we evaluated a simple combination of the `getMatch` and the `getNext` functions which works as follows:

- We first process a query using `getMatch` and collect the statistics about `AvgDescFwd` in each query node.
- Secondly, we use the `getMatch` function for a query node with `AvgDescFwd` larger than 0.03 and use the `getNext` function in the other cases.

This combination of `getMatch` and `getNext` always performs better than or equally to any other advanced preorder filtering function.

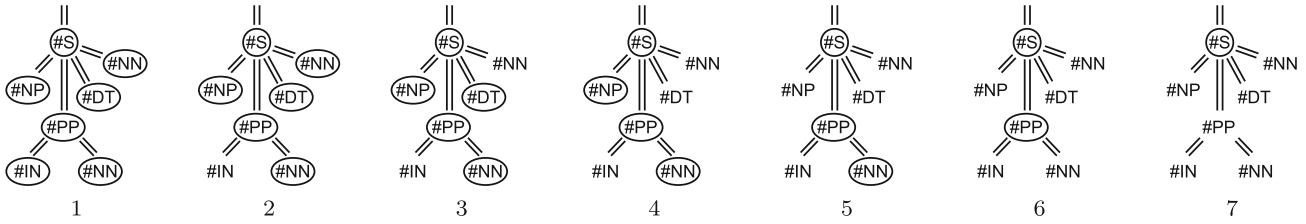


Fig. 12 Variants of the TB12 query with different number of output nodes

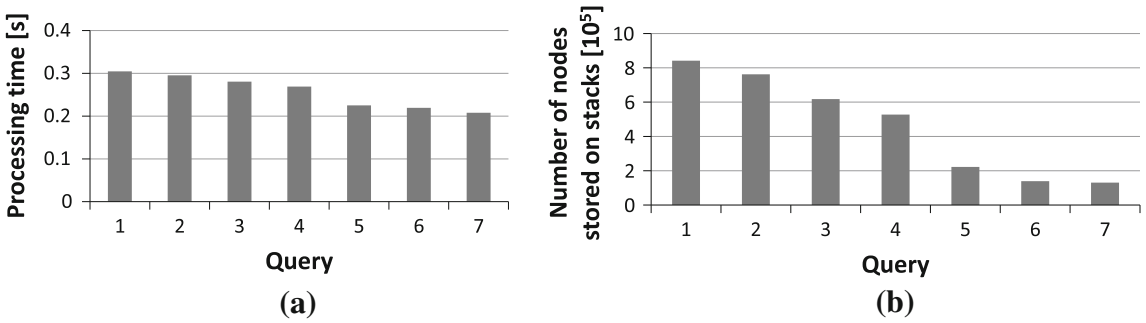


Fig. 13 **a** Processing time of GTPStack+M and **b** number of nodes stored on stacks by GTPStack for each variant of TB12

Similarly, we looked for another two threshold values which would indicate that an algorithm using PathStack performs better than algorithms using `getNext` or `getMatch`. Both algorithms have the threshold value of `AvgFwd` approximately equal to 0.3, where only 5% of the ZIPF queries corresponding to the first and fourth template violate the corresponding processing time properties. In other words, if `AvgFwd` is lower than 0.3, then PathStack performs better in many cases.

8.1.3 Optimization of predicate query nodes

None of the existing holistic algorithms can optimize the query processing time with respect to the number of nodes corresponding to predicate query nodes; therefore, their processing time is the same regardless of the GTP semantics. Their performance is mainly dependent on their preorder filtering as is shown in Sect. 8.1.2. If GTPStack is optimal, then it stores only the nodes corresponding to main branch query nodes on stacks, and thus, it saves some time during the postorder filtering.

In Fig. 12, we can observe seven variants of the TB12 query with a different number of output nodes. We selected the TB12 query because GTPStack+M is optimal for it.

Figure 13a shows how the processing time of GTPStack+M decreases with the decreasing number of nodes stored on stacks. The `AvgDescFwd` and `AvgAncFwd` values are equal to 0.09 and 0.14, respectively, for the TB12 query which indicates (according to the results of Sect. 8.1.2) that `getMatch` should be slower than T2PS and TJStrictPost having the processing times 0.219 and 0.202, respectively. However, for the last query variant, where the number of nodes corresponding to predicate query nodes is six times larger than the number of the other nodes, GTPStack+M performs equally to both algorithms (its processing time is 0.216).

8.1.4 Test of postorder filtering

In this section, we compare three basic postorder filtering techniques: (1) those used by T2PS, (2) those used by TwigList, TJStrictPost, and TJStrictPre, and (3) those used by GTPStack. The purpose of GTPStack's postorder filtering is not to speed up the query processing, but to enable the combined filtering before storing a node in the intermediate storage. Therefore, by this experiment we want to show that GTPStack's postorder filtering does not bring any overhead. An interesting result of this comparison is that T2PS and TJStrictPost perform very similar. Note that the TJStrictPost algorithm is an improved version of TwigList; therefore, our results are in the contrast with the results presented in [33], where TwigList significantly outperforms the Twig^{2}Stack algorithm. In order to get meaningful results, we have to compare algorithms with the same preorder filtering. Therefore, we compared T2PS with TJStrictPost and TJStrictPre with GTPStack+P.

Table 6 shows the summary of the results for the real-world collections. T2PS performs better than TJStrictPost if

Table 6 Relative processing/filtering time improvements of different postorder filterings for the real-world collections

Filtering approach	RPTI (%)	RFTI (%)	Number of queries	
			Faster	Slower
T2PS versus TJStrictPost	0	1	12	4
GTPStack+P versus TJStrictPre	0	0	6	5

Table 7 Relative Ratio of the number of nodes stored in various intermediate storages and the number of relevant nodes for each collection

Method	ZIPF	TB	XM	INEX
T2PS, TJStrictPost	50	527	8.1	90
TJStrictPre	27	163	3.7	10.5
GTPStack	1.3	15	1.4	1.82

it does not have to merge a lot of descendants in its intermediate storage. Otherwise said, it usually performs better than TJStrictPost if a query does not contain any AD relationships. This issue can be observed on the queries TB12 and TB15. TJStrictPost outperforms T2PS for TB12, however, when we replace most of the AD relationships by the PC relationships (the TB15 query), then T2PS outperforms TJStrictPost.

The postorder filtering times of TJStrictPre and GTPStack+P are comparable for three quarters of the real-world queries. The GTPStack+P postorder filtering usually performs better than TJStrictPre if we skip many predicate nodes during the query processing as is described in Sect. 8.1.3.

8.2 Intermediate storage size

Another important property of every algorithm related to the I/O complexity is the intermediate result size. We evaluate the intermediate result size in terms of the number of nodes stored there. For each method we compute a ratio of the number of nodes stored in the intermediate storage and the number of relevant nodes for each collection and filtering method. Table 7 shows this ratio for all queries in each collection computed using the geometric mean. Table 8 shows us the median value of nodes stored in an intermediate storage for each collection.

Generally, GTPStack stores one order of magnitude less nodes than the rest of the tested approaches due to the fact it uses the combined approach. The intermediate result size does not have a significant relationship to the processing time during the main memory run since every algorithm has to perform some extra operations if it wants to avoid storing useless nodes. However, the difference will be huge if an intermediate storage is larger than the main memory, and, in this case, I/O operations have to be included.

Table 8 Relative Median values of nodes stored in an intermediate storage for each collection

	ZIPF (10^3)	TB (10^3)	XM (10^3)	INEX (10^3)
T2PS, TJStrictPost	471	257	725	583
TJStrictPre	296	164	507	98
GTPStack	9	37	130	14
Relevant nodes	8	7	127	9

Table 9 Characteristics of the tested XML collections

Collection	Tag count	Labeled path count	Node count	Size (MB)
TreeBank	251	338,749	2,437,667	86
XMark	77	548	20,532,805	1,192
INEX-wiki	3,608	114,870	98,992,060	4,706
INEX-1.9	217	16,018	16,104,992	945
DBLP	41	170	14,971,785	535
Nasa	69	110	530,528	25
SwissProt	99	264	5,166,890	114
Protein	70	97	22,358,588	716

8.3 Test of optimality of XML documents

In this test, we show that the number of queries satisfying the optimality condition is not negligible. We analyzed several XML documents with different properties and computed the number of optimal tree tags under T+LS and LPS. A large number of optimal tree tags implies a better chance that an advanced preorder filtering algorithm is optimal for a TPQ according to Theorem 2.

In Table 9, we see basic characteristics of the XML collections used in our experiments. The TreeBank [36] collection includes over 300 thousand different labeled paths and the average depth of the XML documents is quite high. On the other hand, the XMark [34] collection includes only 548 labeled paths. We also analyzed DBLP [29], INEX-1.9, and INEX-wiki released by the INEX initiative [13], and also three collections in XML Data Repository [36] (Nasa, SwissProt, and Protein Sequence Database).

8.3.1 Statistics for T+LS and LPS

In Table 10, we see the number of optimal tree tags in each XML collection. Optimality under T+LS of the tested XML documents is high in the cases of the DBLP, Nasa, SwissProt, and Protein collections (the ratio of optimal tree tags exceeds 80 %). On the other hand, the INEX-1.9 and TreeBank collections have quite irregular structure; therefore, their ratio of optimal tags is low.

Under LPS, all collections have more than 80 % of tags optimal. The Nasa, SwissProt, and Protein collections are

Table 10 Number of optimal tree tags

Collection	T+LS		LPS	
	Count	Ratio (%)	Count	Ratio (%)
TreeBank	84	33.47	234	93.33
XMark	60	77.92	75	97.40
INEX-wiki	2,712	75.17	3,435	95.21
INEX-1.9	42	19.35	190	87.56
DBLP	38	92.68	39	97.56
Nasa	60	87.0	69	100
SwissProt	98	98.99	99	100
Protein	57	81.43	70	100

even optimal XML documents under LPS. The XMark collection has only two tree tags (*parlist* and *listitem*) which are not optimal. Similarly, the DBLP collection has only one such tree tag. The INEX-wiki collection includes a very small number of non-optimal tree tags. TreeBank includes only seventeen tree tags which are not optimal and we see that even an XML document with such a recursive structure has many optimal tree tags which can form optimal TPQs. The worst result from the labeled path streaming point of view occurs in the case of the INEX-1.9 collection, however, the number of optimal tree tags is still large.

The above statistics illustrate that holistic algorithms are often optimal for XML documents under LPS according to Theorem 2. Let us point out that the optimality determined by Theorem 2 is only a sufficient condition. Evidently, there are many queries that do not satisfy the assumptions of Theorem 2, but a holistic algorithm is optimal for them according to Theorem 1.

9 Conclusion

In this article, we introduce the first algorithm capable of processing GTPs optimally, which means that only the nodes relevant to the GTP result are stored in the intermediate storage. It is particularly important since GTP includes semantics of XQuery (in the contrast to well researched TPQ model). This is achieved by using the first correct combination of the preorder and postorder filterings before storing nodes in an intermediate storage. This new algorithm is called GTPStack. We also introduce other novel techniques related to the GTPStack algorithm. GTPStack significantly reduces the intermediate storage size (even if the algorithm is not optimal) by one order of magnitude as well as minimizing the number of nodes stored on stacks. We present thorough experiments for four common XML collections and 484 queries and show the conditions under which GTPStack outperforms other state-of-the-art holistic methods.

GTPStack utilizes the new preorder filtering function `getMatch`, which removes unnecessary computations of existing advanced preorder filtering functions. In our experiments, we show that the relative improvement of the `getMatch` filtering time is 43–222 % in average when compared to the other up-to-date methods. Moreover, we put forward that our new method outperforms other existing methods for the most test queries (from 84 to 100 %).

A further contribution of this article is an introduction of a new perspective of holistic algorithm optimality. We show that the optimality depends not only on a query class but also on XML document characteristics. We show that the holistic algorithm can be optimal for any GTP for a document without recursive nodes. In our experiments, we put forward that the number of optimal tree tags in common XML collections is rather large up-to 100 %.

Acknowledgments This work is supported by the Grant of GACR No. GAP202/10/0573. Jiaheng Lu is partially supported by National Science Foundation in China (NO. 61170011).

Appendix 1: Logical expression in getMatch

In Algorithm 5, we show an extended version of the `fwdToAncOf` function for logical expressions with NOT operators. The function is based on the preorder filtering functions for Boolean expressions introduced in [7,18]. The algorithm uses a *logical tree* which is a rooted tree, where the leaf nodes are the query nodes and the non-leaf nodes are the bool nodes. A logical tree represents a logical expression corresponding to a query node $\#q$, where the logical variables of the expression are the child query nodes of $\#q$. The function `ltree(#q)` returns the root node of $\#q$'s logical tree. Our algorithm supposes that every NOT bool node has exactly one child and this child is a query node. We can easily rewrite every logical expression so that every NOT bool node has exactly one query node using DeMorgan's laws. The `isUseless` function, which represents the core functionality of `fwdToAncOf`, evaluates the logical tree and returns true if the head node of the current query node is useless.

Appendix 2: Node concatenation

For the purpose of the concatenation, we store a (prev:next) pair as a part of each stack item, where the prev and next items are pointers to nodes. If we consider a set R of nodes stored in an intermediate storage which correspond to $\#q$ and which are descendants of $n_{\#q}$, then when we pop $n_{\#q}$, the prev item of $n_{\#q}$ points to $n'_{\#q} \in R$ having the lowest document order among all nodes in R and the next item of $n_{\#q}$ points to $n''_{\#q} \in R$ having the highest document order among all nodes in R.

Each pair is set to (empty:empty) when a node is pushed onto a stack. When a node $n_{\#q}$ is popped out from a stack, then

Algorithm 5: Ancestor forward with the NOT bool node

```
Procedure: fwdToAncOf(QueryNode #q)
1  while ¬isEnd(#q) ∧ isUseless(ltree(#q), H(#q))
   do advance(#q);

Function : Bool isUseless(AbstractQueryNode #q, NodeLabel
           label)
1  if #q is query node then
2      return label.preceding(H(#q)) ∨ #q.isEnded;
3  if #q is bool node then
4      if #q is AND/OR bool node then
5          foreach #child ∈ mandatoryChildren(#q) do
6              isUseless(#child, label);
7          return true if any/all isUseless call returns true;
8      if #q is NOT bool node then
9          #child = child query node of #q ;
10         if isEnd(#child) ∨
           ¬isSubtreeOptimal(#child) then
11             return false;
12         return label.ancestor(H(#child));
```

we set up the (prev:next) pair of $n^{anc}_{\#q}$, where $n^{anc}_{\#q}$ is a top node of $S_{\#q}$. If $S_{\#q}$ is empty, then we set the (prev:next) pair corresponding directly to $S_{\#q}$. A procedure setting the (prev:next) pair values runs in a constant time since it accesses only the top item of $S_{\#q}$. The (prev:next) pairs provide enough information to set up lists' pointers in an intermediate storage; therefore, the nodes are sorted in preorder even though they are added in postorder.

Appendix 3: GTP enumeration

In Algorithm 6, we depict the GTP enumeration used by GTPStack. It is a slight modification of the TwigList enumeration, which works with the enumeration query nodes. *Enumeration query nodes* are those main branch query nodes which are a part of the intermediate storage as is described in Sect. 6.2.2. An *enumeration parent relationship* of an enumeration query node #q is AD if any relationship between #q and its enumeration parent query node is AD.

There are three pointers to the intermediate storage list corresponding to every #q: $start[\#q]$, $end[\#q]$, and $move[\#q]$. The $start[\#q]$ and $end[\#q]$ pointers specify an interval in the list and the $move[\#q]$ pointer is always within this interval.

Appendix 4: Early enumeration

The Twig^{2}Stack algorithm introduces an early enumeration algorithm, which starts when we pop the last node from a

Algorithm 6: The output enumeration

```
Procedure: Enumeration()
1  start[#enumRoot] = first item in the #enumRoot list;
2  end[#enumRoot] = last item in the #enumRoot list;
3  openList(#enumRoot);
4  #actual = traverse the GTP tree in postorder and find the first
   query node #q, where move[#q] ≠ end[#q];
5  while #actual ≠ NULL do
6      start[#actual] = move[#actual].nextListNode ;
7      openList(#child);
8      if ¬isEnumerationRoot(#actual) then
9          resetParents(enumerationParent(#actual),
           #actual);
10     #actual = traverse the GTP tree in postorder and find the first
       query node #q, where move[#q] ≠ end[#q];

Procedure: openList(#q)
1  move[#q] = start[#q];
2  foreach #child ∈ enumerationChildren(#q) do
3      start[#child] = move[#q].getSEPair(#child).start ;
4      end[#child] = move[#q].getSEPair(#child).end ;
5      openList(#child);

Procedure: resetParents(#q, #childStop)
1  foreach #child ∈ enumerationChildren(#q), where #q
   < #childStop do
2      openList(#child);
3  if ¬isEnumerationRoot(#q) then
4      resetParents(enumerationParent(#q), #q);
```

stack $S_{\#fbr}$ corresponding to the first branching node. In this way, we can significantly reduce the intermediate storage size. However, this approach has a limitation: if any *top stack* (i.e., any stack corresponding to a query node between the root and the first branching query node) contains more than one node, then the early enumeration outputs an unordered result. To detect this problem we keep a switch E for each top stack, which is set to true when the stack is empty and false when the stack contains more than one node. Note that if the top stack contains exactly one node, then E preserves the current value. The early enumeration can start only if all E switches corresponding to the output nodes are true.

Appendix 5: Real-world queries and processing time

A list of the selected real-world collections' queries together with some important statistics can be found in Tables 11, 12 and 13. The raw data of query processing times for the real-world collections can be seen in Tables 14, 15 and 16.

Table 11 The XMARK queries

	Query	Result size	Source
XM1	//item[//location]//name	217,500	–
XM2	//address[//street]//city	127,306	–
XM3	//item[//location]/description//keyword	273,340	Qin et al. [33]
XM4	//person[//address/zipcode]/profile/education	31,676	Qin et al. [33]
XM5	//item[//location and //mailbox/mail//emph]/description//keyword	173,832	Qin et al. [33]
XM6	//person[//address/zipcode and //id]/profile[//age]/education	15,998	Qin et al. [33]
XM7	//open_auction[//annotation[//person]//parlist]//bidder//increase	282,468	Qin et al. [33]
XM8	//site/open_auctions[//bidder/personref]//reserve	59,348	Chen et al. [8]
XM9	//people//person[//address/zipcode]/profile/education	31,676	Chen et al. [8]
XM10	//item[//location]/description//keyword	273,340	Chen et al. [8]
XM11	//person[//profile[//age and //interest and //education and //gender and //business] and //address]//emailaddress	24,420	Li et al. [22]
XM12	//person[//emailaddress and //homepage and //name] //address[//country]//city	63,904	Li et al. [22]
XM13	//site[//person[//homepage and //emailaddress] and //open_auction [//bidder and //reserve]]//closed_auction[//annotation]//price	97,500	Li et al. [22]
XM14	//closed_auction[//annotation[//description] and //price and //date and //buyer]//seller	97,500	Li et al. [22]
XM15	//open_auction[//bidder[//personref and //time and //date] and //quantity and //reserve]//current	291,140	Li et al. [22]

Table 12 The TreeBank queries

	Query	Result size	Source
TB1	//S/VP//PP[//NP/VBN]/IN	804	Qin et al. [33]
TB2	//S/VP/PP[//IN]/NP/VBN	158	Qin et al. [33]
TB3	//S/VP//PP[//NN and //NP[//CD]/VBN]/IN	63	Qin et al. [33]
TB4	//S[//VP and //NP]/VP/PP[//IN]/NP/VBN	158	Qin et al. [33]
TB5	//EMPTY[//VP/PP//NNP and //S[//PP//JJ]//VBN]//PP/NP//_NONE_	1,589	Qin et al. [33]
TB6	//S[//MD]//ADJ	8	Lu et al. [24]
TB7	//S/VP/PP[/NP/VBN]/IN	151	Lu et al. [24]
TB8	//VP[/DT]//PRP_DOLLAR_	3	Lu et al. [24]
TB9	//S[//VP/IN]//NP	10,675	Lu et al. [24]
TB10	//S[/JJ]/NP	5	Lu et al. [24]
TB11	//S//NP[//PP//TO and //VP//_NONE_]//JJ	2,775	Li et al. [22]
TB12	//S[//NP and //DT and //NN]//PP[//IN]//NN	83,194	Li et al. [22]
TB13	//S[/VP and //NN and /VBD]/NP[/IN]/DT	0	Li et al. [22]
TB14	//S[/NP and /NONE]/VP//PP[/IN]/DT	15	Li et al. [22]
TB15	//S[/NP and /DT and /NN]//PP[/IN]/NN	0	Li et al. [22]

Table 13 The INEX queries

	Query	Result size	Source
I1	//books//article[/fm/hdr//crt[/issn and //onm] and //sec/p]/doi	12,388	–
I2	//article/bdy/sec/ss1/ss2[//theorem//head]	451	–
I3	//books/journal[/article[//issn and //sec/st and /bm//bib/bibl] and //doi]/title	16,308	–
I4	//article[//bdy//fig and //bm//vt and//footnote//list//item]//hdr//ti	4676	–
I5	//bdy[//tbl]//sec[/st and /p]//fig/no	21,145	–
I6	//books//article[/fm/hdr//crt[/issn and //onm] and //sec/p]/doi	266,272	–
I7	//books/journal[/article[//issn and //sec/st and /bm//bib/bibl and /doi]]/title	10,264	–
I8	//sec[/ss1//tbl and //ss2[/ip1 and //tf] and /st]	527	–
I9	//article[//bdy//fig and //bm//vt and //hdr//ti]	8,990	–
I10	//article[//fm[/hdr and /au]]//sec/ss1[//theorem//head]	2,361	–
I11	//bdy[//tbl]//sec[//fig/no and /p]/st	4,578	–
I12	//sec[/p//it]/st	28,806	–
I13	//sec[/p[//it or //ref]]/st	48,390	–
I14	//sec[/ss1//tbl and //ss2[/ip1 and //tf]]/st	527	–

Table 14 Processing times for the XMARK queries

	XM1	XM2	XM3	XM4	XM5	XM6	XM7	XM8	XM9	XM10	XM11	XM12	XM13	XM14	XM15
T2PS	1.045	0.369	1.157	1.520	1.296	0.879	2.230	0.749	0.153	0.198	1.048	0.587	0.683	1.220	1.508
TJStrictPost	1.010	0.357	1.024	1.434	1.202	0.822	1.937	0.688	0.156	0.202	0.972	0.572	0.654	0.978	1.382
TJStrictPre	0.654	0.355	0.655	0.490	0.826	0.49	2.200	0.867	0.097	0.199	0.700	0.629	0.718	1.308	1.387
GTPStack+N	1.038	0.297	1.029	0.514	1.244	0.819	2.966	0.693	0.095	0.22	0.831	0.494	0.498	1.251	1.487
GTPStack+P	0.635	0.402	0.656	0.527	0.888	0.495	2.291	0.736	0.099	0.212	0.691	0.658	0.655	1.331	1.320
GTPStack+M	0.640	0.349	0.651	0.448	0.773	0.450	2.300	0.587	0.071	0.178	0.674	0.567	0.452	1.252	1.129

Table 15 Processing times for the TreeBank queries

	TB1	TB2	TB3	TB4	TB5	TB6	TB7	TB8	TB9	TB10	TB11	TB12	TB13	TB14	TB15
T2PS	0.125	0.109	0.163	0.219	0.206	0.019	0.105	0.028	0.112	0.063	0.145	0.219	0.160	0.194	0.161
TJStrictPost	0.125	0.128	0.160	0.216	0.200	0.019	0.123	0.033	0.113	0.086	0.140	0.202	0.182	0.218	0.194
TJStrictPre	0.050	0.049	0.051	0.094	0.077	0.005	0.048	0.015	0.171	0.085	0.044	0.322	0.134	0.271	0.286
GTPStack+N	0.057	0.054	0.094	0.172	0.127	0.006	0.056	0.019	0.162	0.077	0.063	0.243	0.151	0.253	0.243
GTPStack+P	0.049	0.047	0.050	0.088	0.077	0.004	0.047	0.014	0.183	0.085	0.043	0.280	0.128	0.262	0.279
GTPStack+M	0.043	0.040	0.047	0.069	0.061	0.005	0.041	0.014	0.159	0.084	0.039	0.216	0.097	0.175	0.216

Table 16 Processing times for the INEX queries

	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12	I13	I14
Twig2Stack	0.134	0.028	0.053	0.067	0.169	0.135	0.053	0.091	0.046	0.073	0.169	0.281	0.394	0.091
TJStrictPost	0.150	0.030	0.059	0.057	0.184	0.153	0.059	0.088	0.043	0.063	0.179	0.295	0.370	0.087
TJStrictPre	0.304	0.009	0.102	0.033	0.076	0.321	0.103	0.029	0.046	0.029	0.075	0.245	0.252	0.029
GTPStack+N	0.240	0.012	0.066	0.093	0.209	0.275	0.075	0.087	0.060	0.062	0.200	0.216	0.278	0.086
GTPStack+P	0.278	0.009	0.087	0.032	0.078	0.330	0.095	0.029	0.041	0.029	0.075	0.223	0.252	0.029
GTPStack+M	0.103	0.009	0.032	0.027	0.072	0.180	0.061	0.027	0.032	0.028	0.068	0.159	0.257	0.027

References

1. Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava, D., Wu, Y.: Structural joins: a primitive for efficient XML query pattern matching. In: Proceedings of ICDE 2002, pp. 141–152. IEEE CS (2002)
2. Bača, R., Krátký, M.: On the Efficiency of a prefix path holistic algorithm. In: Proceedings of Database and XML Technologies, XSym 2009, vol. LNCS 5679, pp. 25–32. Springer (2009)
3. Bača, R., Krátký, M., Snášel, V.: On the efficient search of an XML twig query in large dataGuide trees. In: Proceedings of the Twelfth International Database Engineering & Applications Symposium, IDEAS 2008, pp. 149–158. ACM Press (2008)
4. Bača, R., Walder, J., Pawlas, M., Krátký, M.: Benchmarking the compression of XML node streams. In: Database Systems for Advanced Applications: 15th International Conference, DASFAA 2010, International Workshops, vol. 6193, pp. 179–190. Springer (2010)
5. Brantner, M., Helmer, S., Kanne, C.-C., Moerkotte, G.: Full-fledged algebraic XPath processing in Natix. In: Proceedings of Data Engineering, 2005. ICDE 2005, pp. 705–716. IEEE (2005)
6. Bruno, N., Srivastava, D., Koudas, N.: Holistic twig joins: optimal XML pattern matching. In: Proceedings of ACM SIGMOD 2002, pp. 310–321. ACM Press (2002)
7. Che, D., Ling, T.W., Hou, W.-C.: Holistic boolean-twig pattern matching for efficient XML query processing. IEEE Trans. Knowl. Data Eng. **99**, 2008–2024
8. Chen, S., Li, H.-G., Tatemura, J., Hsiung, W.-P., Agrawal, D., Candan, K.S.: Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In: Proceedings of VLDB 2006, pp. 283–294 (2006)
9. Chen, T., Lu, J., Ling, T.W.: On boosting holism in XML twig pattern matching using structural indexing techniques. In: Proceedings of ACM SIGMOD 2005, pp. 455–466. ACM Press (2005)
10. Chen, Z., Jagadish, H.V., Lakshmanan, L.V.S., Paparizos, S.: From tree patterns to generalized tree patterns: on efficient evaluation of XQuery. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, pp. 237–248 (2003)
11. Cooper, B., Sample, N., Franklin, M.J., Hjaltason, G.R., Shadmon, M.: A fast index for semistructured data. In: Proceedings of VLDB 2001, pp. 341–350 (2001)
12. Dietz, P.F.: Maintaining order in a linked list. In: Proceedings of 14th annual ACM symposium on theory of computing (STOC 1982), pp. 122–127 (1982)
13. Fuhr, N., Gvert, N., Malik, S., Lalmas, M., Kazai, G.: INEX (2007) https://inex.mmci.uni-saarland.de/
14. Goldman, R., Widom, J.: DataGuides: enabling query formulation and optimization in semistructured databases. In: Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB 1997, pp. 436–445 (1997)
15. Grimsmo, N., Bjørklund, T.A., Hetland, M.L.: Fast optimal twig joins. In: Proceedings of the 36th International Conference on Very Large Data Bases, VLDB 2010, pp. 894–905. VLDB Endowment (2010)
16. Grust, T., van Keulen, M., Teubner, J.: Staircase join: teach a relational DBMS to watch its (Axis) steps. In: Proceedings of VLDB 2003, pp. 524–535 (2003)
17. Härder, T., Haustein, M., Mathis, C., Wagner, M.: Node labeling schemes for dynamic XML documents reconsidered. Data Knowl. Eng. **60**, 126–149 (2007)
18. Jiang, H., Lu, H., Wang, W.: Efficient processing of XML twig queries with OR-predicates. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of data, pp. 59–70. ACM New York (2004)
19. Kaushik, R., Bohannon, P., Naughton, J., Korth, H.: Covering indexes for branching path queries. In: Proceedings of ACM SIGMOD 2002, pp. 133–144. ACM Press (2002)
20. Krátký, M., Bača, R., Snášel, V.: On the efficient processing regular path expressions of an enormous volume of XML data. In: Proceedings of DEXA 2007, vol. 4653 of LNCS, pp. 1–12. Springer (2007)
21. Krátký, M., Pokorný, J., Snášel, V.: Implementation of XPath axes in the multi-dimensional approach to indexing XML data. In: Current Trends in Database Technology, EDBT 2004, vol. 3268 of LNCS. Springer (2004)
22. Li, G., Feng, J., Zhang, Y., Zhou, L.: Efficient holistic twig joins in leaf-to-root combining with root-to-leaf way. In: Proceedings of the 12th International Conference on Database systems for Advanced Applications, DASFAA '07, pp. 834–849. Springer (2007)
23. Li, J., Wang, J.: TwigBuffer: avoiding useless intermediate solutions completely in twig joins. In: The 13th International Conference on Database Systems for Advanced Applications, DASFAA 2008, vol. 4947, pp. 554–561. Springer (2008)
24. Lu, J., Chen, T., Ling, T.W.: Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In: Proceedings of ACM CIKM 2004, pp. 533–542. ACM Press (2004)
25. Lu, J., Ling, T.W., Bao, Z., Wang, C.: Extended XML tree pattern matching: theories and algorithms. IEEE Trans. Knowl. Data Eng. **23**, 402–416 (2011)
26. Lu, J., Ling, T.W., Chan, C.-Y., Chen, T.: From region encoding to extended Dewey: on efficient processing of XML twig pattern matching. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, pp. 193–204 (2005)
27. Lu, J., Ling, T.W., Yu, T., Li, C., Ni, W.: Efficient processing of ordered XML twig pattern. In: Proceedings of DEXA 2005, vol. 3588 of LNCS, pp. 300–309. Springer (2005)
28. Lu, J., Meng, X., Ling, T.W.: Indexing and querying XML using extended Dewey labeling scheme. Data Knowl. Eng. **70**(1), 35–59 (2011)
29. Ley, M.: The DBLP computer science bibliography, http://www.informatik.uni-trier.de/~ley/db/
30. Michiels, P., Mihaila, G., Siméon, J.: Put a tree pattern in your algebra. In: Proceedings of the 23th International Conference on Data Engineering, ICDE 2007, pp. 246–255 (2007)
31. Moro, M.M., Vagena, Z., Tsotras, V.J.: Tree-pattern queries on a lightweight XML processor. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, pp. 205–216 (2005)
32. Paparizos, S., Wu, Y., Lakshmanan, L.V.S., Jagadish, H.V.: Tree logical classes for efficient evaluation of XQuery. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of data, pp. 71–82. ACM (2004)
33. Qin, L., Yu, J.X., Ding, B.: TwigList: make twig pattern matching fast. In: The 12th International Conference on Database Systems for Advanced Applications, DASFAA 2007, vol. 4443 of LNCS, pp. 850–862. Springer (2007)
34. Schmidt, A.R. et al.: The XML benchmark. Technical Report INS-R0103, CWI, The Netherlands (April 2001), http://monetdb.cwi.nl/xml/
35. Tatarinov, I., Viglas, S.D., Beyer, K., Shanmugasundaram, J., Shekita, E., Zhang, C.: Storing and querying ordered XML using a relational database system. In: Proceedings of ACM SIGMOD 2002, pp. 204–215. New York, USA (2002)
36. University of Washington Database Group: The XML Data Repository http://www.cs.washington.edu/research/xmldatasets/ 2002
37. W3 Consortium: XQuery 1.0: An XML Query Language, W3C Working Draft, 12 November 2003, http://www.w3.org/TR/xquery/

38. Wang, H., Park, S., Fan, W., Yu, P.S.: ViST: a dynamic index method for querying XML data by tree structures. In: Proceedings of the ACM SIGMOD 2003, pp. 110–121. ACM Press (2003)
39. Weiner, A.M., Härder, T.: Using structural joins and holistic twig joins for native XML query optimization. In: Advances in Databases and Information Systems, vol. 5739 of LNCS, pp. 149–163. Springer, Berlin Heidelberg (2009)
40. Weiner, A.M., Härder, T.: An integrative approach to query optimization in native XML database management systems. In: Proceedings of the Fourteenth International Database Engineering & Applications Symposium, IDEAS '10, pp. 64–74. ACM, New York, NY, USA (2010)
41. Wu, H., Ling, T.W., Chen, B., Xu, L.: TwigTable: using semantics in XML twig pattern query processing. In: Journal on Data Semantics XV, vol. 6720 of LNCS, pp. 102–129. Springer, Berlin Heidelberg (2011)
42. Wu, H., Ling, T.W., Dobbie, G.: TP+Output: modeling complex output information in XML twig pattern query. In: Database and XML Technologies, pp. 128–143. Springer (2010)
43. Yang, B., Fontoura, M., Shekita, E., Rajagopalan, S., Beyer, K.: Virtual cursors for XML joins. In: Proceedings of the thirteenth ACM International Conference on Information and Knowledge Management, CIKM 2004, pp. 523–532. ACM (2004)
44. Yoshikawa, M., Amagasa, T., Shimura, T., Uemura, S.: XRel: a path-based approach to storage and retrieval of XML documents using relational databases. ACM Trans. Internet Technol **1**(1), 110–141 (2001)
45. Yu, T., Ling, T.W., Lu, J.: TwigStackList¬: a holistic twig join algorithm for twig query with not-predicates on XML data. In: The 11th International Conference on Database Systems for Advanced Applications, DASFAA 2006, vol. 3882, pp. 249–263. Springer (2006)
46. Zhang, C., Naughton, J., DeWitt, D., Luo, Q., Lohman, G.: On supporting containment queries in relational database management systems. In: Proceedings of ACM SIGMOD 2001, pp. 425–436 (2001)