The Optimality of Holistic Algorithms for XPath

Byron Choi¹, Malika Mahoui¹, and Derick Wood²

¹ University of Pennsylvania
{kchoi, mmahoui}@seas.upenn.edu
² HKUST
dwood@cs.ust.hk

Abstract. Streaming XML, the de-facto standard for electronic data exchange, has quickly gained its practical importance. Holistic algorithms on twig queries have been shown efficient on processing streams of XML documents. We study some theoretical issues, mainly the optimality of holistic algorithms for evaluating queries with any kind of XPath axes. We characterize that such algorithms exist only when the query does not contain any of child, parent, following, following-sibling, preceding, and preceding-sibling axes, regardless the order of nodes in streams. For the XPath queries that optimal holistic algorithms exist, we propose the BagStack algorithm, an extension of the TwigStack algorithm, to find all solution. Since the BagStack algorithm assumes that nodes are sorted by their preorder number, we conclude that preordering is an appropriate order of nodes for streaming XML.

Keywords: Single-pass algorithms, streaming XML, data stream models and computations.

1 Introduction

Database research has recently shifted its focus from relational systems to systems facilitating data exchange on the Web [1]. Streaming [2] XML, the de-facto standard for electronic data exchange, has quickly gained its practical importance [14,12,13,11,9]. XPath [8] is a W3C recommendation on navigating XML documents. We study some theoretical properties of evaluating XPath on continuous streams of XML documents. We believe these properties are essential to elegant XML stream systems.

Application scenario. Consider a large number of users connecting to a mobile network via small devices such as mobile phones or PDAs. Web pages, stock market information, application related data are probably exchanged in XML format nowadays. This results in a large amount of XML streams on the wireless network. Although these devices often have relatively limited computing resources, one may want to perform simple query on the mobile device. For instance, one may want to display the IBM stock and the source of the quote is Reuters. This can be expressed in our syntax as //stock(/name/ibm, /source/routers).
The first question of such applications is that it is possible to collect the relevant part of XML document while it is streamed over the network. There are a few challenges. First, the data streaming context, it is not possible to cache the entire stream since one does not know the end of the stream. It is desirable to write a memory bound for processing the queries. Second, there are bursts of data in a practical network scenario. An XML stream algorithm must process each data item in the stream efficient and discard the irrelevant items as quickly as it can. It appears to be overkill relational joins on streams of XML.

This paper formalizes the above questions and provides some answers on streaming XML. The optimal way for processing a stream is to collect the relevant nodes as they are streamed in. We argue if this is possible for XPath queries. We also argue if XPath queries is computable with bounded memory.

Recently relational databases are extended to support efficient evaluation on XPath queries [3, 17, 16]. These systems typically decompose the queries into sub-queries, compute the intermediate result of the sub-queries and merge them at the end. For systems do not have knowledge on the size of the streams, large intermediate results may be stored during the computation even though the final result can be relatively small. In contrast, holistic algorithms for XPath evaluate the query as a whole. Such algorithms are useful in data streaming since irrelevant nodes are not kept in main memory during the evaluation.

The optimal holistic twig join algorithm called TwigStack was recently proposed by Bruno and his co-researchers [3]. It evaluates twig queries as a whole over streams of XML documents efficiently. Each node in the twig query is associated with a stream of nodes. The algorithm scans the streams only once and assigns constant memory only to the nodes that participate in at least one solution. Thus, the algorithm is optimal among all sequential algorithms that read the entire input. However, the algorithm is suboptimal when the twig queries contain child axes.

Recently, we [7] show that there is no strong optimal (see Definition 2) holistic algorithms for twig queries with child axis. The cause of this negative result is the existence of structural recursions in the document, which are common in practice [6]. In this paper, we perform an analysis on all XPath axes. We show the XPath fragment that cannot be evaluated optimally in Section 3. The memory requirement and the number of necessary scans of XPath are discussed. For the XPath fragment that can be evaluated optimally, we propose the DagStack algorithm (Section 2), which is based on the TwigStack algorithm [3], to find all solution.

Now, we describe the node representation and the assumptions on the streams of nodes that we use. We also briefly describe the syntax and the semantics of XPath. Finally, we describe the technical problem that we investigate.

XPath navigations are applied to the XML documents represent as follows. A document is modeled as a label tree. Nodes are represented by (1) the preorder number, (2) the postorder number and (3) the depth of the node. An example is to be found in Figure 1.

Assumptions. We assume that the preorder number, the postorder number,
the depth and the label are the only accessible information of a node. These assumptions imply the followings:

1. Given any two nodes, one can compute the ancestor-descendant and parent-child relationship of the two nodes in constant time;
2. one can compute the depth of a node in constant time;
3. one can compute the document order of the nodes in constant time.

**Definition 1.** [8] The syntax of an XPath navigation, Twig, is defined as follows in Backus-Naur Form:

- **Step**: ::= / | // | ↑ | ↓ | ≪ | ≫
- **Node Test**: ::= label
- **Path**: ::= Step Node Test | Step Node Test Path
- **Twig**: ::= Path | Path (Twig, Twig, ..., Twig)

An XPath navigation is given as a twig query in Definition 1. The steps (1) ‘/’, (2) ‘//’, (3) ‘↑’, (4) ‘↓’, (5) ‘≪’, (6) ‘≫’, (7) ‘≫≫’ denote advancing one step along the axis (1) child, (2) descendant, (3) parent, (4) ancestor, (5) following-sibling, (6) following, (7) preceding-sibling, and (8) preceding defined in the W3C recommendation on XPath [8]. We do not consider the self axis for simplicity. The query is called XPath navigations since predicates on data values are not considered. Predicates on data values are of practical importance but orthogonal to our analysis on the optimality. The solution of an XPath navigation is the set of all node combinations that satisfy the query.

Our computation model assumes that there is a stream of nodes associated with a node test in the XPath navigation such that the nodes satisfy the node test. A stream (denoted as T) is viewed as a pop-only stack. Since a stream contains only nodes with the same label, the partial ordering of two nodes in a stream is only defined if they have the same label. \( a_1 \prec a_2 \) is interpreted as node \( a_1 \) precedes node \( a_2 \) in the A-stream. For example, given the document and the query shown in Figure 1 and Figure 2, respectively. A, B, C, D, and D are associated with a stream of A, B, C, and D-nodes, denoted by \( T_A, T_B, T_C, T_D \), and \( T_D \) and defined by \( T_A = [a_1, a_2], T_B = [b_1, b_2], T_C = [c_1, c_2], T_D = [ ] \) and \( T_D = [ ] \). We will call the nodes that participate in at least one solution the useful nodes and the remaining nodes useless nodes.
Definition 2. An algorithm for XPath navigation is strongly optimal if and only if it returns the solution of a query by using: (1) a single forward scan of the streams, (2) constant memory for each useful node and (3) constant time processing each of the nodes in the streams.

Definition 3. An algorithm for a problem is optimal if and only if its time and space complexities meet the lower bounds of that problem.

Strong optimality is important for data streaming since one need to collect the useful items while receiving a stream without caching the entire stream.

The problem statement is given as follows. Given a XPath navigation, is it possible to design a strongly optimal algorithm for arbitrary streams?

2 The DagStack Algorithm

In this section, we show that there exists a strongly optimal holistic algorithm for the XPath//\-\% fragment by proposing the DagStack algorithm. The DagStack algorithm (1) incorporates the TwigStack algorithm with the “backward” axes elimination algorithm $\chi_{\alpha \alpha}$ [15] and (2) extends the TwigStack algorithm evaluating dag queries.

First, we illustrate that the evaluation of XPath navigation with other axes (e.g., XPath//\-\%\-\%) is not straightforward. This is due to the fact that the definition of the following nodes (similarly the preceding nodes) of a node $x$ excludes the descendant nodes (ancestor nodes) of $x$.

Proposition 1. If the nodes in streams are sorted by their preorder number, there is no strongly optimal holistic algorithm for XPath//\-\%\-\%.

Proof. Consider a document tree with 4 A-nodes where $a_1$ is an ancestor of $a_{11}$ and $a_{12}$; $a_{11}$ and $a_{12}$ are siblings; and $a_1$ and $a_2$ are siblings. The ordering of the nodes are $a_1 \prec a_{11} \prec a_{12} \prec a_2$. Consider the query $A_1 \prec \prec A_2$ (also $A_1 \prec A_2$). We cannot declare $a_1$ is useful until the stream $T_{a_1}$ is popped to $a_2$. This implies that $a_{12}$ is discarded.

Second, we illustrate the evaluation of “backward” axes in data streaming context by using the algorithm $\chi_{\alpha \alpha}$ [15]. The central of the algorithm is to translate the navigation to a directed acyclic query graph (a DAG query) in which contains “forward” axes only. For example, the query shown in the LHS of Figure 3 is translated to the query shown in the RHS of the figure. The $\chi_{\alpha \alpha}$ algorithm handles only child, descendant, parent and ancestor axes. It is observed that $\chi_{\alpha \alpha}$ can be naturally extended to handle the following-sibling, following, preceding-sibling and preceding axes. This observation is used in our proofs.

Denote also the translation to be $\chi_{\alpha \alpha}$. An evaluation of an XPath navigation $eval$ is a function from a query to its solution. By performing a simple induction on the translation rules, we obtain that $eval(q) \equiv eval(\chi_{\alpha \alpha}(q))$. 
Now, we extend theTwigStack algorithm to handle DAG queries. This leads to a holistic optimal algorithm for XPath navigation with descendant and ancestor axes (denoted as XPath//:/*) – the DagStack algorithm.

The pseudo-code of the DagStack algorithm is shown in Figure 4. We avoid the definition of auxiliary functions appearing in the pseudo-code of TwigStack if possible. However, the procedure getNext requires some explanations. The intuition of getNext is to pop the streams until the next possible solution is on the top of the streams. The nodes discarded in getNext are guaranteed to be useless (Lemma 4.2 of the TwigStack tech. report [4]). When q_{act} is returned by getNext, it guarantees that the top of stream of q_{act} descendants form a solution of the sub-query rooted at q_{act} (Lemma 4.1 of the tech. report [4]). Another fact is that TwigStack algorithm is sound and complete [3].

We use the translation χ_αΩ [15] on the query q (Line 01). In general, we obtain a dag g. Second, spanningTree(g) returns the spanning tree t of g (Line 02). We also obtain the set of edges E where t + E = g. The rest of the code are the same as the TwigStack except that Line 11-19 is added. It repeatedly calls getNext for locating the next solution (Line 04-05) for t. When Line 10 is reached, TwigStack guarantees the top of the streams of the descendants of q_{act} form a solution of t. Line 12-18 checks if the top of these streams satisfy all the edges in E. If it does, a solution of g is declared (Line 15-17). The stacks for storing the useful nodes are maintained as in TwigStack (Line 07-08, 20, 22-23).

Theorem 1. DagStack is sound and complete.

Proof. Given an XPath//:/* query q. Let g = χ_αΩ(q) and (t, E) = spanningTree(g).

Soundness: When Line 06 is reached, it implies that the top of streams and useful nodes found thus far form a new solution for t. If this solution also satisfies the edges E, it is a solution of g.

Completeness: Suppose s is a solution for g and hence t, s must be reported at Line 10 by the completeness of TwigStack algorithm. Since s is a solution of g, s must pass the tests on E in Line 11-18. Hence s is reported at Line 20. This implies that DagStack does not miss any solution.

Proposition 2. DagStack is strongly optimal.

This is established by the fact that DagStack performs constant amount more work than TwigStack does (which is essentially Line 11-18). DagStack also assigns memory to the useful nodes only. (Line 20)
Procedure DagStack(q)
01 q = χαοκ(q)
02 (t, E) = spanningTree(g)
03 // the extension TwigStack(t)
04 while ¬end(t)
05 qact = getNext(t)
06 if(¬isRoot(qact))
07 pop result stack of parent(qact)
08 until its top is an ancestor of top(Tqact)
09 if(isRoot(qact) \¬empty(result stack parent(qact)))
10 // a solution of t is found
11 dagSol = true
12 for (qz, a, qy) in E
13 if(qz, qy ∈ descendants of qact)
14 if(¬ a(top(Tqz), top(Tqy)))
15 dagSol = false
16 else if only qz ∈ descendants of qact
17 if(¬a(top of result stack of(qz), top(Tqz)))
18 dagSol = false
19 if dagSol
20 maintain intermediate result stack of qact
21 if(isLeaf(qact))
22 check if some intermediate result can be
23 removed from the result stacks
24 else advance(Tact)

Figure 4. The DagStack algorithm.

3 XPath Axes and the Optimality

In the last section, we show a strongly optimal algorithm for XPath/\,↑↑. In this section, we show that given the assumptions in Section 1, there is no strongly optimal holistic algorithm for an XPath navigation with any other axes.

Lemma 1. Given the assumptions in Section 1, there is no ordering of nodes such that the holistic evaluation for XPath/\,↑↑ is strongly optimal.

Proof. Suppose there exists such an evaluation eval for q ∈ XPath/\,↑↑. We can use the translation χαοκ to obtain a DAG d in which all parent axes are eliminated. By using the technique shown in Section 2, we can obtain an optimal evaluation eval> for the dag with child axes. It is known [7] that such evaluations for twig queries with child axes, and hence the dag queries, do not exist. Therefore, eval does not exist.

We use two propositions to establish similar result for XPath/\,↑↑,↑,↑↑.

Proposition 3. To satisfy the memory requirement of data streaming XPath/\,↑, and to allow one scan of the streams, without loss of generality, if a↓ < a↓ and
$a_i$ is not an ancestor of $a_j$ and vice versa, then for all pairs $(b_i, b_j)$, $b_i < b_j$, where $p$ in XPath$//_p$, $b_i \in a_i//p$ and $b_j \in a_j//p$, respectively [?].

**Proposition 4.** To satisfy the memory requirement of data streaming XPath$//_p$, $<\prec,\prec$, and to allow one scan of the streams, if $a_i$ and $a_j$ are siblings and $a_i$.preorder# < $a_j$.preorder#, then only either of the below properties holds for all streams.

1. $a_i \prec a_j$
2. $a_i \succ a_j$

**Proof.** Proof by induction on the number of A-children of a node. Consider a document rooted at $r$ with $n$ A-children and the query $A_1 \prec A_2$. The statement $\Phi(m, n)$ to be proved; given $a_1$.preorder# < $a_2$.preorder# < ... < $a_m$.preorder#, $m < n$, the ordering that can satisfies the memory requirement is either $a_1 \prec a_2 ... \prec a_m$ or $a_m \prec a_{m-1} ... \prec a_1$.

**Base Case** ($m = 3$): Assume that $a_2 \prec \{a_1, a_3\}$. At the beginning, the top of $T_{A_1}$ and $T_{A_2}$ are $a_2$. Popping $T_{A_1}$ will discard the solution $(a_2, a_3)$ while popping $T_{A_2}$ will discard the solution $(a_1, a_2)$. Hence, the only valid order is $a_1 \prec a_2 \prec a_3$ or $a_3 \prec a_2 \prec a_1$.

**Inductive step** ($m = k+1$): The induction hypothesis $\Phi(k, n)$ is true. Assume that $a_1 \prec a_2 ... \prec a_k, a_{k+1}$ must follow $a_k$ or at least a solution (e.g., $(a_k, a_{k+1})$) will be missed. Similarly, assume that $a_k \prec a_{k-1} ... \prec a_1, a_{k+1}$ must precede $a_k$ or at least a solution will be missed.

Hence $\Phi(m, n)$ is true for all $m < n$.

By using Proposition 3, if property (1) (property (2)) is true among the children of one node, property (1) (property (2)) holds for the entire document.

![Figure 5](image-url) One of the XML documents used in the proof of Lemma 2 and Lemma 3.

**Lemma 2.** Given the assumptions in Section 1, there is no ordering of nodes such that the holistic evaluation for XPath$//_p$ is strongly optimal.

**Proof.** Consider the document shown in Figure 5 except that only $a_1, a_{11}$ and $a_2$ are A-nodes while others are B-nodes and the query $A_1 \prec A_2$. Suppose $a_1$.preorder# < $a_2$.preorder#. Suppose property (1) of Proposition 4 holds for the document, i.e. $a_1 \prec a_2$. 


Case 1. $a_1 \prec a_{11} \Rightarrow a_2 \prec$ descendants of $a_2$ (Proposition 3) $\Rightarrow a_1 \prec$ descendants of $a_1$ (Proposition 3). There must not be useful $A_2$ nodes, and hence $A_1$ nodes (Proposition 4 property (1)), in between $a_1$ and $a_2$ or at least a solution is missed regardless of the order the streams are popped. $a_{11}$ is an useful node because of $a_{12}$. Consider the order $a_1 \prec a_2 \prec$ descendants of $a_1 \prec$ descendants of $a_2$ and the query $A_1 // A_2$. Consider a similar document in which there is a solution $(a_{x_1}, a_{y_1})$ in $a_1$ subtree. No strongly optimal evaluation can return all solution by using $a_{11} \prec a_2 \prec \{a_{x_2}, a_{y_2}\}$. We obtain a contradiction.

Case 2. $a_{11} \prec a_1 \Rightarrow$ descendants of $a_2 \prec a_2$ (Proposition 3). Since $a_2$ is the only node causing $a_1$ useful. There must not be any $A_2$ useful nodes, and hence $A_1$ useful nodes (Proposition 4 property (1)), in between $a_1$ and $a_2$ for the similar reason in the above case. Consider the order $a_{11} \prec$ descendants of $a_2 \prec a_1$ and the query $A_1 // A_2$. Consider a similar document in which there is a solution $(a_{x_1}, a_{y_1})$ in $a_2$ subtree. No strongly optimal evaluation can return all solution by using $a_{11} \prec \{a_{x_2}, a_{y_2}\} \prec a_1$.

Now, we perform a similar case analysis on assuming property (2) of Proposition 4 holds for the document. We established similar contradictions.

Lemma 3. Given the assumptions in Section 1, there is no ordering of nodes such that the holistic evaluation for XPath $\mathfrak{f}$ is strongly optimal.

Proof. The proof of this lemma is very similar to the proof of Lemma 2.

Consider the document as described in the proof of Lemma 2 and the query $A_1 \prec A_2$. Suppose $a_{11 \text{preorder#}} \prec a_{2 \text{preorder#}}$. Suppose property (1) of Proposition 4 holds for the document. $a_{11} \prec a_2$

Case 1. $a_{11} \prec a_{11}$. The analysis of this case is the same as Case 1 in the proof in Lemma 2, except that $a_{11}, a_{11}, a_{12}$ and $a_{2}$ are the only $A$-nodes while the others are altered to $B$ nodes for this case analysis.

Case 2. $a_{11} \prec a_1 \Rightarrow$ descendants of $a_1 \prec a_1$ (Proposition 3). Consider the query $A_3 // A_4$ and a similar document in which only $a_{11}, a_{11}, a_{12}, a_{12}$ and $a_{2}$ are $A$-nodes while others are $B$-nodes. The only node that makes $a_{11}$ useful (for $A_4$) is $a_{1}$. By using Proposition 4 property (1), we have $a_{11} \prec a_{12}$. Consider a similar document in which there is a $A$-descendant node of $a_{12}$ $a_{x}$. $a_{x}$ must precedes $a_{1}$. Neither $a_{11} \prec a_{12} \prec a_{x} \prec a_{1}$ nor $a_{11} \prec a_{x} \prec a_{12} \prec a_{1}$ supports strongly optimal evaluation for $A_3 // A_4$. This implies that descendants of $a_{12} \prec a_{11} \prec a_{12} \prec a_{1} \Rightarrow a_{12} \prec a_{11} \prec a_{12} \prec a_{1}$. Consider the query $A_3 \prec A_2$ again. $a_{12}$ cannot be declared useful (for $A_2$) until a node in $a_{2}$ subtree is encountered . The solution $a_{11}$ will not be reported.

Now, we perform a similar case analysis on assuming property (2) of Proposition 4 holds for the document. We established similar contradictions.

Lemma 4. Given the assumptions in Section 1, there is no ordering of nodes such that the holistic evaluation for XPath $\mathfrak{f}$ is strongly optimal.

Proof. Suppose there exists such a strongly optimal evaluation eval for $q \in$ XPath $\mathfrak{f}$. We can use the translation $\chi$ to obtain a DAG $d$ in which all preceding and preceding-sibling axes are eliminated. By using the technique
shown in Section 2, we can obtain an optimal evaluation \( \text{eval}^R \) for the dag with following and following-sibling axes. Lemma 2 and 3 show that such evaluations for twig queries with following and following-sibling axes, and hence the dag queries, do not exist. Therefore, \( \text{eval} \) does not exist.

**Theorem 2.** Given the assumptions in Section 1, there is no ordering of nodes such that the holistic evaluation for query beyond the XPath\( //\) fragment is strongly optimal.

**Proof.** By putting Lemma 1, 2, 3 and 4 and the result in [7] together.

### 3.1 Multiple Scans and Memory Requirement

Similar to twig queries with child axes [7], XPath navigations requires large number of scans if memory is assigned to useful nodes only.

**Proposition 5.** If memory is assigned to useful nodes only, the lower bound of the number of scans required by XPath navigation on arbitrary streams is exponential to the depth of the document.

However, when the memory requirement of the streaming evaluation is relaxed, a large fragment of XPath can be evaluated efficiently [10].

**Proposition 6.** XPath navigation is P-complete with respect to the combined complexity [10].

Proposition 6 indicates that the lower bound of the space complexity of the evaluation of XPath is in LOGSPACE. In contrast, we obtain that the lower bound of the space complexity of XPath\( //\) fragment is linear to the size of solution and the space complexity of any larger XPath fragment is higher than linear to the size of solution.

Since the size of streams is usually not known a priori, it is a desirable property if the space requirement of the query is not given by the size of input but some other parameters.

**Definition 4.** A query \( q \) is memory-bounded computable if and only if there is a constant \( M \) by which \( q \) can be evaluated upon arbitrary data streams.

Although the memory-bounded computability is an important property of data streaming, it is natural to see that streaming XML does not have this property.

**Proposition 7.** XPath navigations are not memory-bounded computable.
4 Conclusions and Future Work

We propose the DagStack algorithm for XPath//\n. We provide a characterization of XPath navigations. Our result is summarized in Table 4. We also present the lower bound of the number of scans if memory is assigned only to useful nodes and the memory requirement of XPath navigations.

This work shows that the preorder walk of the tree (the DagStack algorithm) can be used to evaluate exactly the XPath fragment for which optimal evaluations exist. On the contrary, for the XPath fragment could not be evaluated optimally by a preorder walk of the tree will not be evaluated optimally by any other walk. For the future work, we aim at studying the power of a preorder walk on XML trees with bounded memory.

Table 1. Table of Result

<table>
<thead>
<tr>
<th>Solved by DagStack</th>
<th>No strongly optimal holistic algo</th>
</tr>
</thead>
<tbody>
<tr>
<td>descendant</td>
<td>child</td>
</tr>
<tr>
<td>ancestor</td>
<td>parent</td>
</tr>
<tr>
<td></td>
<td>following</td>
</tr>
<tr>
<td></td>
<td>following-sibling</td>
</tr>
<tr>
<td></td>
<td>preceding</td>
</tr>
<tr>
<td></td>
<td>preceding-sibling</td>
</tr>
</tbody>
</table>

Acknowledgements

We thank Nicolas Bruno for discussions on the TwigStack algorithm. We are grateful to Susan Davidson, Yosub Han and Wilfred Ng for their insightful comments. The first and the last author were supported under a Research Grants Council Earmarked Research Grant. Byron Choi is currently visiting HKUST.

References