StreamTX: Extracting Tuples from Streaming XML Data* †

Wook-Shin Han[1]‡ Haifeng Jiang[2] Howard Ho[3] Quanzhong Li[4]

[1] *Department of Computer Engineering, Kyungpook National University, Republic of Korea*
[2] *Google Inc., Mountain View, California*
[3,4] *IBM Almaden Research Center, San Jose, California*
[1]*wshan@knu.ac.kr,* [2]*jianghf@google.com,* [3]*ho@almaden.ibm.com,* [4]*quanzhli@us.ibm.com*

ABSTRACT

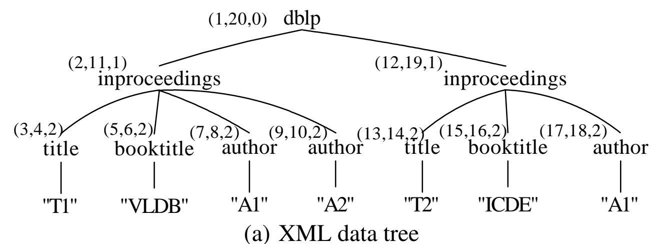
We study the problem of extracting flattened tuple data from streaming, hierarchical XML data. Tuple-extraction queries are essentially XML pattern queries with *multiple* extraction nodes. Their typical applications include mapping-based XML transformation and integrated (set-based) processing of XML and relational data. Holistic twig joins are known for the optimal matching of XML pattern queries on parsed/indexed XML data. Naïve application of the holistic twig joins to streaming XML data incurs unnecessary disk I/Os. We adapt the holistic twig joins for tuple-extraction queries on streaming XML with two novel features: first, we use the *block-and-trigger* technique to consume streaming XML data in a best-effort fashion without compromising the optimality of holistic matching; second, to reduce peak buffer sizes and overall running times, we apply *query-path pruning* and *existential-match pruning* techniques to aggressively filter irrelevant incoming data. We compare our solution with the direct competitor TurboXPath and other alternative approaches that use full-fledged query engines such as XQuery or XSLT engines for tuple extraction. The experiments using real-world XML data and queries demonstrated that our approach 1) outperformed its competitors by up to orders of magnitude, and 2) exhibited almost linear scalability. Our solution has been demonstrated extensively to IBM customers and will be included in customer engagement applications in healthcare.

1. INTRODUCTION

Querying XML streams has become a crucial problem in modern information systems. In contrast to XML that is parsed and stored in databases, streaming XML arrives in order (typically as a sequence of SAX events) and can be most efficiently processed by consuming such SAX events on-the-fly without extensive buffering.

We study efficient algorithms for tuple-extraction queries against streaming XML data. XML tuple-extraction queries are XML pattern queries with *multiple* extraction nodes. XML tuple-extraction queries can be expressed as XQuery queries containing both multiple projection nodes in the return clause and twig pattern matching expressions. The result of a tuple-extraction query is a stream of flat tuples.

As an example, Figure 1(b) shows a pattern that extracts tuples with two fields, `title` and `author`, from the DBLP data in Figure 1(a). The meaning of the triplets alongside tree nodes is explained in Section 2. The result of the tuple-extraction query is a set of three tuples as shown in Figure 1(c).



(a) XML data tree

/dblp
/inproceedings
/title# /author#

(b) Extraction pattern

	title	author
t_1	T1	A1
t_2	T1	A2
t_3	T2	A1

(c) Extracted tuples

Figure 1: Example of tuple extraction from XML data.

Tuple extraction from streaming XML data is an important operation in many crucial XML applications. For example, it is shown to be a core operation for data transformation in schema-mapping systems [13]. In such mapping-based XML transformation, we need to extract mapped values from streaming XML data sources. The extracted values are in the form of flat tuples, which are then transformed to the target based on a mapping specification. We note that tuple extraction has also been identified as a computationally expensive operation in the integrated processing of XML and relational data [15]. Specifically, a tuple stream can be extracted from an XML data source and then sent to a relational operator for further processing, such as joining with other relational tables.

Despite the significant amount of work on streaming XML processing, most of the work focused on XML filtering (such as XFil-

*The work was carried out while the first two authors were with IBM Almaden Research Center.

†Part of the work was supported by U.S. Air Force Office for Scientific Research Grant under contract FA9550-07-1-0223.

‡Wook-Shin Han was supported by the Korea Research Foundation Grant funded by the Korean Government(MOEHRD) (KRF-2007-521-D00399).

Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than VLDB Endowment must be honored.
Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212) 869-0481 or permissions@acm.org.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

ter [1]) or *single* extraction node [20] (not *multiple* extraction nodes). As a result, the work on efficient algorithms for tuple extraction is rather limited. TurboXPath [15] is the latest system specifically designed for tuple extraction from streaming XML data and has been incorporated in DB2 XML [2]. However it demonstrates exponential complexity when dealing with recursions. Additionally, although most XSLT/XQuery engines can support tuple-extraction queries, they do not provide satisfactory performance due to efficiency and scalability problems.

Our solution has been designed in the context of the Clio project [12] for mapping-based XML transformation, where XML files are read in a streaming fashion in an ETL environment. In this situation, to handle very large XML files, we need to provide robust and optimal worst-case performance with a fixed buffer size for any large XML files. This motivated us to adapt TwigStack (robust and disk-based implementation) for streaming processing.

1.1 Our Approach

Although holistic twig joins [4] provide optimal matching of XML pattern queries against XML data stored in databases, they are not directly applicable for processing streaming XML data. A naïve adaptation that parses the streaming data and then applies the holistic joins is neither efficient nor feasible for continuous XML streams.

In this paper, we adopt the paradigm of holistic matching and present the StreamTX algorithm, which can process streaming XML for tuple extraction progressively without extensive buffering. Specifically, we introduce a block-and-trigger mechanism during holistic matching so that incoming XML is consumed in a best-effort fashion without compromising the optimality of holistic matching. However, the blocking mechanism may cause some incoming data to be buffered. To reduce the buffer sizes, two pruning techniques are deployed. In particular, the query-path pruning guarantees that each buffered element satisfies its query path; the existential-match pruning guarantees that we only buffer elements that participate in final results.

Our contribution can be summarized as follows:

- We first review the existing TwigStack algorithm [4] for matching patterns on stored XML and then adapt it into the TwigStackTX algorithm, which can support tuple-extraction from streaming XML data.
- We next present a full-fledged algorithm, namely StreamTX, which marries the holistic-matching paradigm with the streaming XML processing and provides highly efficient tuple extraction with minimal buffer sizes.
- We conduct extensive experiments with StreamTX in comparison with other existing approaches, using real-world XML documents and queries. The experimental results show that StreamTX is significantly more efficient and scalable than its competitors. For many data and query sets, the performance advantage reaches orders of magnitude.

1.2 Paper Organization

We formally define the problem of tuple extraction in Section 2. We discuss the related work in Section 3. Section 4 goes over the existing paradigm of holistic matching and proposes its adaptation for handling streaming XML. In Section 5, we present our tuple-extraction algorithm, StreamTX, for streaming XML. Experiments are reported in Section 6. We conclude the paper in Section 7.

2. PROBLEM DEFINITION

2.1 XML data model and region encoding

We follow the XPath 2.0 specification [3] and model XML data as a tree, where nodes represent elements, attributes and text data, and parent-child pairs represent nestings between XML element nodes. Data tree nodes are often encoded with positional information for efficient evaluation of their positional relationships.

In Figure 1(a), the tree nodes are assigned with *region encoding* [9, 27, 24], which is a triplet $(start, end, level)$ representing the positional information of each element. The root is a `dblp` element spanning from position 1 to 20. The first `inproceedings` element spans from 2 to 11, and so on. The $level$ value records the distance from the element to the root element, whose $level$ value is zero.

The region encoding supports efficient evaluation of ancestor-descendant or parent-child relationship between element nodes. Formally, element u is an ancestor of element v if and only if $u.start < v.start < u.end$. For the parent-child relationship, we also test whether $u.level = v.level - 1$.

2.2 XML tuple-extraction queries

XML tuple-extraction queries are XML pattern queries with multiple extraction nodes. A tuple-extraction query can be represented as a labeled query tree with one or multiple extraction nodes. Query tree nodes are also called QNodes. For example, Figure 2 shows two tuple-extraction patterns.

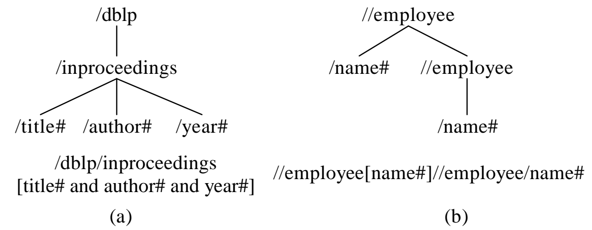


Figure 2: Example tuple-extraction patterns.

A single slash "/" stands for a parent-child relationship between the QNode and its parent, while a double slash "//" means an ancestor-descendant relationship. The symbol # indicates that the QNode is an atomic element node and its value is extracted. The pattern in Figure 2(a) extracts triplets of (`title`, `author`, `year`). The pattern in Figure 2(b) returns pairs of names of employees with a management relationship.

For easy reference, we represent tuple-extraction patterns with pseudo XPath queries with the symbol # for extraction nodes. In Figure 2, the pseudo XPath queries are listed beneath the corresponding patterns.

To define the answer to a tuple-extraction query, we first define the concept of a full match of a tuple-extraction query:

DEFINITION 1. ***(Full matches of tuple-extraction queries)*** *A full match of a tuple-extraction pattern Q in an XML database D (modelled as a tree) is identified by a mapping from nodes in Q to nodes in D, such that: (1) QNode predicates (if any) are satisfied by the corresponding database nodes; and (2) the structural (ancestor-descendant or parent-child) relationships between QNodes are satisfied by the corresponding database nodes.*

A full match can be represented as an n-ary relation where each tuple $(e_1, e_2, \cdots, e_n)$ consists of the database nodes. For the ex-

	DBLP	inproceedings	title	author
t_1	(1,20,0)	(2,11,1)	(3,4,2):T1	(7,8,2):A1
t_2	(1,20,0)	(2,11,1)	(3,4,2):T1	(9,10,2):A2
t_3	(1,20,0)	(12,19,1)	(13,14,2):T2	(17,18,2):A1

Table 1: Full matches of the query in Figure 1(b).

traction nodes in the pattern, the corresponding text values are associated with the matched element nodes. The *answer* to a tuple-extraction query is the set of full-match tuples projected on the extraction nodes.

EXAMPLE 1. *Given the XML data in Figure 1(a) and the tuple-extraction query in Figure 1(b), there are three full matches as shown in Table 1. Each element is identified with its region code. For the extraction nodes, their elements are also attached with text values. To obtain the answer from the full matches, we project the full-match tuples on the extraction-node columns. Figure 1(c) lists the final answer. The region codes are omitted after projection.*

3. RELATED WORK

Holistic XML matching algorithms are prevalent for matching pattern queries over stored XML data. They demonstrate good performance due to their ability to minimize unnecessary intermediate results.

In particular, Bruno et al. [4] proposed the first merge-based algorithm, which scans input data lists sequentially to match twig patterns. Such merge-based algorithms can be further improved by structure indexes that can reduce sizes of input lists [6]. Index-based holistic joins [14] were also proposed to speedup the matching of selective queries, as an improvement over merge-based algorithms.

In contrast, algorithms for streaming XML assume that XML documents are not parsed in advance and they come in the form of SAX events. Sometimes even ad-hoc XML documents can be regarded as streaming XML if using SAX parser is the best way to access them. We highlight below some existing work that supports tuple extraction from streaming/ad-hoc XML documents.

XSQ [20] is a system for querying streaming XML data using XPath. It uses pushdown transducers and can efficiently support features like multiple predicates, closures, and aggregation. TurboXPath [15] is the recent work on streaming tuple extraction, and has been incorporated in DB2 XML [2]. It has been shown to outperform XSQ for tuple extraction. We note that TurboXPath is optimized for handling plain XML data and can degrade significantly for recursive queries and data.

Besides the XQuery engines compared in our experiments, we have seen other new XQuery systems. BEA/SQRL [10] focuses on XQuery processing and optimization. The processing of each step in an XPath expression is based on the operator and iterative model. There is no special handling for data recursions, and intermediate results are materialized in worst cases. Flux [17] and a recent system [18] can optimize XQuery queries and translate them into other host languages such as Java. They use static analysis to optimize complex queries and buffer sizes, which is not sufficient to capture the whole picture of buffer minimization. Our work, on the other hand, focuses on the techniques of minimizing memory usage at run time for efficiently processing twig queries with multiple extraction points. The work in [7] deals with streaming matching of *XPath* queries with single extraction node. The proposed compact stack encoding is based on existing work. Twig^{2}Stack [5] can be viewed as an in-memory version of twig join algorithms. Although the performance is improved for many easy cases, the in-memory algorithm essentially trades the ability to handle skewed cases for efficiency for these easy cases.

To minimize the memory usage during XQuery execution, Schmidt et. al. [23, 16] proposed a buffer management scheme which statically rewrites a query and inserts "signOff" statements. When a node is cached by the buffer manager, one or more "roles" will be assigned based on matching path expressions. The inserted signOff statements signal the buffer manager at runtime that certain nodes lose roles. If all roles of a node are removed, a garbage collector will reclaim its memory. The role assignment is closely coupled with projection trees and lazy DFAs. Similar to TurboXPath, DFA based approaches may exhibit exponential complexity when dealing with recursion. In addition, this work does not directly support full XPath expressions with multiple extraction points. For example, a query as in Figure 1(b) needs to be expressed using the for-clause in XQuery for processing. However, a series of nested for-loops will be generated for the query, which limits the opportunities to optimize the query as can be exploited in the holistic approach presented in this paper. Furthermore, our work can greatly minimize memory usage for this type of query, and has optimal worst-case I/O and CPU cost.

4. TwigStackTX: THE FIRST ATTEMPT

Holistic twig joins were first proposed by Bruno et al. [4] as an optimal means for matching XML patterns over XML data stored in databases. In particular, their proposed algorithm, TwigStack, can avoid irrelevant intermediate results and achieve optimal worst-case I/O and CPU cost—that is, *linear* to the total size of input and output data.

In this section, we go over the TwigStack algorithm and then propose an adaptation algorithm, TwigStackTX, which can execute tuple-extraction queries for streaming XML data.

4.1 The data structures

We use the symbol q (with or without subscript) to refer to a QNode in a query tree. For example, q_a, q_b and q_c refer to the three QNodes in Figure 3(b). The function `isLeaf`(q) examines whether a QNode q is a leaf node or not. The function `children`(q) gets all child QNodes of q. For example, `children`(q_a) is the list $\{q_b, q_c\}$.

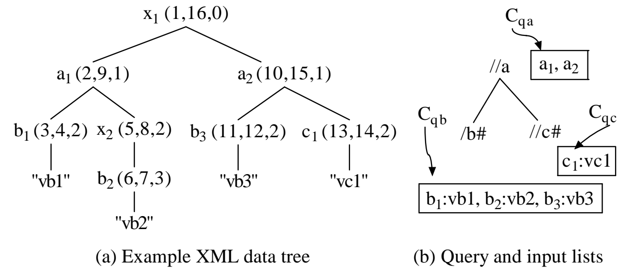


Figure 3: Example XML data and query with input lists.

There is an input element list associated with each node in the query tree. All elements in the lists are assigned with region codes and they are sorted according to their $start$ attributes in each list. Note that the elements for extraction QNodes (such as q_b and q_c) are also associated with text values.

We assume that, for each QNode q, there is a cursor, denoted as C_q. Each cursor C_q points to an element in the corresponding input list of q. Henceforth, both "C_q" and "element C_q" mean the ele-

ment that C_q points to. We access the region code of the cursor element by $C_q \rightarrow start$, $C_q \rightarrow end$ and $C_q \rightarrow level$. $C_q \rightarrow$advance() can be invoked to forward the cursor to the next element in the list for QNode q.

4.2 Two main modules in TwigStack

The key idea of the holistic algorithm is a multi-way merge-join on the input element lists. An element can appear in an intermediate path solution only if it is guaranteed to participate in final results. For the element a_1 in Figure 3, it does not have a subtree match due to a missing c-element.

The TwigStack algorithm has a main algorithm and a core subroutine GetNext, as illustrated in Figure 4. The main algorithm calls GetNext to get the next QNode q whose cursor element is processed. It either caches or discards the cursor element C_q and then forwards C_q to the next element. The process in the main algorithm also includes assembling full matches and generating tuple-extraction results with projection. We omit the details of the main algorithm because it is not crucial for understanding the current work. We refer interested readers to [4, 14].

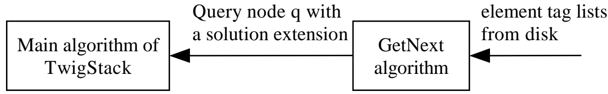


Figure 4: The modules in TwigStack.

GetNext(q) returns a QNode in the subtree q such that the node q has a solution extension, defined as follows:

DEFINITION 2. ***solution extension*** *We say that a QNode q has a solution extension if the current cursor elements for each query edge in subtree q satisfy the ancestor-descendant relationship.*

Informally, when a node q has a solution extension, the cursor elements in the subtree make up a *match* of the subtree if we regard all the query edges as ancestor-descendant edges.

In what follows, we explain the GetNext algorithm.

4.3 The GetNext algorithm

GetNext is initially called by the main algorithm (with the query root as input) to return the highest possible QNode with a solution extension within the whole query. The code is listed in Algorithm 1.

Algorithm 1 GetNext(q)

```
1:  if isLeaf(q) then
2:      return q;
3:  for each q_i in children(q) do
4:      q' = GetNext(q_i);
5:      if q' ≠ q_i then
6:          return q';
7:  end for
8:  q_min = arg min_{q_i} {C_{q_i}→start};
9:  q_max = arg max_{q_i} {C_{q_i}→start};
10: while C_q→end < C_{q_max}→start do
11:     C_q→advance();
12: end while
13: return (C_q→start < C_{q_min}→start)? q : q_min;
```

Consider Algorithm 1. GetNext(q) returns a QNode q' *within* the subtree q such that q' has a solution extension. To achieve that goal, GetNext(q) first recursively calls itself for each of the child nodes of the current input q (lines 3–7).

Given that all child nodes have their own solution extensions (after line 7), in order for node q to be returned, we make sure that the cursor element C_q is a common ancestor of all the child cursor elements (C_{q_i}) by advancing C_q (line 11). Here, the function $\arg\min_{q_i}\{C_{q_i} \rightarrow start\}$ returns the QNode among all the returned QNodes (at line 4) that has the smallest $start$ value. Similarly, $\arg\max_{q_i}\{C_{q_i} \rightarrow start\}$ returns the QNode with the maximum $start$ value. Ties are resolved arbitrarily.

If no common ancestor for all C_{q_i} is found in q, we return the child node with the smallest start value (i.e., q_{min}). We can return q_{min} because no cursor element of q_{min} can be extended with its parent q. Note that as long as q_{min} is returned in line 13, the outer recursive calls of GetNext will return the same node q_{min} all the way up through lines 5–6.

EXAMPLE 2. *We use the data and query in Figure 3 to show how GetNext works. Table 2 lists the cursor elements after each call of GetNext(q_a). After the first call, the cursor elements are (a_2, b_1, c_1). The cursor of q_a was forwarded from a_1 to a_2 at line 11. Given that a_2 is not a common ancestor of b_1 and c_1, we return q_b (line 13). As a convention, the cursor element of the returned QNode is enclosed by parentheses in the table. (The main algorithm will forward C_{q_b} to b_2 after it consumes b_1) Similarly, the second call of GetNext(q_a) returns q_b too with the cursor element b_2. It is interesting to note that both b_1 and b_2 elements are actually discarded by the main algorithm because no a-element has been returned in our example. At the third call of GetNext(q_a), we will return the root q_a because the current cursors make up a solution extension.*

	init	1	2	3	4	5	6
C_a	a_1	a_2	a_2	(a_2)	end	end	end
C_b	b_1	(b_1)	(b_2)	b_3	(b_3)	end	end
C_c	c_1	c_1	c_1	c_1	c_1	(c_1)	end

Table 2: Cursor elements in Example 2.

4.4 Adapting TwigStack for streaming XML

When XML data is streamed in (or provided as a disk file such that we need to access it using SAX parser), the elements arrive in strict document order. For example, when the XML data in Figure 3 is accessed in a streaming fashion, we will not see the element c_1 until we see all the a- and b-elements. As a result, TwigStack is not directly applicable, because it must begin with valid cursor elements—(a_1, b_1, c_1) in this example.

Before we proceed, we need to formally define how streaming XML data is accessed. Usually streaming XML data is encoded as a series of SAX events, including start element (SE), attributes, end element (EE) and text. In our work, text values are part of the elements corresponding to extraction QNodes. Attribute events can be simulated as element events. Without loss of generality, we only consider SE and EE in our exposition. For example, if the XML data tree in Figure 3 is served in a streaming format, we would receive the following sequence of SAX events (with those for x-elements omitted): SE(a_1), SE(b_1), EE(b_1), SE(b_2), EE(b_2), EE(a_1), SE(a_2), SE(b_3), EE(b_3), SE(c_1), EE(c_1) and EE(a_2).

We can easily find a naïve adaptation of TwigStack to handle streaming XML: we parse the incoming XML, store the parsed data in temporary files and then run TwigStack.

However, there are two problems with the naïve approach. First, it causes unnecessary I/Os because all the incoming data needs to be stored onto disk and then read back to run TwigStack. Second and more importantly, it cannot handle continuous streaming XML, which would require infinite temporary disk storage.

TwigStackTX: making cursors stream-aware

We observe that TwigStack can work with streaming XML data as long as the cursor elements are ready when they are accessed in the `GetNext` algorithm; there is no need to wait for all the streaming data to be parsed, which is infeasible for continuous streaming XML. The observation results in the TwigStackTX algorithm, which is the same as TwigStack except that, when access to a cursor element is attempted, we progressively parse the incoming XML data until the cursor element is found from the input.

Take the example in Figure 3. In the first call of $\texttt{GetNext}(q_a)$, the cursor elements C_{q_b}, C_{q_c} and C_{q_a} are accessed in that order. When C_{q_b} is accessed, we parse elements a_1 and b_1. When C_{q_c} is accessed, we parse the data all the way to c_1. Parsed elements are buffered before they are returned to the main algorithm.

Although TwigStackTX seemingly needs to buffer all the elements as does the naïve adaptation for Figure 3, it is more effective in general. For example, even when the a_2 element has many following sibling elements, TwigStackTX can start to work as soon as c_1 is encountered. The naïve approach, however, still needs to parse and store *all* the data.

5. StreamTX: A FULL-FLEDGED ALGORITHM

Although the TwigStackTX algorithm can handle streaming XML by making cursors progressively prepare their next elements, it still has potential buffering problems, which could be serious sometimes. For the example in Figure 3, TwigStackTX needs to buffer both the two a-elements (a_1 and a_2) and the three b-elements (b_1, b_2 and b_3) before it reaches c_1 and starts to do the matching and consume the buffered elements. However we can know that a_1 does not participate in any result when we see the end-element event for a_1 (i.e., EE(a_1)); we can remove a_1 from buffer at that moment, reducing the buffer size.

The aforementioned problem with TwigStackTX is caused by the fact that each cursor aggressively searches for its next element without considering other cursors. In other words, the cursors are not coordinated.

In this section, we present StreamTX, a highly optimized algorithm for tuple extraction from streaming XML. StreamTX is also built on top of the holistic-matching paradigm as TwigStackTX. However, a unique feature of StreamTX is that we coordinate the cursors with *blocking*. At any point during the matching, we may have some cursors do not have associated elements—that it, they are blocked—but we may still be able to continue the matching with non-blocked cursors and emit results. By doing so, incoming elements are consumed as much as possible so that the buffering is minimized, and the response of the tuple-extraction query is also improved.

In Section 5.1, we present the basic StreamTX algorithm that implements our idea of holistic matching with coordinated cursors with blocking. We complete the algorithm in Section 5.2 by proposing two techniques for pruning incoming elements so that the buffering is minimized and at the same time the query performance can be greatly improved.

5.1 Holistic matching with blocking cursors

To realize our idea of coordinated cursors with blocking, we replace `GetNext` (in TwigStackTX) with its streaming counterpart `GetNextStream`, which can block itself and return a *blocked* QNode if it cannot proceed without seeing more SAX events. To implement such a processing paradigm, given each incoming SAX event, we invoke the main algorithm, which repeatedly calls $\texttt{GetNextStream}(root)$ to get the next element for processing until $\texttt{GetNextStream}(root)$ returns a blocked QNode.

5.1.1 Required data structures

We need some special data structures that support the processing of streaming XML data:

- We propose to maintain dynamic element queues in place of the static input lists for QNodes. Element queues can grow at the tail as new elements come (in the form of SE events) and shrink after the head element is processed.
- The cursor on an element queue either points to a valid element in the queue or is in the blocked state. The latter occurs when the element queue is empty.
- For those elements whose EE events have not arrived, their end values are *open*. Interestingly we can still evaluate ancestor-descendant and parent-child relationships with open-ended region codes as follows: given two elements u and v, if element u is open-ended, then u is an ancestor element of v if $u.start < v.start$; on the other hand, if u is not open-ended, we follow the rule defined in Section 2.1 for the evaluation. When the EE event of an open-ended element arrives, we can complete its region code.

5.1.2 The `GetNextStream` algorithm

Algorithm 2 lists the code lines of $\texttt{GetNextStream}(q)$, which returns a QNode either with a valid cursor element (as in $\texttt{GetNext}(q)$) or with a blocked cursor element.

Algorithm 2 $\texttt{GetNextStream}(q)$

```
1: if isLeaf(q) then
2:    return q;
3: for each q_i in children(q) do
4:    q'_i = GetNextStream(q_i);
5:    if not blocked(C_{q'_i}) and q'_i ≠ q_i then
6:       return q'_i;
7: end for
8: q_min = arg min_{q'_i} {C_{q'_i}→start};
9: q_max = arg max_{q'_i} {C_{q'_i}→start};
10: while C_q→end < C_{q_max}→start do
11:    C_q→advance();
12: end while
13: Decide which QNode to return based on the blocking states of
      C_q, C_{q_min} and C_{q_max}. The actions are listed in Table 3.
```

The first half of `GetNextStream` (lines 1–7) is similar to `GetNext` except that $\texttt{GetNextStream}(q_i)$ may return a blocked QNode and we need to take care of such a case (line 5).

When $\texttt{GetNextStream}(q_i)$ returns a blocked QNode, it means that the subtree rooted at q_i is blocked and we are unable to further process subtree q_i before we see more SAX events. Notice that, not only can q_i cause the blocking of the subtree q_i but other descendant QNodes in the subtree can also cause blocking. We can use any blocked QNode in the subtree q_i as the result of $\texttt{GetNextStream}(q_i)$ if the subtree q_i is blocked.

In the second half of the algorithm (starting from line 8), we need to return a QNode with a solution extension, as in `GetNext`. The main difference from `GetNext` is that some child subtrees

case#	q	q_{min}	q_{max}	action
c_1	B	B	B	q
c_2	B	B	NB	(impossible case)
c_3	B	NB	B	return q_{min}
c_4	B	NB	NB	return q_{min}
c_5	NB	B	B	return q_{min}
c_6	NB	B	NB	(impossible case)
c_7	NB	NB	B	$(C_q \rightarrow start < C_{q_{min}} \rightarrow start)$? return q_{max}: return q_{min}
c_8	NB	NB	NB	$(C_q \rightarrow start < C_{q_{min}} \rightarrow start)$? return q: return q_{min}

Table 3: The decision-table for Algorithm 2.

may be blocked and we should decide which QNode to return even in the presence of such blocked subtrees. The underlying idea is that we try the best to return a QNode (as the result of `GetNextStream`(q)) with a solution extension. A blocked QNode is returned *only when* we are unable to find such a solution extension before we see more SAX events. In other words, we adopt a *best-effort* approach. We describe how the best-effort approach is implemened next.

After the recursive calls for child subtrees of q (after line 7), each of the child subtree q_i is associated with its `GetNextStream`(q_i) value q_i', which can be either the same as q_i (i.e., the subtree has a solution extension) or a blocked QNode (i.e., the subtree is blocked). To perform meaningful comparison operations between blocked and unblocked QNodes, we define the $start$ and end values of the cursor of a blocked QNode as follows:

DEFINITION 3. ***($start$ and end values of a blocked cursor)*** *If a QNode is blocked, then the start and end values of its (blocked) cursor are defined as a constant value r_{max}. The value r_{max} is greater than the start and end values of any unblocked cursor.*

The intuition behind the above definition is that the next element of a blocked cursor is always *behind* the current cursors of the unblocked QNodes—i.e., the $start$ and end values of the next element are always greater than the $start$ and end values of the unblocked cursors.

The end value of an open-ended cursor can be defined in a similar spirit:

DEFINITION 4. ***(end value of an open-ended cursor)*** *If the cursor of a QNode is open-ended, the end value of the cursor is defined as the constant value r_{max}.*

According to Definition 3, the function $\arg\max_{q_i'}\{C_{q_i'} \rightarrow start\}$ will return a blocked QNode if there exists one, because a blocked QNode has the largest $start$ value. The function $\arg\min_{q_i'}\{C_{q_i'} \rightarrow start\}$ return a blocked QNode only if all the QNodes in question are blocked.

We skip the elements for the QNode q if it cannot be an ancestor element of the $C_{q_{max}}$—that is, its end value is smaller than $C_{q_{max}} \rightarrow start$ (lines 10–12). In the boolean expression at line 10, both C_q and $C_{q_{max}}$ can be blocked. The expression handles any combination of the blocking conditions of these two QNodes in a semantically correct way. For example, if C_q is blocked, the expression is always `false` because r_{max} cannot be smaller than any $start$ and end values. In other words, if C_q is blocked, we are unable to advance the cursor C_q—which is obviously a correct decision. As another example, if $C_{q_{max}}$ is blocked, the expression is always `true` until C_q is open-ended or blocked. It means that all the closed cursors from C_q can be skipped if $C_{q_{max}}$ is blocked.

As the last step, we decide which QNode to return based on the blocking states of the three QNodes (q, q_{min} and q_{max}) in consideration (line 13). Since each QNode can be either blocked or unblocked. There are eight possible cases in total.

Table 3 is the action table for the eight cases. In the decision table, 'B' stands for "the cursor is blocked" and "NB" stands for "not blocked". For example, the first line (i.e., case c_1) states that if all the three QNodes are blocked, we return a blocked QNode q (we can also return q_{min} or q_{max} because blocked QNodes are treated the same way when returned).

The rules in the action table guarantee that we return a blocked QNode *only when* we are not able to return a QNode with a solution extension before we see more SAX events. With such a best-effort approach, the buffering due to blocking is minimized. For example, under case c_3 where both C_q and $C_{q_{max}}$ are blocked, we can still return the unblocked q_{min} to the main algorithm for processing. The reason is that we know the current solution extension of q_{min} cannot be extended with any (new) element for q. In contrast, we need to continue to buffer $C_{q_{min}}$ in TwigStackTX.

5.1.3 A running example

We proceed to show a running example of `GetNextStream`, using the data and query in Figure 3.

The running statistics are shown in Table 4. The column headers show the SAX events in their arriving order. Here, we use "x" to stand for SE(x), "$/x$" for EE(x), and "init" for the initial state. The rows starting with C_{q_a}, C_{q_b} and C_{q_c} show the content of the corresponding element queue after the incoming SAX event is added to the corresponding element queue. An open-ended element is identified with a hat, such as $\hat{a_1}$. The head of an element queue is the cursor element. If the queue is empty, the cursor is in a blocked state.

	init	a_1	b_1	$/b_1$	b_2	$/b_2$
C_{q_a}	-	$\hat{a_1}$	$\hat{a_1}$	$\hat{a_1}$	$\hat{a_1}$	$\hat{a_1}$
C_{q_b}	-	-	$\hat{b_1}$	b_1	$b_1, \hat{b_2}$	b_1, b_2
C_{q_c}	-	-	-	-	-	-
action	-	c5	c7	c7	c7	c7

	$/a_1$	a_2	b_3	$/b_3$	c_1	$/c_1$	$/a_2$
C_{q_a}	a_1	$\hat{a_2}$	$\hat{a_2}$	$\hat{a_2}$	$\hat{a_2}$	-	-
C_{q_b}	b_1, b_2	-	$\hat{b_3}$	b_3	b_3	-	-
C_{q_c}	-	-	-	-	$\hat{c_1}$	-	-
action	(*)	c5	c7	c7	(*)	c1	c1

Table 4: The running statistics after each SAX event (explained in Section 5.1.3).

After each SAX event, the `GetNextStream`(q_a) is called by the main algorithm of StreamTX. The action row shows which case of the decision table is used to return a QNode in `GetNextStream`(q_a). For the example in Table 4, `GetNextStream`(q_a) always returns a blocked QNode except for the two columns with whose actions are filled with "(*)". We explain these two columns next.

Given EE(a_1), we first update the end value of the region code of a_1. When `GetNextStream`(q_a) is called, since the C_{q_c} is still blocked, we will skip a_1 (line 11 of Algorithm 2) and C_{q_a} becomes blocked; we return QNode q_b with the element b_1 (case 3 of the decision table). Similarly, element b_2 will be consumed too. As a result, before the event SE(a_2), all the element queues are empty.

When the event SE(c_1) arrives, all the three cursors are now holding valid elements. The main algorithm calls `GetNextStream`(q_a)

three times to consume a_2, b_3 and c_1 (the corresponding QNodes of these cursor elements are returned by cases 8, 4 and 3, respectively).

This example shows that StreamTX can consume incoming SAX events greedily based on the decision table, so that the buffer to hold parsed elements is kept as small as possible. In particular, the maximum length for the element queue of QNode q_a is one although there are totally two a-elements. In contrast, both of the two adaptations of TwigStackTX need to cache both a-elements.

5.2 Minimizing element queues with pruning

The `GetNextStream` algorithm in Section 5.1 exploits the relationship between QNodes that are not blocked and QNodes that are blocked, and consumes elements with best-effort without compromising the optimality of holistic twig joins. It however does not guarantee the minimality of element queues, because the holistic matching only looks at the *head* of each element list when searching for a solution extension. Such a conservative approach may cause long element queues unnecessarily.

We present two techniques to minimize the sizes of buffered element queues based on the following two observations:

O1: When a start-element event arrives, all its ancestor elements have arrived.

O2: When an end-element event arrives, all its descendant elements have arrived.

Based on O1, when a start-element event comes, we can check whether the element has corresponding ancestor elements to satisfy the query path. A query path is defined as the path from the root QNode to the QNode corresponding to the element in question. For example, for the QNode q_b in Figure 3(b), its query path is //a/b#. If the element does not satisfy the query path, it can be discarded immediately. We call the technique *query-path pruning*.

Based on O2, when an end-element event arrives, if the element does not have descendant elements to make up a match for the subtree, we can not only prune the element itself but also prune its descendant elements in the element queues. Since the pruning is based on the criterion whether there exists *at least one* subtree match for the closing element, We call the technique *existential-match pruning*. There is no need to instantiate all the matching instances for the closing element to implement the pruning.

5.2.1 Query-path pruning

We explain the idea behind query-path pruning with the example in Table 4. In this example, both b-elements are buffered. If we look closely, we can actually know (at the moment when SE(b_2) arrives) that element b_2 does not have a parent a-element. The reason is that when SE(b_2) arrives, all the start-element events of its ancestors must have arrived; from these arrived ancestor elements (if any), we can make the judgement. In this particular example, the only ancestor element is a_1, which is not a parent element of b_2. As a result, we can discard b_2 without adding it to the element queue C_{q_b}.

Although the query-path pruning only checks the ancestor-descendant or parent-child relationship between an incoming element and its parent element queue, we are virtually checking whether the element has a match for the query path from the root QNode to the QNode where the element belongs.

We can implement the query-path pruning in such a way that the cost of the match-test for each element is always *constant*. The reason is as follows. Given a new incoming open-ended element e to QNode q, all its ancestors in the element queue of `parent`(q) must be open-ended elements at that moment (and in fact, these ancestor elements are nested within each other). As a result, we can maintain a pruning stack of open-ended elements for each element queue. An open-ended element is removed from the stack upon the arrival of its corresponding EE event. To check whether an element has a parent or ancestor element in the element queue of `parent`(q), we check with the top element of the stack maintained for the element queue of `parent`(q)—a constant cost.

THEOREM 1. *The query-path pruning guarantees that each element e (either open or closed) buffered in element queues satisfy its corresponding query path. That is, there exist ancestor elements $a_1, a_2, \cdots, a_n$ such that the element path $a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n \rightarrow e$ satisfies the corresponding query path.*

PROOF. Assume that each element currently in any element queue satisfies the property stated in the theorem–that is, for each element e, there exist ancestor elements for the element e to satisfy its query path. Note that all the open-ended elements maintained in the pruning stacks also belong to the element queues and thus satisfy the same property by assumption. When a new element e arrives, we will add it to the corresponding element queue if and only if there exists an ancestor element a in the pruning parent stack such that e and a satisfy the corresponding structural relationship between them. By our assumption, there exist ancestor elements a_1, a_2, $\cdots$, a_n for element a to satisfy its query path, and thus, it is proved that there exist ancestor elements $a_1, a_2, \cdots, a_n, a$ for the element e to satisfy its query path. □

5.2.2 Existential-match pruning

To implement the existential-match pruning, we need to keep a matching flag for each non-leaf open-ended element in element queues. The flag is a boolean value indicating whether the element has matching descendant elements according to the query pattern.

To maintain the flag, whenever the SE of a leaf QNode arrives, we update the flags of all the open-ended elements along the query path. When the EE event of element e arrives, we are able to decide whether element e has a solution extension based on the value of the flag, because future elements cannot be descendant elements of e. If the flag indicates that the element does not a solution extension, we will simply discard it. As a result, the existential-match pruning guarantees that for each closed element e buffered in any element queue, there exists a solution extension rooted at e.

Without providing the implementation details, we show how the existential-match pruning can help reduce element buffer sizes with a very simple example.

EXAMPLE 3. *Suppose the incoming XML is a path with three elements: $a_1 \rightarrow a_2 \rightarrow b_1$, where a_1 is the root element and b_1 is the leaf element. The query is `//a[b#]//c#` (see, the pattern in Figure 3(b)). Table 5 shows the running statistics of StreamTX without the existential-match pruning (see, Section 5.1.3 for basic explanation of the table content). When the end-element event of a_2 (i.e., $/a_2$) arrives, a_2 and b_1 are still kept in the element queues. However, it is clear that element a_2 does not have a subtree match due to the missing c-element descendant element. If we have enabled the existential-match pruning, then the flag for element a_2 is `false` Therefore, we can remove a_2 and also the element b_1 because a_2 is the only ancestor element of b_1. Under the extreme case where a_2 has many following sibling a-elements that only have b descendants, we can use the existential-match pruning to prune these a-elements, which otherwise would stay in the buffer until EE(a_1) arrives.*

There are two points worth noting about the existential-match pruning. First, we can apply the cascaded pruning of descendant

	init	a_1	a_2	b_1	$/b_1$	$/a_2$	$/a_1$
C_{q_a}	-	$\hat{a_1}$	$\hat{a_1}$,$\hat{a_2}$	$\hat{a_1}$,$\hat{a_2}$	$\hat{a_1}$,$\hat{a_2}$	$\hat{a_1}$,a_2	a_1,a_2
C_{q_b}	-	-	-	$\hat{b_1}$	b_1	b_1	b_1
C_{q_c}	-	-	-	-	-	-	-
action	-	c5	c5	c7	c7	c7	(*)

Table 5: The running statistics without existential matching (explained in Example 3).

elements only if they do not match with other valid ancestor/parent elements. Second, if the cascaded pruning is applicable (i.e., the aforementioned condition is satisfied), it can be carried out efficiently because all the pruned descendant elements are clustered at the tails of the corresponding element queues.

We can maintain the flag to indicate the existence of a solution extension with little CPU cost. The main reason is that this flag only tests the existence of solution extensions; it does not enumerate all possible solution extensions or output any solution extension.

6. PERFORMANCE EVALUATION

We study the performance of StreamTX for tuple extraction, in comparison with other alternatives, and report some experiment results. Specifically, we study the impact of the pruning techniques in Section 6.2. A comparison between StreamTX and its direct competitors TurboXPath and TwigStackTX is conducted in Section 6.3. In Section 6.4, we compare StreamTX with some XQuery engines for tuple extraction. Finally, in Section 6.5, we show the effectiveness of our memory minimization technique by comparing with GCX, an in-memory XQuery engine that uses state of the art buffer minimization techniques.

6.1 Experimental setup

All the experiments were conducted on a Pentium IV 1.80GHz Thinkpad with 1GB RAM and an 80G hard drive, running Windows XP. All the algorithms StreamTX, TwigStackTX and TurboXPath are implemented with Microsoft Visual C++. In particular, the algorithms are built on top of a buffer manager and a disk manager so that all disk accesses are done through the buffer manager. For each element queue, we used a modified heap-file implementation that supports removing elements from its head and adding elements to its tail. We set the buffer manager to use 5MB main memory for all the experiments. This size is large enough to hold overhead memory required by our system while it is still too small to hold the whole input data used in the experiments.

We used the Xerces-C++ (version 2.6.0) [26] for parsing input XML documents into SAX events. Particularly, we used the pull-based mode to parse XML documents—that is, our algorithms call the `parseNext` function of the SAX parser to get the next SAX event.

6.1.1 XML data

We used two real-world XML data sets: DBLP [8] and Treebank [25]. As mentioned earlier, DBLP contains bibliographical data and has relatively flat structure. The Treebank data comes from the Penn Treebank Project, which annotates naturally-occurring text for linguistic structure. Tags in the Treebank data are recursive and highly nested.

The DBLP data is about 180MB in size, containing all the inproceedings entries. The Treebank data is of 160MB in size.

6.1.2 Queries tested

Table 6 lists the eight tuple-extraction queries used in our experiments. The first three are DBLP queries and the remaining five are for Treebank data.

Name	Query
QD1	/dblp/inproceedings[title#]/author#
QD2	/dblp/inproceedings[title# and booktitle#]/author#
QD3	/dblp/inproceedings[title# and booktitle# and year#]/author#
QT1	//S[.//VP[.//JJ][.//VBD]]//NP[.//WP]//DT#
QT2	//S//NP[./IN][.//VBN]//JJ#
QT3	//S[.//VBP][.//SBAR]//S//NP[.//IN]//DT#
QT4	//S[.//NP[.//DT][.//NN]]//PP[.//TO]//NN#
QT5	//S[.//S][.//VP[.//VBD]]//NP[.//IN]//DT#

Table 6: The positive queries used in our experiments: three DBLP queries and five Treebank queries.

The queries in Table 6 are *positive* queries in that they all return non-empty results. To test the performance of different approaches under selective queries, we created a negative query (an extremely selective query) from each positive query by changing the first tag to a `dummy` tag. For example, for QD1, we change the `inproceedings` tag to `dummy` to create the corresponding QDN1. Such extremely selective queries can be frequently used for heterogeneous XML data where users do not know XML schemas in advance.

6.2 Impact of the pruning techniques

We are interested in learning the impact of the pruning techniques (see, Section 5.2) to the basic StreamTX algorithm, with respect to query complexity and selectivity.

The baseline of our study is the StreamTX algorithm with no pruning. The results are presented using the running-time ratio between the baseline algorithm and the StreamTX algorithm with the specified pruning technique(s). The common time used by the SAX parser is excluded to mimic a streaming environment. For example, the baseline algorithm used 168 seconds for the query QD1 and 76 seconds was spent by SAX parsing. So the baseline time is 92 seconds.

Table 7 shows the comparative results for the 180MB DBLP data, and Table 8 is for the 160MB Treebank data. The numbers show the speedup from the pruning techniques (QPP, EMP or both) over the baseline algorithm (without pruning). QPP stands for query-path pruning, EMP stands for existential-match pruning, NONE stands for the baseline StreamTX and BOTH means both pruning techniques are applied.

	QD1	QD2	QD3	QDN1	QDN2	QDN3
QPP/NONE	0.97	0.96	0.96	10.13	13.16	22.05
EMP/NONE	0.95	0.95	0.94	0.66	0.60	0.62
BOTH/NONE	0.94	0.95	0.94	5.77	13.16	20.79

Table 7: Speedups of DBLP queries: pruning / no pruning.

There are two findings about the query-path pruning (QPP):

- QPP is very effective for negative queries. For example, the speedup is 22 times for QDN3. The reason is that, since the topmost tag (i.e., `dummy`) in the negative queries does not match any incoming element, elements designated to all the element queues are pruned. The running time with QPP is less than one second for 180MB DBLP data.
- The overhead of the QPP pruning is small. We can see it from the results for positive queries, for which the QPP prunes much

	QT1	QT2	QT3	QT4	QT5
QPP/NONE	0.86	1.03	1.13	1.02	0.99
EMP/NONE	0.65	0.78	0.82	0.74	0.76
BOTH/NONE	0.67	0.76	0.85	0.82	0.80

(a) Positive queries

	QTN1	QTN2	QTN3	QTN4	QTN5
QPP/NONE	11.20	7.25	16.26	7.13	8.13
EMP/NONE	0.72	0.69	0.65	0.68	0.74
BOTH/NONE	9.45	4.17	12.17	6.08	6.73

(b) Negative queries

Table 8: Speedups of Treebank queries: pruning / no pruning.

fewer elements. For most queries, using QPP does not slow down the algorithm. The only case is QT1, for which the ratio is 0.86, meaning that the baseline algorithm is a little faster than using the QPP.

The results show that the existential-match pruning (EMP) does not contribute any speedup to the queries tested. Conversely, the EMP pruning slows down the running time. The negative effect of EMP is more noticeable for negative queries than for positive queries (0.60 for QDN2). Given that the absolute running times for negative queries are much smaller than those for positive queries (the baseline algorithm needs 92 seconds for QD1 but only 9 seconds for QDN1), the overhead of EMP becomes more noticeable.

It is interesting to note that combined impact of these two pruning techniques is not a simple summation of the individual impacts. For example, for the negative queries, since the QPP dominates, the negative impact of EMP is almost negligible.

Although the EMP did not bring speedup to StreamTX for the tested queries, it is still an important part of the StreamTX algorithm because it guarantees the "goodness" of elements in the queues and helps avoid worst-case scenarios in buffer sizes. It is up to the applications to decide whether to turn on the EMP based on query/data statistics and memory availability. For example, we may know that the existential-match pruning can be turned off without penalty for the DBLP and Treebank data after running a few example queries.

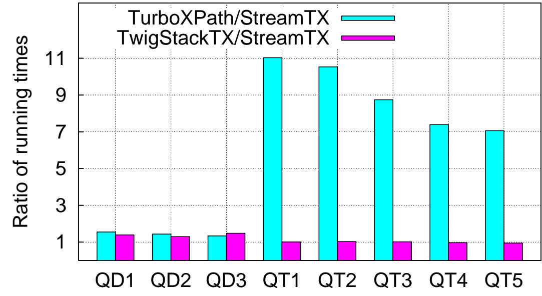
In the following experiments, we use the results of StreamTX that uses the query-path pruning only.

6.3 Comparison with direct competitors

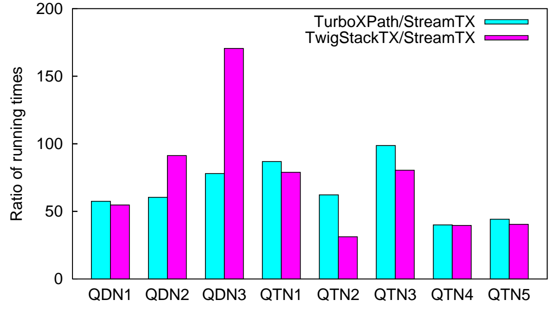
We also ran all the positive and negative queries using TurboXPath and TwigStackTX. The comparative results are displayed in Figure 5.

The comparative performance between StreamTX and TurboXPath can be summarized as follows:

- **Positive queries** Not surprisingly, the speedup of StreamTX for the nested Treebank data is greater than that for the plain DBLP data. In particular, StreamTX is about 1.5 times faster than TurboXPath while it can be up to 11 times faster for Treebank. The underlying reason for the difference is that TurboXPath is optimized for plain data and does not handle recursions effectively. Specifically, whenever an incoming element matches with a QNode q, TurboXPath creates so-called work-array (WA) entries for the child QNodes of q. If multiple incoming elements match with the same QNode, WA entries are repeatedly created for each match. For QT1 which only has 7 QNodes, TurboXPath created up to 197 WA entries during the match. The large number of duplicate WA entries for same QNodes brings significant overhead to the processing. In contrast, StreamTX handles recursions gracefully with open-ended region codes and inter-linked stacks of elements (in the main algorithm). With inter-linked stacks, we can encode potentially exponential number of results with a linear data structure.
- **Negative queries** The biggest speedup numbers of StreamTX come from negative queries (e.g., 110 times for QTN3). Although TurboXPath spends less time for negative queries (about half of that for the corresponding positive queries), StreamTX is more effective in reducing running time for negative queries, primarily due to the query-path pruning.



(a) Positive queries.



(b) Negative queries.

Figure 5: Speedups of our StreamTX over TurboXPath and our TwigStackTX.

6.3.1 *Summary of StreamTX vs. TurboXPath*

StreamTX has demonstrated superior performance advantage over TurboXPath, both for positive queries and negative queries. The advantage is particularly significant for negative queries.

Next, we analyze the relative performance between StreamTX and TwigStackTX:

- **Positive queries** For DBLP queries, StreamTX is about 1.4 times faster than TwigStackTX. The main reason is that, after the first (and the only) `dblp` element is consumed, the cursor corresponding to the QNode `/dblp` in TwigStackTX greedily searches for the next *nonexistent* `dblp` element. During the process, all the encountered elements are buffered, causing buffer overflow and disk I/Os. The same problem does not exist for the Treebank data in which elements corresponding to QNodes are almost uniformly interleaved and the match can start after a moderate number of elements are buffered; no disk I/O is involved. As a result, TwigStackTX and StreamTX are almost the same for positive Treebank queries.
- **Negative queries** StreamTX is significantly better than TwigStackTX for negative queries. In particular the speedup reaches up to

170 times for QDN3 (StreamTX takes 790 milliseconds while TwigStackTX takes up to 113 seconds). The cause of the results is also clear: since the `dummy` tag in the negative queries does not have corresponding elements, when TwigStackTX searches for the first `dummy` tag, actually *all* the incoming data is buffered. Such behavior is unacceptable if the XML stream is continuous.

6.3.2 *Summary of StreamTX vs. TwigStackTX*

StreamTX is much more efficient and robust than TwigStackTX. The TwigStackTX algorithm becomes problematic when the arrival rates of elements for element queues are very different—that is, there are many elements to some element queues while there are very few elements to some other element queues. In such cases, the high-rate elements are extensively buffered regardless of query selectivity. In contrast, StreamTX may be able to consume these high-rate elements even in the absence of some low-rate elements. The negative queries in our experiments are one such example where the element arrival rate for the `dummy` QNode is zero.

6.4 Comparison with streaming XQuery engines

We also compare StreamTX with some XQuery engines that are publicly available. Specifically, we use Galax (version 0.5.0) [11], Quip (version 2.2.1.1) [21], MonetDB/XQuery (version 0.10.2) [19] and Saxon for .NET (version 8.7) [22].

We note that TurboXPath was compared to many other approaches and showed performance advantage over them [15]. In particular, it is experimentally shown that TurboXPath performs much better than the XSQ streaming engine [20].

We ran all these engines with different input data sizes. We get the data sets of smaller sizes by extracting an appropriate portion of data from the full data set. We show the results for QD3 and QT1; the results for other queries are analogous.

Figure 6(a) shows the timings for the DBLP data sets. We observe that StreamTX scales almost linearly with the input data sizes. Other engines, except for Saxon, failed at some input data sizes due to different reasons: (Galax, 20MB, finished without any output), (MonetDB/XQuery, 60MB, exit with “operation failed”), (Quip, 20MB, out of memory).

Saxon performed quite well for smaller DBLP data sets. However for 180MB input data, its performance degraded significantly. Saxon builds an in-memory DOM tree for the whole data, and therefore cannot process large data sets or streaming data.

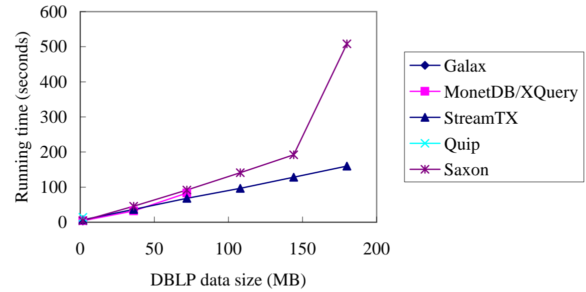
We show the timings for QT1 in Figure 6(b). Again, StreamTX can gracefully handle large input data without any problem, while all the tested engines failed at some point for the reasons listed below: (Galax, 80MB, finished without output), (MonetDB/XQuery, 80MB, system crash), (Quip, 4MB, out of memory), (Saxon, 320MB, out of memory).

In fact, we have tested StreamTX for large data sets of GB-range without any problem.

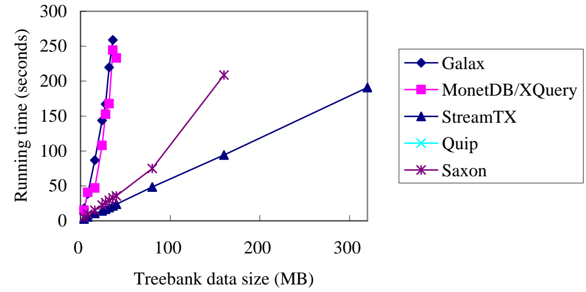
6.5 Comparison with state of art buffer minimization technique

To show the effectiveness of our memory minimization technique, we compare with GCX 1.0 [1] [23, 16], a state of the art in-memory XQuery engine. For a fair comparison, we modified GCX to use the same parser (i.e., Xerces-C++) used by StreamTX. The hardware used for these tests consists of an Intel Core 2 Quad 2.4GHz PC with 2GB RAM, running Windows Vista. We note that

[1]The source code can be downloaded from http://dbis.informatik.uni-freiburg.de/index.php?project=GCX.



(a) DBLP with the query QD3.



(b) Treebank with the query QT1.

Figure 6: Comparison of our StreamTX with XQuery engines.

we used a different physical environment to show the stability of our system in different hardware environments.

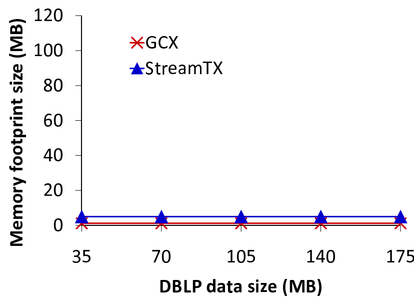
We measure memory footprint sizes as well as overall running times of both systems for different input data sizes. We show the results for QD3 and QT1; the results for other queries are analogous.

Figure 7(a) shows memory footprint sizes for DBLP data sets. Both systems required constant memory size. This means that the buffer minimization technique of GCX works as good as StreamTX for very flat XML data such as DBLP. However, as in Figure 7(b) showing memory footprint sizes for Treebank data sets, GCX required about the same memory size as the input XML file size whereas StreamTX used only 5 Mbytes memory for its buffer manager and 1∼4 Mbytes for its execution code and stack. This indicates that GCX is not scalable for large (recursive and highly nested) XML files.

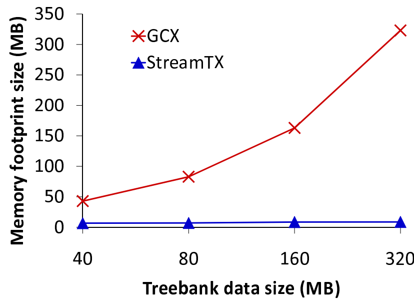
Figure 8(a) shows running times for DBLP data sets. StreamTX performed marginally worse than GCX in terms of elapsed time. However, considering that StreamTX is a disk-based engine using a buffer manager as opposed to GCX which is an in-memory engine, StreamTX showed fast performance results for flat XML files. Figure 8(b) shows running times for Treebank data sets. StreamTX performed at least two times faster than GCX. If the size of an input XML file is larger than the physical memory size of the system, GCX would perform much slower than StreamTX. As mentioned in Section 3, in GCX, full XPath expressions with multiple extraction points need to be expressed using the for-clause in XQuery for processing. However, a series of nested for-loops limits the opportunities to optimize the query as can be exploited in the holistic approach of StreamTX.

7. CONCLUSION AND FUTURE WORK

We have studied the problem of extracting flat tuple data by means of pattern matching from streaming XML data. Our contri-

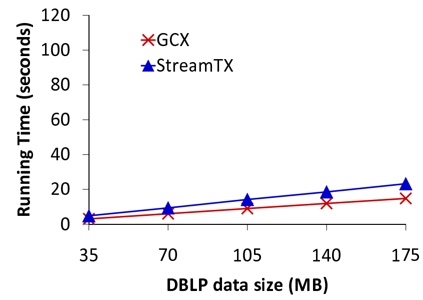


(a) DBLP with the query QD3.

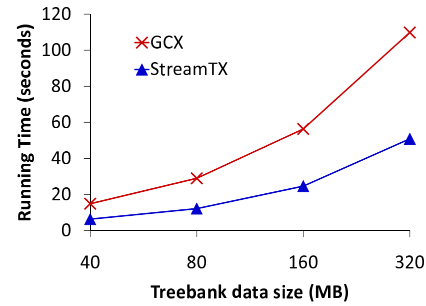


(b) Treebank with the query QT1.

Figure 7: Memory footprint sizes for StreamTX and GCX.



(a) DBLP with the query QD3.



(b) Treebank with the query QT1.

Figure 8: Comparison of our StreamTX with GCX.

bution is StreamTX, an efficient and scalable algorithm for implementing such tuple extraction. We approached the tuple-extraction problem from an angle different from that most exiting work has taken: we started from the scalable holistic-matching paradigm originally designed for parsed XML data and adapted it for handling streaming XML with novel block-and-trigger mechanism and pruning techniques. The outcome is an algorithm that guarantees optimality in terms of input/output sizes and needs only minimal buffer sizes for incoming data. Our experiments demonstrate very encouraging performance benefits of StreamTX compared to the existing state-of-the-art, and we expect the adoption of StreamTX by high-performance systems.

Real-world applications motivate us to consider more flexible tuple-extraction problems such as optimizing the matching of multiple tuple-extraction queries, handling optional extraction nodes, extracting XML subtrees as values (in contrast to direct element texts), and combining texts of repeatable sibling elements into one extraction field. Our future work should address these practical problems.

8. REFERENCES

[1] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB*, 2000.

[2] A. Balmin, T. Eliaz, J. Hornibrook, L. Lim, G. M. Lohman, D. Simmen, M. Wang, and C. Zhang. Cost-based optimization in db2 xml. *IBM Syst. J.*, 45(2):299–319, 2006.

[3] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0. Technical report, W3C Working Draft, 2003.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, 2002.

[5] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig^{2}stack: Bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proc. of the 32nd Int'l Conference on Very Large Data Bases*, pages 283–294, Seoul, Korea, September 2006.

[6] T. Chen, J. Lu, and T. W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *SIGMOD*, 2005.

[7] Y. Chen, S. B. Davidson, and Y. Zheng. An efficient XPath query processor for XML streams. In *Proc. of the 22nd Int'l Conference on Data Engineering*, pages 79–79, Atlanta, GA, April 2006.

[8] DBLP. http://dblp.uni-trier.de/xml/.

[9] P. F. Dietz. Maintaining order in a linked list. In *Proc. of the 18th Annual ACM Symposium on Theory of Computing*, 1982.

[10] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL streaming XQuery processor. In *VLDB*, 2003.

[11] Galax (version 0.5.0). http://www.galaxquery.org/.

[12] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: From research prototype to industrial tool. In *SIGMOD*, 2005.

[13] H. Jiang, H. Ho, L. Popa, and W.-S. Han. Mapping-driven xml transformation. In C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, editors, *WWW*, pages 1063–1072. ACM, 2007.

[14] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *VLDB*, 2003.

[15] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *VLDB Journal*, 2005.

[16] C. Koch, S. Scherzinger, and M. Schmidt. The gcx system: Dynamic buffer minimization in streaming xquery evaluation. In *VLDB*, pages 1378–1381, 2007.

[17] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *VLDB*, 2004.

[18] X. Li and G. Agrawal. Efficient evaluation of XQuery over streaming data. In *VLDB*, 2005.

[19] MonetDB/XQuery (version 0.10.2). http://monetdb.cwi.nl/XQuery.

[20] F. Peng and S. S. Chawathe. XSQ: A streaming XPath engine. *TODS*, 30(2), 2005.

[21] Quip (version 2.2.1.1). http://www.softwareag.com.

[22] Saxon for .NET. http://saxon.sourceforge.net.

[23] M. Schmidt, S. Scherzinger, and C. Koch. Combined static and dynamic analysis for effective buffer minimization in streaming xquery evaluation. In *ICDE*, 2007.

[24] D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[25] Treebank. http://www.cis.upenn.edu/ treebank/.

[26] Xerces-C++ (version 2.6.0). http://xml.apache.org/xerces-c/.

[27] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.