The VLDB Journal (2013) 22:177–202
DOI 10.1007/s00778-012-0281-y

REGULAR PAPER

A survey on XML streaming evaluation techniques

Xiaoying Wu · Dimitri Theodoratos

Received: 9 July 2011 / Revised: 11 May 2012 / Accepted: 16 May 2012 / Published online: 10 June 2012

Abstract XML is currently the most popular format for exchanging and representing data on the web. It is used in various applications and for different types of data including structured, semistructured, and unstructured heterogeneous data types. During the period, XML was establishing itself, data streaming applications have gained increased attention and importance. Because of these developments, the querying and efficient processing of XML streams has became a central issue. In this study, we survey the state of the art in XML streaming evaluation techniques. We focus on both the streaming evaluation of XPath expressions and of XQuery queries. We classify the XPath streaming evaluation approaches according to the main data structure used for the evaluation into three categories: automaton-based approach, array-based approach, and stack-based approach. We review, analyze, and compare the major techniques proposed for each approach. We also review multiple query streaming evaluation techniques. For the XQuery streaming evaluation problem, we identify and discuss four processing paradigms adopted by the existing XQuery stream query engines: the transducer-based paradigm, the algebra-based paradigm, the automata-algebra paradigm, and the pull-based paradigm. In addition, we review optimization techniques for XQuery streaming evaluation. We address the problem of optimizing XQuery streaming evaluation as a buffer optimization problem. For all techniques discussed, we describe the research issues and the proposed algorithms and we compare them with other relevant suggested techniques.

Keywords XML streaming evaluation · XPath · XQuery · XML query optimization

1 Introduction

Extensible Markup Language (XML) is by now the de facto standard for exporting and exchanging data on the web due to its flexibility of organizing data: its inherent self-describing capability, and its semistructured characteristics [4]. XML has become the popular exchange format for representing many types of data, including structured, semistructured, and unstructured heterogeneous records, "e-science" data (in astronomy, biology, earth science, etc.), and digitized images, among others. As increasing amounts of information are stored, exchanged, and presented using XML, it becomes increasingly important to effectively and efficiently query XML data sources.

Along with XML, a new data evaluation model for queries called *streaming* model has emerged. In the framework of the streaming model, data are assumed to arrive continuously in the form of *streams*, are unindexed, and can potentially be unbounded. Because of the limited storage space available, systems that query data streams require algorithms that process data in only one sequential scan and return query results on the fly. Streaming processing is the only option in a number of applications such as financial applications, data monitoring in sensor networks, managing network traffic information, telecommunications data management, web applications, security, manufacturing, distributed and grid-

Electronic supplementary material The online version of this article (doi:10.1007/s00778-012-0281-y) contains supplementary material, which is available to authorized users.

X. Wu (✉)
State Key Lab. of Software Engineering,
Wuhan University, Wuhan, China
e-mail: xiaoying.wu@gmail.com

D. Theodoratos
New Jersey Institute of Technology, Newark, NJ, USA
e-mail: dth@cs.njit.edu

based computing, and others [10,38,59]. Efficiently detecting patterns in streams has become increasingly important also for modern enterprizes that need to react quickly to critical situations.

Different applications use streaming data in XML format. Expressed in a platform-neutral format, XML streams are particularly suitable for data archiving or publication on the web [81]. For these reasons, the processing and querying of XML streams has become an important topic over the past few years. An overview of recent XML streaming applications is presented in "Appendix A in Electronic Supplementary Material".

1.1 The problem

The purpose of querying data streams is to identify specific data patterns in the continuous flow of data. Research on XML streams has mainly focused on queries expressed using XML query languages such as XPath [5] and XQuery [6]. We briefly introduce XPath and XQuery below.

XPath and XQuery. XPath [5] is a navigational language for querying and transforming XML data. It is used for navigating through the hierarchical tree structure of an XML document and for matching (testing whether or not a collection of nodes matches a pattern). A simple XPath expression consists of a sequence of axes and labeled nodes. Two commonly used axes are the *child* axis and the *descendant* axis. Generally, an XPath query might involve multiple predicates and includes one output node. XPath lies at the core of XQuery [6], which is the standard query language for XML data. The XQuery language is designed to extract and restructure subtrees within XML documents. An XQuery query is composed of FOR, LET, WHERE, and RETURN clauses, which together form a FLWR block. The FOR and LET clauses provide a series of XPath expressions for selecting input nodes. The WHERE clause defines selection and join predicates. The RETURN clause creates the output XML structure. XQuery expressions can be nested within the above clauses to build hierarchical expressions.

Challenges and issues. The problem of XPath query evaluation against (persistent or streamed) XML data is a fundamental database problem in the context of XML [77]. The evaluation of an XPath query yields a *set* of nodes that is further sorted in document order. This incurs an expensive duplicate elimination operation that can impact query performance considerably. Thus, the key issue of an efficient XPath evaluation [40] is avoiding duplicate generation at any time during processing. The compositional syntax of XQuery makes XQuery much more expressive than XPath. As we will show later, the rich semantics of XQuery makes its evaluation and optimization problem even more challenging. A survey on the XML query evaluation techniques over persistent (stored) XML data is presented in [44].

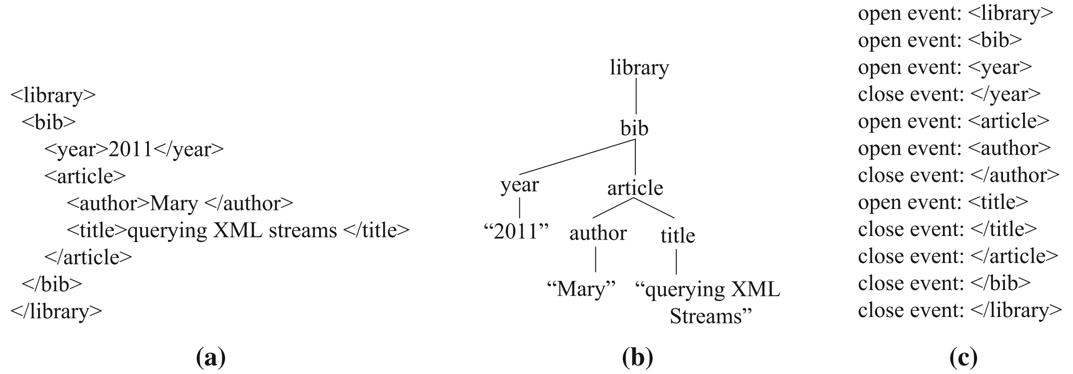
Data streams pose new challenges to query evaluation. In the streaming environment, data arrive continuously, are unindexed, and can be unbounded. Because of the limited storage space available, systems that query data streams require algorithms that process data in only *one sequential* scan and return query results on the fly. New techniques are needed to evaluate possibly complex queries in real-time using as little space as possible for temporary results. Unlike relational streaming data, which are flat and consist of attribute-value pairs or tuples, XML data streams have tree-like structures, possibly recursive, whose size and nesting depth can potentially be unbounded. All these characteristics make the evaluation of XML data streams particularly challenging.

Over the past few years, considerable research efforts have been directed on developing evaluation algorithms over streaming XML documents. A large number of these streaming algorithms consider evaluating queries belonging to different subclasses of XPath [13,27,43,48,66,82,85,86,101]. During this time, various XQuery stream query engines have been developed [31,33,36,52,54,58,64,66,69,91]. Most of these engines support only a subset of the XQuery language. These query evaluation algorithms for XML data streams share the following common characteristics: (1) they consider single-pass query evaluation techniques that require limited storage or no storage at all of the input data stream, and (2) they realize techniques for dealing with and reducing the amount of data buffered in main memory (and this is particularly true for algorithms in XQuery stream query engines).

XML versus relational streaming. A lot of research has also been done on processing relational streams. In a relational streaming environment, a stream consists of multiple homogeneous substreams of tuples, explicit punctuations are used to identify "end of processing", and multi-way joins are employed to process the data. In contrast, in the XML streaming context, a stream is a heterogeneous sequence of XML constructs, open and close events are used to identify query scope, and path matching techniques are employed to process the data [59]. Relational stream processing techniques are summarized and reviewed in [10,38,59].

1.2 Outline

In this study, we survey the state of the art in XML streaming evaluation techniques. We focus on two problems: XPath streaming evaluation (Sect. 3) and XQuery streaming evaluation (Sect. 4). We classify the XPath streaming evaluation approaches according to the main data structure used for matching of structural relationships into three categories: automaton-based approach (Sect. 3.3), array-based approach (Sect. 3.4), and stack-based approach



```
<library>
 <bib>
   <year>2011</year>
   <article>
      <author>Mary </author>
      <title>querying XML streams </title>
   </article>
 </bib>
</library>
```

```
open event: <library>
open event: <bib>
open event: <year>
close event: </year>
open event: <article>
open event: <author>
close event: </author>
open event: <title>
close event: </title>
close event: </article>
close event: </bib>
close event: </library>
```

Fig. 1 An example **a** XML document, **b** XML tree, and **c** the sequence of stream events

(Sect. 3.5). We review and compare the major techniques of these approaches. We also review streaming evaluation techniques for XPath queries with ordered axes (Sect. 3.6) as well as multiple query streaming evaluation techniques (Sect. 3.7). For the XQuery streaming evaluation problem, we classify the existing XQuery stream query engines by their processing paradigms and provide a high level description for each paradigm (Sect. 4.1). In addition, we review optimization techniques for XQuery streaming evaluation (Sect. 4.2). An overview of the XML streaming evaluation model is presented in the next section. We summarize and conclude in Sect. 5.

2 Data and evaluation models

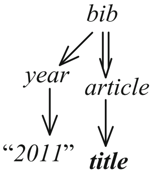
In this section, we introduce the basic terminology and notation. We first define the XML data model and the syntax and semantics of queries. Then, We introduce the event-based XML streaming evaluation model. Finally, we provide definitions of some concepts used throughout the article.

2.1 Data model and query language

Data model. An XML document is commonly modeled by a tree structure. Tree nodes represent and are labeled by elements, attributes, or values. Tree edges represent element–subelement, element–attribute, and element–value relationships. Figure 1a shows the XML tree corresponding to the XML document of Fig. 1b.

Fragments of XPath. Let / denote the child axis, // the descendant axis, and [] denote the branching predicate in XPath. Let also XP$^{\{/,//,[]\}}$ denote the fragment of XPath corresponding to XPath expressions that involve only /, //, and []. We call tree-pattern queries (TPQs) the fragment XP$^{\{[],/,//,*\}}$ of XPath that involves predicates([]), child (/) and descendant (//) axes, and wildcards (*). Existing XML streaming algorithms focus almost exclusively on TPQs. We call directed acyclic graph (*dag*) queries, the fragment of XPath, denoted XP$^{\{[],/,//,*,\backslash,\backslash\backslash\}}$, that involves, in addition, parent axes (\), and ancestor axes (\\). Such queries can be represented as dags that involve child and descendant relationships. A TPQ or a dag includes one distinguished node called *output node*. In the following, an unqualified reference to a query refers to a dag query. Throughout the article, when we do not use names, we often use capital letters to denote query nodes and lowercase letters to denote nodes in XML trees. In the figures, a single (resp. double) edge between two nodes of a query denotes a child (resp. descendant) structural relationship between the two nodes. A node label shown in bold denotes the output node of the query. Figure 2 shows the XPath query //bib[year=ˋ] //article/title on the XML tree of Fig. 1a as a TPQ.

Fig. 2 A TPQ for the XPath query $//bib[year =$ “2011″]$//article/title$



Query embeddings. An *embedding* of a query Q into an XML tree T is a mapping M from the nodes of Q to nodes of T such that: (a) a node in Q labeled by A is mapped by M to a node of T labeled by a; (b) if there is a single (resp. double) edge between two nodes X and Y in Q, $M(Y)$ is a child (resp. descendant) of $M(X)$ in T.

Query answers. The image of the output node of Q under an embedding of Q to T is a *solution* of Q on T. The *answer* of Q on T is the set of all the solutions of Q on T.

2.2 Streaming evaluation model

In a streaming evaluation, an XML document tree flows in as a stream of *open* and *close* events. The appearance of events corresponds to the pre-order traversal of the XML document tree. For each element node in the tree, an *open* event is produced when the opening tag of the node is encountered and the node is called *open* from then on until it closes. After the subtree rooted at that node is processed, a *close* event is produced when the closing element tag of that node is encountered. At this time, the node *closes*. The sequence of open and close events for the XML tree of Fig. 1b is shown in Fig. 1c.

An XML streaming algorithm uses an event-based parser, for example, a SAX XML parser [72], to scan an XML document and produce a stream of events. The streaming algorithm registers functions that are invoked by the parser on open and close events.

Lazy and eager evaluation. Streaming algorithms can be categorized into two classes based on when they evaluate the query predicates [43]. A *lazy* streaming algorithm evaluates query predicates only at close events, whereas an *eager* streaming algorithm eagerly evaluates query predicates before their corresponding close events are encountered. The lazy evaluation strategy is more straightforward than the eager one. However, the eager evaluation strategy has the advantage of optimizing memory usage and reducing query response time.

2.3 Preliminaries

We now introduce concepts used below in describing streaming evaluation algorithms and analyzing their computational complexity.

Ancestor and descendant queries. Given a query Q and a node X in Q, we call *ancestor* query of X the subquery of Q that consists of X and the ancestor nodes of X. We call *descendant* query of X the subquery of Q rooted at X that consists of X and all the descendants of X in Q. For example, consider node $article$ in the query of Fig. 2. Its ancestor and descendant queries are $//bib//article$ and $//article/title$, respectively.

If X is the output node of Q, the ancestor nodes of X are called *backbone* nodes of Q, and the rest of the nodes of Q are called *branching (predicate)* nodes.

Ancestor and candidate query matches. Let x be a node in an XML tree T. Node x is an *ancestor match* of query node X if it is the image of X under an embedding of the ancestor query of X to the path from the root of T to x in T. Node x is a *candidate match* of X if it is the image of X under an embedding of the descendant query of X to the subtree rooted at x in T. A candidate match of X is a *candidate output* if X is the output node of Q.

Recursion depth. An XML document contains recursive structures when more than one node in the same path of the data tree has the same label. The *recursion depth* of a query node X in Q on an XML tree T is defined as the maximum number of nodes in a path of T that are ancestor matches of X [11]. For example, the recursion depth of node A in the query `//A//B/C` on the XML fragment of Fig. 3b is two.

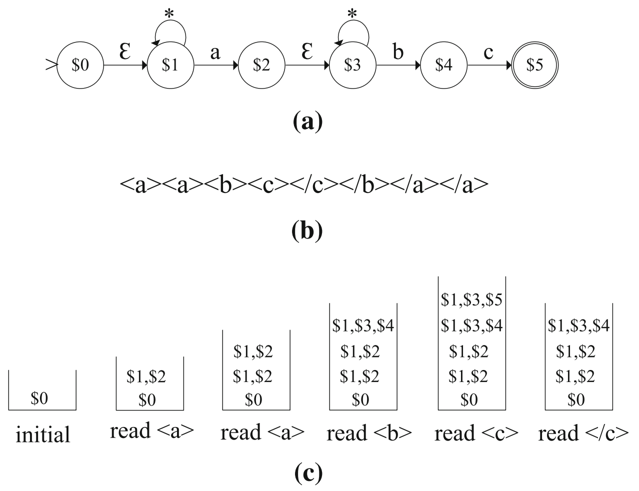


Fig. 3 Evaluating a path query with a basic automaton-based evaluation paradigm. **a** A NFA for the XPath query `//A//B/C`, **b** an XML fragment, **c** the snapshot of the runtime stack

3 XPath streaming evaluation

Two problems are commonly studied in the context of XPath streaming evaluation. One of them is the stream *filtering* problem: given a set of queries, the goal of the stream filtering problem is to determine which of them have a nonempty output on an incoming XML document stream. The other one is the stream *querying* problem which requires finding all the matches of the output nodes of a given set of queries against the XML stream. Techniques for the filtering problem are simpler than those for the querying problem since they only need to return a boolean indicator instead of query matches. In this survey, we focus on reviewing the major techniques for the stream querying problem, while only giving a brief overview on the stream filtering techniques.

3.1 Filtering XML streams

The XML filtering problem arises in the context of *selective dissemination of information* (SDI) [9]. This problem has attracted a lot of research attention. Various versions of filtering algorithms for path pattern queries or TPQs have been proposed [9,22,24,30,39,45,46,52,60,84,74]. Most of these algorithms are based on finite state automata [51].

Finite state automata. A *pushdown automaton* (PDA) is a non-deterministic finite automaton (NFA) equipped with a stack [51]. A PDA's transition is based on the current state, the input symbol, and the symbol at the top of the stack. A *pushdown transducer* (PDT) is an extended PDA that can emit a string in each move [47]. Actions are defined along the transition arcs of the automaton. An output operation is specified as a PDA transition function. A PDT is commonly used to transform data by using a state transition function with an output operation. In the paper, we may use the terms "NFA" and "transducer" to denote "PDA" and "PDT," respectively.

Filtering paradigm. The working paradigm of a filtering algorithm goes as follows. An input query is translated into an automaton by mapping steps of the query to states in the automaton. During execution, a transition from an active state is triggered when an event that matches the transition arrives. Finally, if an accepting state is reached, a value *true* is returned which indicates that the document satisfies the query. Data structures varying from stacks [30,45] to tries [24] and hash tables [9] are used to compute states of the automaton at runtime.

Filtering systems are typically designed to process large number of queries over relatively small size XML documents and are based on the premises that in the SDI context, queries representing user interests share significant commonality. In order to improve the filtering performance, filtering evaluation techniques exploit common subexpressions among query expressions. For instance, *YFilter* [30] and *PrefixFilters* [39] identifies common subquery prefixes, *XPush* [46] computes common query predicates, and *XTrie* [24] finds common substrings, among the input queries.

3.2 Querying XML streams

Several streaming algorithms focus on the querying problem [25,27,43,48,54,66,78,82,85,86]. The majority of these algorithms center on tree-pattern queries (TPQs). TPQs correspond to XPath queries that involve only *child* and *descendant* axes (Sect. 2). Streaming algorithms for TPQs can be extended to process XPath queries with ordered axes (*following*, *following-sibling*, *preceding*, and *preceding-sibling*). We will introduce processing techniques on ordered axes in Sect. 3.6.

Streaming algorithms broadly fall in three categories: the *automaton-based* approach [66,78,82], the *array-based* approach [54], and the *stack-based* approach [25,27,43,48, 85,86].

3.3 The automaton-based approach

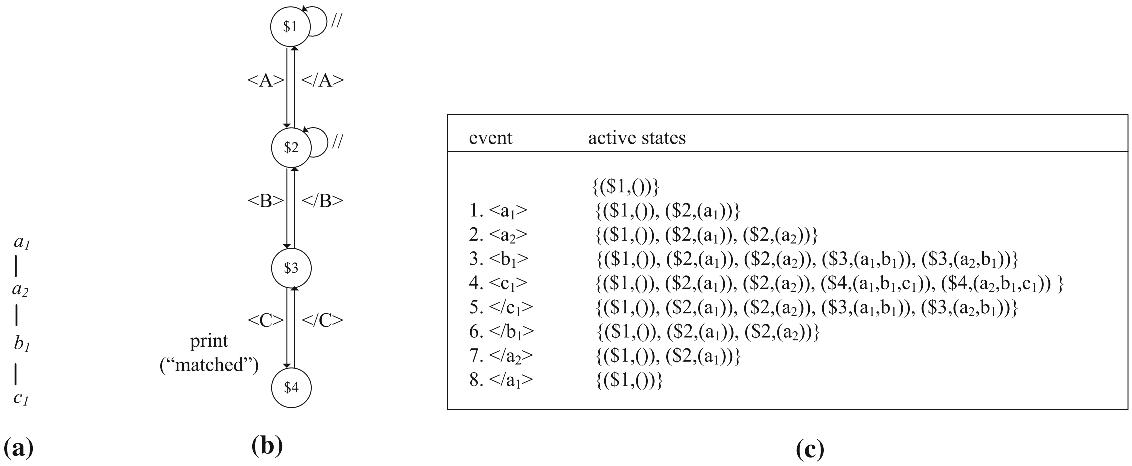
Automata are widely used [31,45,52,66,78,82,81] for pattern retrieval against the XML stream.

A basic evaluation paradigm. A basic automaton-based evaluation paradigm first compiles the input query into a NFA. This NFA is then used to compute pattern matches against the XML stream. Figure 3a shows a NFA for an XPath query, where * denotes a wildcard. The NFA is executed in an event-driven fashion. In order to enable backtracking, a stack is used to store the history of state transitions. Figure 3c shows a snapshot of the runtime stack of the NFA of Fig. 3a while the XML fragment of Fig. 3b is streaming in. When an open event arrives, the NFA looks up its tag in the transition entries of all currently active states (they are at the stack top) and follows all matching transitions. The states that are transitioned to are activated and pushed onto the stack. For example, in Fig. 3c, when the first $\langle a \rangle$ is read, state $0 transitions to both states $1 and $2 and $1 and $2 pushed onto the stack. Active states usually correspond to ancestor matches (See Sect. 2) identified in the data. If a state transitioned to is an accepting state, then a query match has been identified. When a close event is encountered, the NFA performs backtracking by popping the top set of states off the stack. The NFA can be converted to a deterministic automaton (DFA) so as to reduce the time spent on matching transitions to incoming events. The problem of exponential state blow-up in DFA is addressed in [45] by using a *lazy* DFA, which is a DFA constructed in a lazy way by adding new states only when needed.

The basic automaton-based evaluation paradigm is efficient for processing path queries. It is used in [31,45,52] and serves as the core of more sophisticated automaton-based streaming engines for complex XPath queries [66,78,81,82].

XPath streaming engine *XSQ***.** Algorithm *XSQ* [82,83] supports a larger fragment of XPath than the earlier automaton-based algorithms [31,45,52]. The fragment of XPath queries it supports includes *child* and *descendant* axes and predicates with at most one nesting step (i.e., predicates which are either value-based predicates or node tests). *XSQ* uses pushdown transducers as the basic building block for its system design. Each of the transducers corresponds to one step of the input query.

A transducer transforms data using a state transition function with output operations. Besides storing the history of state transitions, the stack of a *XSQ* transducer records also the run-time information of which nodes in the input lead to the current state. Figure 4c shows the run-time states of *XSQ* for evaluating the path pattern query `//A//B/C` (whose pushdown transducer is shown in Fig. 4b) against the data path of Fig. 4a. For instance, when $\langle b_1 \rangle$ is read, a transition is taken from both states ($2,$(a_1)$) and ($2,(a_2)) and results in states ($3,$(a_1, b_1)$) and ($3,(a_2, b_1)). Both pairs of nodes (a_1, b_1) and (a_2, b_1) are matches of the pattern `//A//B` on the input data. Comparing this with the execution of the basic evaluation paradigm for the same query and data shown in Fig. 3, one can see that, besides active states, *XSQ* stores for each incoming open event the set of pattern matches in which the stream node of the event and its ancestor nodes occur. Therefore, the states of a transducer essentially enumerate the patterns that the transducer matches against the input stream. Enumerating and storing pattern matches



event	active states
	$\{(\$1,())\}$
1. <a_1>	$\{(\$1,()), (\$2,(a_1))\}$
2. <a_2>	$\{(\$1,()), (\$2,(a_1)), (\$2,(a_2))\}$
3. <b_1>	$\{(\$1,()), (\$2,(a_1)), (\$2,(a_2)), (\$3,(a_1,b_1)), (\$3,(a_2,b_1))\}$
4. <c_1>	$\{(\$1,()), (\$2,(a_1)), (\$2,(a_2)), (\$4,(a_1,b_1,c_1)), (\$4,(a_2,b_1,c_1))\}$
5. </c_1>	$\{(\$1,()), (\$2,(a_1)), (\$2,(a_2)), (\$3,(a_1,b_1)), (\$3,(a_2,b_1))\}$
6. </b_1>	$\{(\$1,()), (\$2,(a_1)), (\$2,(a_2))\}$
7. </a_2>	$\{(\$1,()), (\$2,(a_1))\}$
8. </a_1>	$\{(\$1,())\}$

Fig. 4 Evaluating the query $//A//B/C$ with *XSQ* [82,83]. **a** An XML data path, **b** the pushdown transducer for the query //A//B/C, **c** the states table of (**b**) on the input of (**a**)

is needed in *XSQ* for handling query predicates as shown below.

The evaluation problem is more complicated for a query with predicates. Recall that in the streaming context, nodes arrive in their pre-order appearance in the XML document tree. Because of the presence of predicates, there might be nodes, at a time, whose membership in the solutions cannot be determined based on the part of the document seen so far (they are *candidate outputs*). Candidate outputs have to be buffered by a streaming algorithm. To this end, *XSQ* augments each transducer with a buffer and a set of buffer operations. For each candidate output, *XSQ* enumerates every possible path that the node matches the query. This path is used to identify each candidate output in the buffer, so that the buffer operation can be correctly carried out. Further, in order to keep track of predicate satisfaction of candidate outputs, *XSQ* creates multiple instances of a transducer, each of which corresponds to a possible combination of predicate results, and connects transducer instances in a hierarchical manner to build an automaton called *hierarchical automaton*.

Figure 5 shows the hierarchical automaton for the query `//A//B[.//C]//D`. Notice that from the automaton state \$5 there is a transition to state \$6 on $\langle /C\rangle$. In state \$6, the automaton starts matching $\langle D\rangle$ again, just as in state \$4. In the general case, if a query node has n predicate children, *XSQ* would need $n!$ state transitions to take care of all the permutations of the n children. The number of transducers in the obtained automaton can thus be exponential in the size of the query. When feeding the XML tree of Fig. 8a to the automaton of Fig. 5, a set of four pattern matches $\{(a_1, b_1, c_1, d_1),\ (a_1, b_2, c_1, d_1),\ (a_2, b_1, c_1, d_1),\ (a_2, b_2, c_1, d_1)\}$ will be constructed and stored by the automaton. Each of these matches corresponds to the same solution d_1 to the query.

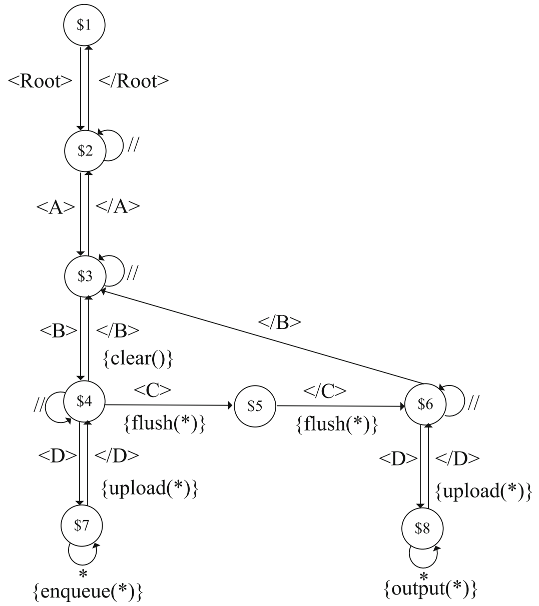


Fig. 5 The hierarchical pushdown transducer for the query $//A//B[.//C]//D$ built by *XSQ* [82,83]

The main advantages of *XSQ* are its clean system design and its straightforward evaluation process. However, *XSQ* needs to explicitly enumerate and store all pattern matches for an input query during its execution. For this reason, *XSQ* suffers from exponential states blow-up and its worst case complexity can be exponential in the size of the query. Let $|Q|$ and $|T|$ denote the size of Q and T, respectively. As shown in [82,83], given a query Q on an XML document T, *XSQ* works in $O(|T| \times 2^{|Q|} \times k)$ time, where k is $O(r^{|Q|})$ in the worst case and r is the recursion depth of Q on T. Also, since a candidate output can participate in multiple matches of the query, *XSQ* might have to buffer multiple copies of the same candidate output at a time. The buffer requirement for possibly multiple copies of candidate outputs together

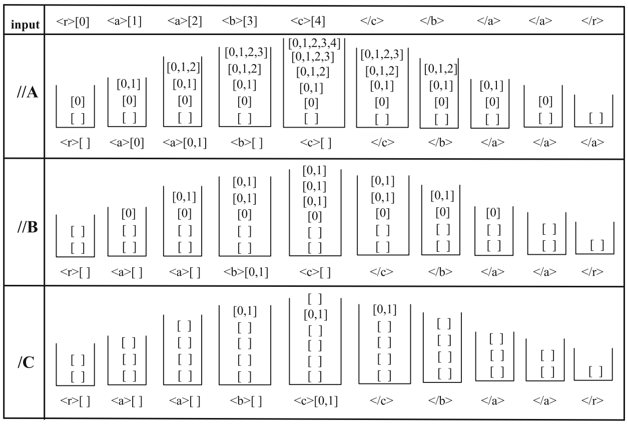


Fig. 6 Evaluating the query $//A//B/C$ with *SPEX* [78,79]

with the need for storing an exponential number of transition states results in the high space cost for *XSQ*.

XPath streaming evaluator *SPEX***.** *SPEX* is chronologically the first polynomial streaming algorithm [78,79]. It is a streaming XPath evaluator which is also based on transducers. *SPEX* translates an input query into a transducer network by generating an independent transducer for each construct corresponding to an XPath subexpression of the input query. Unlike *XSQ*, the number of transducers in a *SPEX* transducer network is linear in the size of the query. Each *SPEX* transducer is associated with a stack of configurations. The stack is used to keep track of the depth of the nodes of the input XML document stream and to store annotations (described below) read from the input stream. This is in contrast to a *XSQ* transducer that uses a stack to enumerate subquery matches against the input XML stream.

SPEX uses a different evaluation strategy than *XSQ*. It assumes a pipeline evaluation paradigm, where each *SPEX* transducer processes an event stream before forwarding it to subsequent transducers. The communication between transducers is realized by annotating stream nodes output from one transducer and providing them as input to the succeeding transducers. A node annotation is expressed as a list of positive integers in ascending order. It marks a state transition and is used by transducers to differentiate matched nodes from unmatched node in the input/output streams. The annotation of each stream node is propagated to its children, descendants, or following siblings etc., depending on the query axis the transducer corresponds to.

Figure 6 shows an example of evaluating the path query `//A//B/C` with *SPEX* on the XML data path of Fig. 4a. Consider, for instance, the execution of the `//A` transducer. When $\langle a \rangle[2]$ is read, the transducer pushes onto the stack the annotation [0, 1, 2], which is the union of the received annotation [2] with the top annotation [0, 1]; then, it outputs $\langle a \rangle$ followed by [0, 1], which is the union of the annotations of the ancestor nodes of $\langle a \rangle[2]$ in the XML stream. The annotated nodes in the output stream of the transducer of the query output node (e.g., the annotated node $\langle c \rangle[0, 1]$ in the output stream of the $/C$ transducer in Fig. 6) represents the query answer.

When the input XPath query has branching predicates, *SPEX* independently evaluates each predicate and the backbone part of the query, and then merges their intermediate results. Specifically, for a transducer with multiple immediate successors (corresponding to a query node with multiple children connected by boolean operators AND and OR), the output stream from the parent transducer is sent simultaneously to each of its child transducers. The output stream from each successor is then composed (by a transducer) to make a single aggregated stream. In order to uniquely identify a stream node which appears in each of the output streams and check whether it satisfies the query predicates, *SPEX* uses a four-phase annotation-mapping process. Each of the four phases is conducted by different transducers. Figure 7 shows an example of processing the query predicate `[./B and.//C]` with *SPEX*. The annotated nodes in the output stream from the fourth transducer denote those input stream nodes that satisfy the predicate. For example, in Fig. 7, nodes a_1 and a_2 satisfy the predicates, since their annotations appear in the annotations of nodes c_1 and b_2 in the output stream.

SPEX can be used to evaluate a restricted form of dag queries called *single-join* dag [79]. A single-join dag is a dag where any two distinct paths share at most one node. To evaluate such a dag query, *SPEX* uses a process similar to that used for queries with predicates. The core idea is to evaluate independently distinct paths that share the same sink node and then intersect the results of these evaluations to produce the matches for the sink node. For a more complex dag query, *SPEX* needs to decompose it to simple subqueries, such as path, tree queries, or single-join dags before doing the evaluation. For a query involving reverse axes (such as *parent* and *ancestor*), *SPEX* rewrites it into disjunctions of path or tree queries [80]. In this case though, the size of the resulting queries can be exponential in the size of the initial query.

Given a query Q (without reverse axes) and an XML document T, let B and h denote the fan-out of Q and the height of T respectively. *SPEX* works in $O(|T| \times |Q| \times h)$ time and uses $O(|T| \times B \times h + |Q| \times h^2)$ space [78].

There are two potential problems of *SPEX*. First, each incoming stream event is processed by the entire transducer network by default. Therefore, the size of messages communicated between each pair of immediately connected transducers in the *SPEX* network reaches the size of the input XML document stream. To reduce the stream traffic, *SPEX* introduces *filter* transducers to the transducer network. A filter transducer filters input stream events and forwards to subsequent transducers only stream events relevant to query evaluation. While effective for selective queries, running

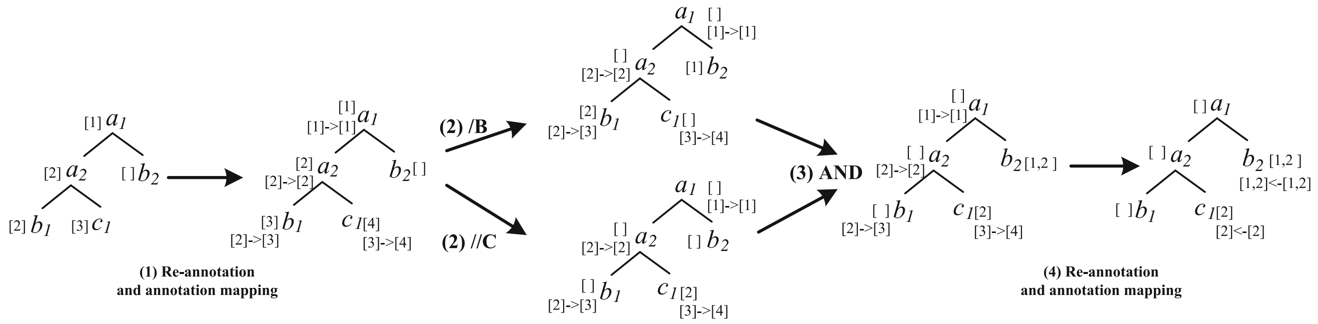


Fig. 7 The four-phase process for evaluating the query predicate $[./B\ and\ .//C]$ in *SPEX* [78,79]: an example

additional filters incurs overhead that may degrade query performance for non-selective queries.

Second, *SPEX* evaluates query predicates inefficiently. For each incoming stream event, it evaluates the query predicates and the backbone part of the query independently, and merges the intermediate results. Because the size of the intermediate results tends to be comparable with the size of the input XML document stream, the overall performance can be negatively impacted.

A double-layered XPath streaming engine. A recent XPath streaming engine presented in [81] uses two NFAs that are organized in two layers (*Layered NFA*) to evaluate XPath queries involving the axes *child*, *descendant*, *following siblings*, and *following* as well as predicates.

The core idea of the Layered NFA method is to separate automaton states that depend on the input XML stream events from those that are static and can be determined at the query compilation stage. To this end, two NFAs are constructed which handle the evaluation of query predicates and the evaluation of the backbone part of the query, respectively. Specifically, this method parses an input XPath query and compiles it into a NFA which constitutes the first layer. The first layer NFA is used to evaluate the backbone part of the query.

Its result is fed into the second layer NFA which handles the query predicates. The second layer NFA is dynamically maintained at runtime using a lazy approach [46]. This means that only those states of the first layer NFA that are accessible for the current node in the XML stream are expanded in the second layer. Matches of query nodes with predicates are maintained in a tree structure called *context node tree*. A node of a context node tree records evaluation results of the predicates of the corresponding query node. It also keeps scope information which indicates the period during evaluation when the node may become a candidate match (See Sect. 2.3). During evaluation, predicate results and candidate outputs will be propagated upwards in the tree so as to check whether context nodes of predicates become candidate matches. This propagation technique is similar to that employed by the stack-based algorithm *TwigM* [27] and will be explained in Sect. 3.5.2. The idea of separating the evaluation of predicates from the rest of the query is also employed by the stack-based algorithm presented in [85,86].

The Layered NFA method needs to keep track of multiple active states and to support multiple transitions from each state. However, it improves over *XSQ* [82,83] in the following two important ways. First, the Layered NFA method dynamically maintains the second layer NFA to avoid generating states from all possible combinations of predicate results. In contrast, *XSQ* constructs $O(2^{|Q|})$ transducers beforehand to differentiate between predicates that have been satisfied and predicates that are unsatisfied and wait to be evaluated. Second, in order to avoid the exponential growth in the state size when a query contains the *descendant* or *following* axes, the Layered NFA method introduces state sharing and state pruning optimization techniques. As shown in [81], thanks to these two improvements, the Layered NFA method is able to achieve $O(|T| + |Q| \times h)$ space and $O(|T| \times |Q|)$ time complexity, respectively.

An approach presented in [18] also uses separate NFAs to handle the evaluation of query predicates and the evaluation of the rest of an XPath query. It decomposes an XPath query into its backbone path and a set of predicate paths and constructs for each query path a NFA. Unlike the Layered NFA method, in this approach, all SAX events are regarded as input for both the backbone NFA and the predicate NFAs. The approach is based on reducing every axis to three primitive axes: *firstchild*, *nextsibling*, and *self* using the query rewriting rules in [40]. Accordingly, the input SAX event stream is transformed into a SAX event stream that uses first-child and next-sibling relations to represent the XML tree. Because of these transformations, the approach only needs to handle four types of state transitions during query evaluation.

Summary. The theory of finite automata is the root of all the XML streaming evaluation techniques. Automaton-based streaming algorithms adapt the mature techniques of automata—typically used for matching patterns over strings—to the query evaluation process. This often makes for a natural design for a streaming system. Many XML streaming prototypes have been built using automaton-based evaluation techniques. Automaton-based algorithms are generally efficient for stream processing of simple path queries, but their extensions to handle TPQs (or XPath queries with predicates) turned out to be hard. The reason is that, by undertaking query evaluation at the granularity of stream tokens, the automaton-based approach fails to completely capture the tree-structured nature of the XML data [63]. A typical problem with automaton-based algorithms is that they all require to compute automata states and the corresponding transition tables at runtime. Further, they generally spend a significant amount of time matching transitions to incoming events. The experimental evaluation results in [27,43] show that the performance of automaton-based streaming algorithms such as *XSQ* cannot be compared with that of the stack-based streaming algorithms introduced later.

3.4 The array-based approach

We start with algorithms that assume tree-pattern queries and then continue with algorithms that consider a more general class of queries.

Streaming algorithms for tree-pattern queries. Since array-based approaches do not translate the given query to finite state automata they do not have to explicitly compute states and the corresponding transition tables. *TurboXPath* [54] is a representative of the array-based approach. *TurboXPath* first builds a parse tree for a given query and then finds matches of the parse tree nodes on the input XML stream.

The algorithm uses mainly an array data structure, denoted as Work Array (WA), to record the matching status during evaluation. Each WA entry has four fields: (1) a pointer to the corresponding parse tree node; (2) the level of the corresponding document node; (3) a set of references (pointers) between parent and child entries; (4) a status flag that is used during evaluation to indicate whether the corresponding document node has satisfied the query conditions. The WA is dynamically maintained by the algorithm. At any point during the evaluation, the WA stores entries corresponding to nodes in the part of the XML document already read provided they are relevant to the query processing.

Figure 9 shows an illustration of *TurboXPath* evaluating the TPQ of Fig. 8b on an XML document of Fig. 8a. In the figure, only fields (1) and (4) for a WA entry are shown for clarity. The algorithm works by trying to match each incoming (open or close) event with all the entries in the WA. A match occurs when the corresponding document node is an ancestor match (see Sect. 2) of the corresponding parse tree node X of a WA entry. When a match is found, new entries are inserted into the WA at the open events and old entries are removed from the WA at the close events. More specifically, when an open event arrives, for each child Y of the parse tree node X of the matched WA entry, a new child array entry is added to the WA. For instance, in Fig. 9, when $\langle a_1 \rangle$ is read, an entry for the parse tree node B is added to the WA. If node X is a leaf node in the parse tree, the status flag of the matched WA entry is set to *true*. For instance, the status of all the C entries in the WA are set to *true* when $\langle c_1 \rangle$ is read (Fig. 9). When a close event arrives, the following two actions are performed: (1) the status of a matched WA entry is evaluated by checking the status of its child WA entries; (2) the child WA entries of the matched WA entry are removed from the WA.

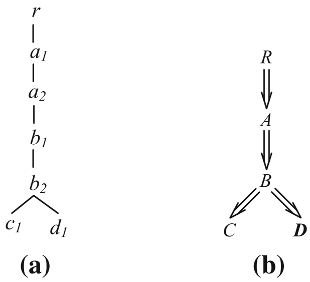
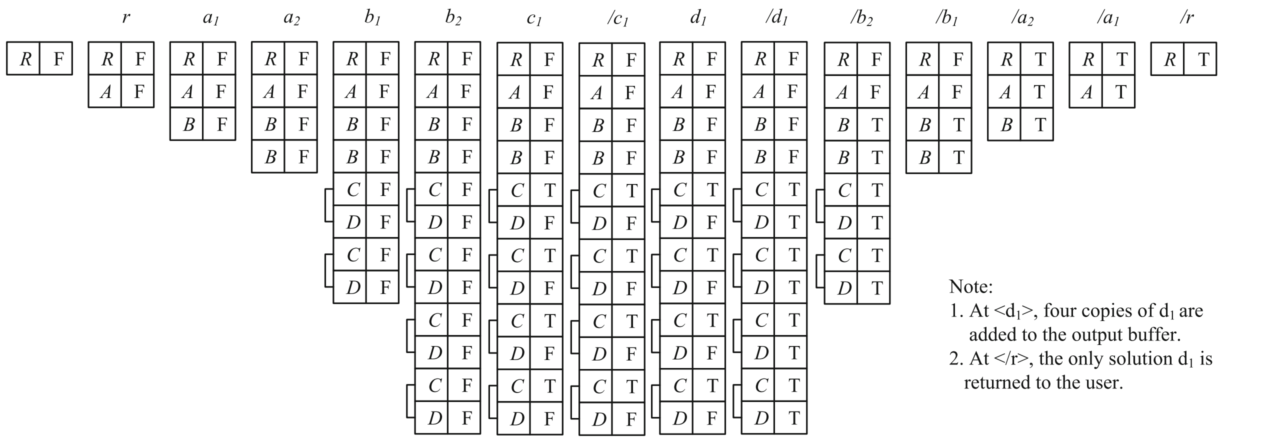


Fig. 8 **a** XML tree, **b** The parse tree for the query $//A//B[.//C]//D$. The node D, shown in bold, is the output node

TurboXPath supports TPQs with multiple output nodes. The query answer is a sequence of tuples of XML fragments matching the output nodes. All the candidate matches that may participate in result tuples are stored in buffers. In order to construct tuples, *TurboXPath* uses a nested-loop join algorithm which computes all the possible combinations of buffered candidate matches.

Algorithm *TurboXPath* works efficiently for queries on non-recursive XML documents. However, it exhibits exponential behavior for queries on recursive XML documents [48]. Recall that *TurboXPath* scans *all* the entries in the WA in order to find a matching entry for *each* incoming event and inserts into the WA a set of new child entries for each matched WA entry. When multiple document nodes match the parse tree node of a matched WA entry, multiple copies of the same set of child entries for that entry can coexist in the WA. In the example of Fig. 9, the WA contains two B entries after the $\langle a_1 \rangle$ event and before $\langle /a_2 \rangle$ event because both a_1 and a_2 match the WA entry for A. Each of the two B entries adds two pairs of C and D entries to the WA because they are matched by both b_1 and b_2. Clearly, the large number of duplicate WA entries for the same query node incurs significant overhead to the evaluation.

Given a query Q on an XML document T, the size of the WA (number of entries) is, in the worst case, $O(r^H)$ where r and H denote, respectively, the recursion depth of Q on T and the height of Q. Since for each XML document



	r	a_1	a_2	b_1	b_2	c_1	$/c_1$	d_1	$/d_1$	$/b_2$	$/b_1$	$/a_2$	$/a_1$	$/r$
R F	R F	R F	R F	R F	R F	R F	R F	R F	R F	R F	R F	R T	R T	R T
	A F	A F	A F	A F	A F	A F	A F	A F	A F	A F	A F	A T	A T	
		B F	B F	B F	B F	B F	B F	B F	B F	B T	B T	B T		
			B F	B F	B F	B F	B F	B F	B F	B T	B T			
				C F	C F	C T	C T	C T	C T	C T				
				D F	D F	D F	D F	D F	D T	D T				
				C F	C F	C T	C T	C T	C T	C T				
				D F	D F	D F	D F	D F	D T	D T				
					C F	C T	C T	C T	C T					
					D F	D F	D F	D F	D T					
					C F	C T	C T	C T	C T					
					D F	D F	D F	D F	D T					

Note:
1. At <d_1>, four copies of d_1 are added to the output buffer.
2. At </r>, the only solution d_1 is returned to the user.

Fig. 9 Evaluating the query of Fig. 8b on the XML tree of Fig. 8a using Algorithm phTurboXPath [54]: state of WA after each incoming event has been processed

node, *TurboXPath* needs to scan the WA to locate matches, the total running time of *TurboXPath* is $O(|T| \times r^H)$. The time complexity of *TurboXPath* improves the $O(|D| \times r^{|Q|})$ time complexity of the automaton-based algorithm *XSQ* [82] (notice that $H = O(log|Q|)$), as *TurboXPath* does not need to enumerate all possible combinations of query predicate results during evaluation. However, the size of the WA can still grow exponentially on the height of the query tree on recursive XML documents. This could impact negatively the time performance of *TurboXPath*on recursive XML document streams.

Filtering algorithms for tree-pattern queries that are based on the array-based approach which are similar to *TurboXPath* are presented in [11,13]. In [12], an array-based streaming algorithm is given that works for tree-pattern queries on non-recursive XML documents.

A streaming algorithm for tree-pattern queries extended with reverse axes. In [14], a streaming algorithm called X_{aos} is presented which supports an extension of tree-pattern queries with reverse axes (parent and ancestor). The class of queries supported by Algorithm X_{aos} belongs to the fragment of XPath $XP^{\{[],/,//,*,\backslash,\backslash\backslash\}}$. Algorithm X_{aos} is an array-based algorithm which extends Algorithm *TurboXPath*.

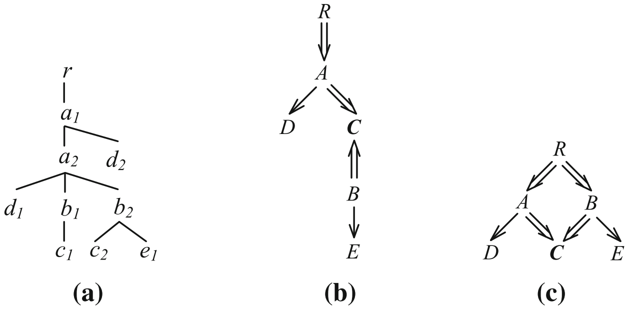


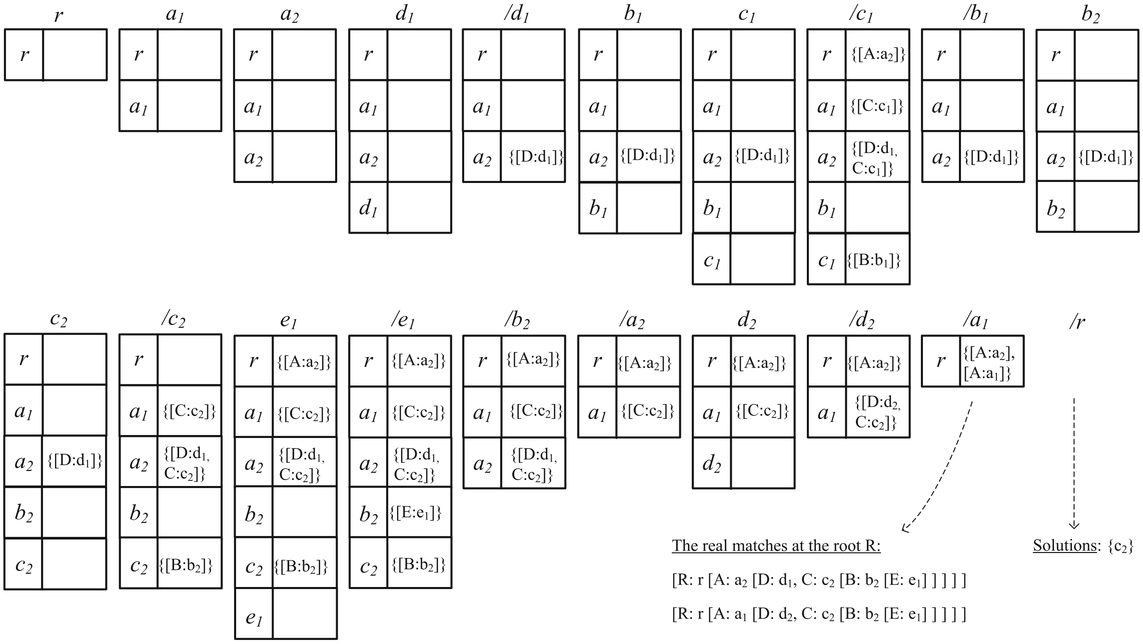
Fig. 10 **a** XML tree, **b** The parse tree for the query $//A[D]//C[\backslash\backslash B/E]$, **c** The dag for the same query. The node C shown in bold is the output node

For an input query, it not only builds a parse tree, but also constructs a directed acyclic graph (dag) in which all reverse axes are converted into their symmetrical forward axes. Figures 10b and c show the parse tree and the dag for an XPath expression. Note that in the parse tree of Fig. 10b, the edge from node B to node C denotes an *ancestor* edge. Algorithm X_{aos} uses the dag to determine whether an XML document node is an ancestor match of a dag node while it constructs real matches based on the parse tree. Figure 11 illustrates the evaluation process of Algorithm X_{aos} on the query and the XML tree of Fig. 10.

At an open event, if the corresponding XML document node is an ancestor match of a dag node, Algorithm X_{aos} creates and adds to WA a new entry called *matching-structure* [14]. A matching-structure consists of three fields: (1) a parse tree node X; (2) an XML tree node matching node X; and (3) a submatching for each child of X in the parse tree. A submatching at child Y of X is a set of matching-structures at Y.

Algorithm X_{aos} utilizes a technique called *propagation* for a matched WA entry at a close event. Recall that a match occurs when the corresponding document node of the WA entry is an ancestor match of a dag node. A matched entry e is propagated to each of its parent entries to become their submatching. The parent entries of e refer to the matching-structure at the parent parse tree node of e. More specifically, for each matched entry e at an incoming close event, Algorithm X_{aos} determines whether e is a real match by checking whether all its child entries are real matches. If this is the case, entry e is propagated to its parent entries. In Fig. 11, after $\langle /d_1 \rangle$ is read, entry d_1 is propagated to its parent entry a_2 as its submatching.

When *ancestor* and *parent* edges are present in the parse tree, X_{aos} is unable to determine conclusively whether a



The real matches at the root R:

[R: r [A: a2 [D: d1, C: c2 [B: b2 [E: e1]]]]]

[R: r [A: a1 [D: d2, C: c2 [B: b2 [E: e1]]]]]

Solutions: {c2}

Fig. 11 Evaluating the query of Fig. 10b on the XML tree of Fig. 10a using Algorithm X_{aos}[14]

matched entry e is a real match at its close event. The reason is that the close event of a child entry of e (which corresponds to a parent or ancestor node of e) has not arrived yet and consequently X_{aos} is unable to determine at this time whether that child entry is a real match. In this case, X_{aos} *optimistically* propagates all the child entries to entry e assuming these entries are real matches. In the example of Fig. 10b, query node C has an outgoing *ancestor* edge. As shown in Fig. 11, after $\langle /c_1 \rangle$ is read, the child entry b_1 of c_1 is propagated into the submatching of c_1. Subsequently, entry c_1 is propagated to its parent entries a_1 and a_2. Since at that time all the submatchings of entry a_2 are non-empty, entry a_2 is identified as a real match and is propagated to its parent entry r. A similar process is followed after $\langle /c_2 \rangle$ is read. If at some point, it can be determined conclusively that an entry is not a real match, X_{aos} reverses the previous optimistic propagation by removing the entry from the submatching of all its parent entries. This undo operation is cascaded to all affected WA entries. In Fig. 11, after $\langle /b_1 \rangle$ is read, b_1 can be identified as a non-real match since its submatching for E is empty. Therefore, entry b_1 is removed from its parent entry c_1. The undo propagation is then recursively applied to entries c_1, a_1, a_2, and r. Clearly, such an undo operation affects negatively the time performance of X_{aos}.

After all the XML document nodes have been processed, Algorithm X_{aos} produces query answers by traversing the matching-structure of the query root and projecting the matches for the output node of the input query. In Fig. 11, after the close event for the document root r is read, two matching-structures are traversed and a single solution c_2 is produced (the same solution c_2 occurs in the two matching-structures).

Summary. Originated from the theory of finite automata, the array-based approach [54] avoids the expensive steps of translating queries to automata and computing transition states of the automaton-based approach. Nevertheless, it has three major limitations: (1) it enumerates and records in memory all the query node matches and then iterates on each of the matches for every incoming event during evaluation. These operations incur exponential memory usage and time complexity for queries on recursive XML documents; (2) it can store multiple copies of the same answer node in memory. For example, as illustrated in Fig. 9, *TurboXPath* stores in memory four copies of the answer node d_1. Also, as shown in Fig. 11, X_{aos} constructs two matching-structures at the query root corresponding to the same solution c_2. As a result, the array-based approach needs an additional process to eliminate duplicate answers at the final stage; and (3) it does not deliver query answers until the entire document stream is processed. In the case of an unbounded stream, the evaluation may be unnecessarily postponed. Because of these limitations, this type of processing is inefficient and not viable for applications that require incremental outputs and process streams that are unbounded and/or have recursive structures.

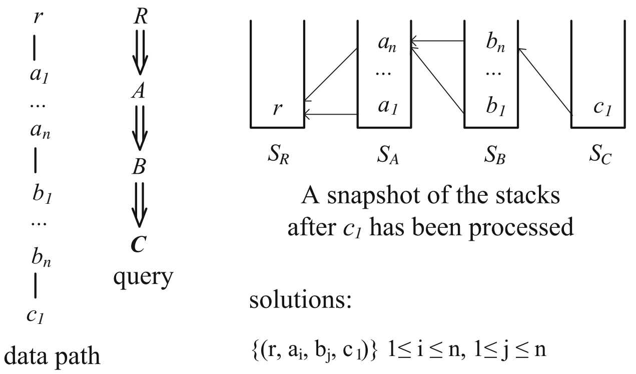


Fig. 12 The path stack technique: an example

3.5 The stack-based approach

Algorithm *PathStack* [21] was designed to evaluate path queries on preprocessed XML data. A number of more recent algorithms [27,43,48,85,86] exploit its ideas in order to evaluate TPQs on streaming XML data. These stack-based algorithms extend *PathStack* to compactly encode query pattern matches in a chain of stacks. This technique avoids the enumeration and explicit storage of the query pattern matches during the evaluation. The stack-based algorithms evaluate TPQs on XML streams in polynomial time and space, and this is a significant improvement over the automaton-based and the array-based approaches.

3.5.1 The path stack technique

Given a path pattern query Q and an XML tree T, the path stack technique [21] creates a stack for each query node of Q and uses these stacks to compactly represent all possible embeddings of Q in a tree path of T.

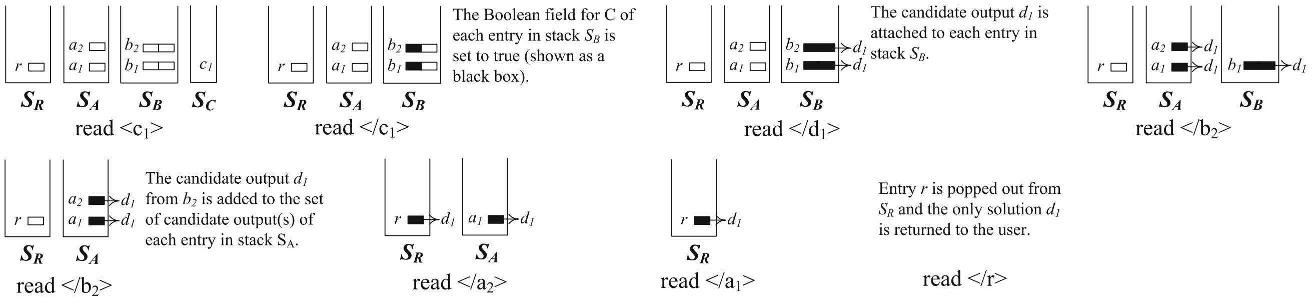
Figure 12 shows an example of the evaluation of a path query over an XML tree path using Algorithm *PathStack* [21]. Tree nodes are read in their pre-order appearance in the tree and are pushed into their corresponding query stacks. Entries in the stack correspond to tree nodes. The entries below an entry in a stack correspond to nodes in the XML tree path that are ancestors of that node. In addition, each entry e in a stack has a pointer to an entry in its parent stack which is the highest entry in that stack among the entries corresponding to ancestors of e in the XML tree. These pointers are used to construct query solutions. Solutions (in the form of tuples) are generated by following the pointers of the stack entries once a tree node for the leaf query node is pushed into its stack. An important feature of such a stack-based organization is that it encodes a potentially exponential number of solutions in a linear space. In the above example, a total number of n^2 solutions is encoded using $2n+2$ entries in the stacks.

In order to evaluate a TPQ Q against an XML document stream T, the stack-based approach extends the path stack technique in the following two ways. First, each node in Q can now have multiple child nodes. In order to keep track of whether a stack entry has satisfied its branching predicates, the approach associates each stack entry with a boolean array indexed by its child nodes (seen as branching predicates), and employs a procedure to determine when an entry satisfies/fails a predicate. Second, since the answer of Q now consists of the set of matches of the output node of Q on T (and not of a set of tuples), instead of computing all the embeddings of Q on T, the stack-based approach utilizes a propagation technique to maintain a set of possible output node matches (i.e., candidate outputs). Maintaining candidate outputs is an important and complex task of a stack-based algorithm which affects its efficiency. Below, we describe five representative stack-based streaming algorithms.

3.5.2 Stack-based streaming algorithms for TPQs

Algorithm *TwigM*. Algorithm *TwigM* [27] is a stack-based streaming algorithm for evaluating TPQs. It is a *lazy* algorithm because it evaluates query predicates only at close events. Figure 13 shows an example of the evaluation of the query of Fig. 8b against the XML tree of Fig. 8a using *TwigM*.

Let Q and T denote the input query and an XML tree, respectively. At every open event of T, *TwigM* determines whether the corresponding XML document node x is an ancestor match of a query node X of Q. It does so by checking the existence of a *parent* entry y of x, that is, an entry y in



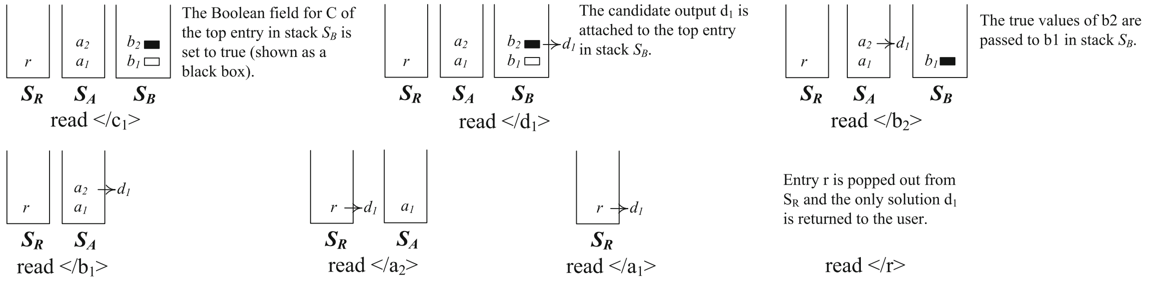
The Boolean field for C of each entry in stack S_B is set to true (shown as a black box).

The candidate output d_1 is attached to each entry in stack S_B.

The candidate output d_1 from b_2 is added to the set of candidate output(s) of each entry in stack S_A.

Entry r is popped out from S_R and the only solution d_1 is returned to the user.

Fig. 13 Evaluating the query of Fig. 8b on the XML tree of Fig. 8a using Algorithm *TwigM* [27]



The Boolean field for C of the top entry in stack S_B is set to true (shown as a black box).

The candidate output d_1 is attached to the top entry in stack S_B.

The true values of b2 are passed to b1 in stack S_B.

Entry r is popped out from S_R and the only solution d_1 is returned to the user.

Fig. 14 Evaluating the query of Fig. 8b on the XML tree of Fig. 8a using Algorithm *LQ* [43]

the stack of X's parent Y such that the structural relationship between x and y satisfies the structural relationship between X and Y in Q. If it is the case, a new entry for x is created and pushed onto the stack of X. At every close event, stack entries which match that event are popped out from their stacks. For each popped entry x which is a candidate match (see Sect. 2) of a non-root query node X, *TwigM* sets the field for X in the boolean array of every parent entry y of x to true. This denotes that a candidate match to the child node X has been found. *TwigM* also propagates the candidate outputs associated with x (if any) to y. Duplicate candidates are eliminated. If entry x is not a candidate match, it is simply discarded. In the example of Fig. 13, when $\langle/b_2\rangle$ is read, the candidate output d_1 (associated with b_2) is propagated to both a_1 and a_2 in stack S_A. Similarly, when $\langle/b_1\rangle$ is read, the candidate output d_1 (associated with b_1) is propagated to both a_1 and a_2 and is eliminated because it is already associated with them. If X is the query root of Q, the candidate outputs associated with the candidate match x are returned as solutions to the user.

Algorithm *TwigM* works in $O(|T| \times |Q| \times (|Q| + B \times h))$ time and uses $O(|T| \times h + |Q| \times r)$ space in the worst case, where B, h, and r denote the fan-out of Q, the height of T, and the recursion depth of Q on T, respectively. Clearly, it improves over the array-based algorithm *TurboXPath* because it avoids explicitly enumerating and storing query pattern matches during execution. Nevertheless, *TwigM* has the following limitations: (1) it may have to examine exhaustively a large number of stack entries for each incoming event. In particular, at a close event, it needs to propagate the matching information and candidate outputs to multiple stack entries. In Fig. 13, when $\langle/d_1\rangle$ is read, all the entries in stack S_B (b_1 and b_2) are accessed. For each of them, the boolean array field for D is set to *true* and d_1 is uploaded as a candidate output. (2) It may have to store multiple physical copies of a candidate output at a time in different stack entries. As a result, it needs to eliminate redundant candidate outputs in order to avoid duplicate solutions. Figure 13 shows one such case of duplicate elimination.

Algorithm *LQ*. In [43], a stack-based algorithm called *LQ* was presented to address the limitations of *TwigM*[27]. Algorithm *LQ* also works in a lazy fashion. Figure 14 shows the evaluation of the query of Fig. 8b against the XML tree of Fig. 8a using *LQ*. Notice that the boolean array of each stack entry is now indexed only by the branching child nodes (seen as branching predicates) of the corresponding node instead of all the child nodes as in *TwigM*.

At each open event, *LQ* exploits better the features of the stack-based organization and determines the existence of a parent entry by examining *only* the structural relationship between the XML document node of the event and the top entry of the corresponding parent stack. At every close event, for each corresponding entry x that is a candidate match of a query node X, Algorithm *LQ* visits at most two entries. One is the top entry in the parent stack of X. It is visited for recording the matching information (in its boolean array) if X is a branching node, or for uploading the candidate outputs associated with x if X is a backbone node. The other is the entry below x (if any) in the stack of X. It gets the matching information of x if X is related to its parent with a descendant relationship. In Fig. 14, when $\langle/b_2\rangle$ is read, the candidate output d_1 associated with b_2 is attached to a_2 in stack S_A. Also, the truth value for C in the boolean array of b_2 is transferred to b_1 in stack S_B.

In the case that entry x is not a candidate match of X (which implies that X is a backbone node), at most one entry is visited. When X is linked with a descendant edge to its backbone child node, the candidate outputs associated with x are propagated to the entry below x in stack S_X. Otherwise, these candidate outputs are propagated to the top entry of the first descendant node of X which is linked with a descendant edge to its backbone child node. Figure 16 shows an example. When $\langle/b_2\rangle$ is read, the candidate output c_1 associated with b_2 is attached to a_2 in stack S_A. Subsequently, when $\langle/a_2\rangle$ is read, since a_2 is not a candidate match of A (the branching predicate D is not satisfied) and the edge between A and B in the query is $//$, the candidate output c_1 is propagated to b_1 in stack S_B.

The proper propagation of candidate outputs allows *LQ* to avoid unnecessarily accessing many query matches and storing multiple copies for each candidate output. These improved propagation techniques are also used in the

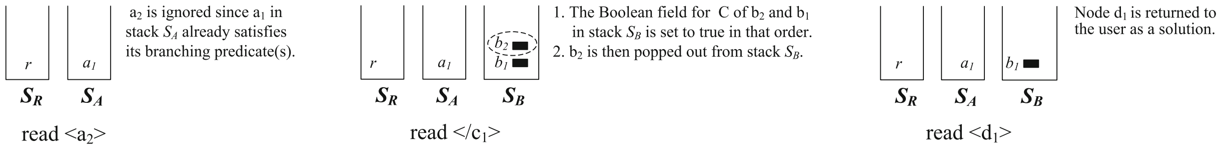

r
a1
SR
SA
a2 is ignored since a1 in stack SA already satisfies its branching predicate(s).
read <a2>
r
a1
b2
b1
SR
SA
SB
1. The Boolean field for C of b2 and b1 in stack SB is set to true in that order.
2. b2 is then popped out from stack SB.
read </c1>
r
a1
b1
SR
SA
SB
Node d1 is returned to the user as a solution.
read <d1>

Fig. 15 Evaluating the query of Fig. 8b on the XML tree of Fig. 8a using Algorithm *EQ* [43]

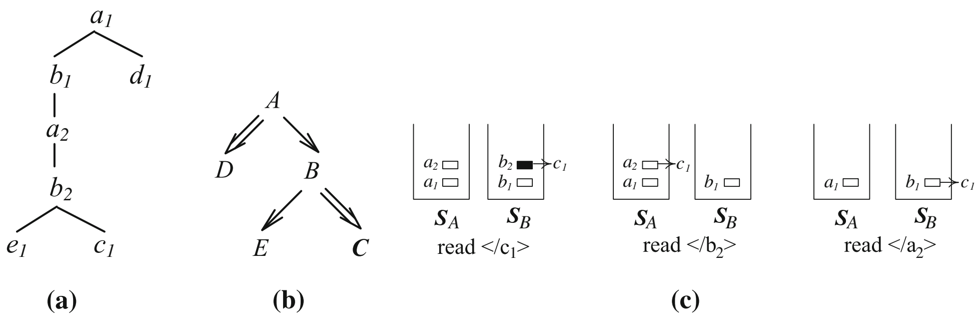


Fig. 16 The propagation technique used by Algorithm *LQ* and *EQ* [43] in the presence of child edges: an example. **a** Data tree, **b** query, **c** snapshot of the runtime stacks

stack-based streaming algorithms presented in [103] and in [85,86] for queries with only descendant edges. Another improvement realized by *LQ* is the exploitation of the existential semantics of the queries during the evaluation. For instance, consider again evaluating the query of Fig. 8b against the XML tree of Fig. 8a. Suppose that c_1 has many following siblings matching query node C. Those siblings will not be processed and stored by *LQ* since c_1 has already satisfied the branching predicate C of node B in the query. All these improvements together make *LQ* achieve $O(|T| \times |Q|)$ time performance and $O(|T| + |Q| \times r)$ space performance. Note that the algorithms of [85,86] use a more complicated procedure than that of *LQ* to evaluate queries with child edges. This results in their worse complexity as shown in Fig. 20.

Algorithm *EQ***.** Based on *LQ*, another stack-based streaming algorithm, called *EQ* [43], was developed that further optimized the space performance of *LQ*. Algorithm *EQ* is an *eager* streaming algorithm which eagerly evaluates branching predicates before their close events arrive. Figure 15 shows an example of an evaluation of the query of Fig. 8b against the XML tree of Fig. 8a using *EQ* (Fig. 16).

The key principle of an eager streaming algorithm is that when a branching predicate is satisfied, the algorithm checks whether the candidate outputs within the scope of the newly satisfied predicate become solutions. Specifically, whenever a branching predicate of a node becomes true, *EQ* checks whether the branching predicates of all the ancestor nodes also become true. During the checking process, matches that are not useful for determining solutions (*redundant matches*) are discarded. Candidate outputs are returned to the user as soon as they can be determined to be solutions. Also, *EQ* stores at open events only nodes that possibly contribute to new solutions. In Fig. 15, when $\langle a_2 \rangle$ is read, a_2 is not stored in its stack since entry a_1 has trivially satisfied its branching predicates (node A has no predicate child nodes). When $\langle c_1 \rangle$ is read, the predicate for C in the boolean array of b_2 in stack S_B is evaluated to *true*. Subsequently, the predicate for b_1 is also evaluated to *true*. Then, b_2 is determined to be redundant and is discarded. Intuitively, a redundant match (like b_2) at the top of its stack might hide a "good" match (i.e., one that has satisfied its predicates, like b_1 here) below it. Storing a redundant match would not only cost memory space but also delay the return of solutions. Finally, when $\langle d_1 \rangle$ is read, d_1 is determined to be a solution and is returned to the user right away. Notice that, during the evaluation, *EQ* stores *zero* candidate outputs, compared with one for *LQ* and two for *TwigM*.

Similar eager evaluation strategies as described previously were also used by the streaming algorithm presented in [85, 86]. The eager evaluation can save substantial memory space and provide better query response time. First, storing query node matches can be partially avoided. Second, it can be determined earlier which candidate outputs can be returned to the user as solutions (even without buffering them). Experimental results in [43] show that among the tested algorithms (*XSQ* [82], *TwigM* [27], *LQ*, and *EQ*), *EQ* has the best space performance without trading off the time performance of *LQ* (which shows the best time performance).

Algorithm *StreamTX***.** In [48], a streaming algorithm called *StreamTX* is presented which supports TPQs with multiple output nodes. *StreamTX* adapts the twig join algorithm *TwigStack* [21] so that it can compute TPQs over streaming XML data. Algorithm *TwigStack* is a holistic algorithm that represents the state of the art for evaluating TPQs over XML data that are preprocessed (each node is assigned a positional representation) and stored in a database. In order to find all the pattern matches of a TPQ over an XML tree, *TwigStack* first decomposes a given TPQ Q into multiple root-to-leaf path patterns and computes solutions to those path patterns using the aforementioned path stack technique. Then, it merge-joins solutions of the path patterns to produce the TPQ answer. When all edges in Q are descendant relationships, $TwigStack$ is CPU and I/O optimal for computing Q [21]. There is one issue that needs to be addressed for adapting $TwigStack$ to the streaming evaluation: at each evaluation step, $TwigStack$ needs to access multiple input XML tree nodes in order to determine whether the node under consideration should be stored in its stack. However, in the streaming environment, the document is sequentially scanned and each tree node can only be accessed in the preorder appearance of the node in the XML tree. As a consequence, when a determination needs to be made about the storage of a node e in its stack, the relevant nodes may have not arrived yet (node e is characterized as *blocked*). For instance, consider evaluating the query of Fig. 8b against the XML tree of Fig. 8a. When $\langle a_1 \rangle$ is read, $TwigStack$ needs the presence of b_1 and c_1 in order to determine if a_1 should be pushed onto its stack. However, none of b_1 and c_1 has arrived yet. Clearly, it would not be efficient for *StreamTX* to buffer a_1, a_2, b_1, and b_2 before it reaches c_1 and start its matching process by accessing the buffered nodes. Instead, *StreamTX* uses a blocking technique which attempts to continue the evaluation process with non-blocked nodes which have been buffered. Nevertheless, in the worst case, *StreamTX* still has to buffer all the incoming tree nodes in memory. In order to minimize the buffering space, it uses two optimization techniques. The first one buffers at open events only nodes that are ancestor matches. The other one prunes at close events non-candidate match nodes from buffers.

Experimental results in [48] show that *StreamTX* has superior performance over the array-based algorithm *TurboXPath* [54] on queries with multiple output nodes. However, the time and memory performance of *StreamTX* might be inferior to algorithms *LQ* and *EQ* [43] for the following reasons: (1) *StreamTX* needs to compute all the pattern matches of the given query. Those pattern matches are projected over output nodes to produce the final answer. As multiple pattern matches may contribute to the same solution, like the array-based approach, an additional process is needed to eliminate duplicate solutions at the final stage. In contrast, *LQ* and *EQ* compute query answers without enumerating query pattern matches; and (2) there is an overhead in *StreamTX* associated with finding non-blocked nodes for each incoming open event during evaluation.

Algorithm ***Twig*^{2}*Stack*.** A twig join algorithm called *Twig*^{2}*Stack* presented in [25] can be viewed as a stack-based streaming algorithm. Unlike the original (non-streaming) stack-based algorithm *TwigStack* [21], *Twig*^{2}*Stack* does not decompose the input TPQ into root-to-leaf path patterns. Therefore, it does not need to merge-join path solutions to produce the query solutions. Instead, it evaluates the input TPQ against XML data in a bottom-up way. The bottom-up evaluation entails a post-order document traversal. However, in the streaming context, nodes arrive in the pre-order appearance in the document tree. In order to address this discrepancy, *Twig*^{2}*Stack* maintains during the evaluation a global stack for the nodes that are on the same path of the XML tree. Solutions to twig patterns (subpatterns of the input TPQ) are incrementally encoded using a proposed hierarchical stack encoding scheme. Figure 17 shows an example of the hierarchical stack encoding. A node above another node in a stack is an ancestor node. A pointer from a node in a stack to a node in another stack indicates that the former node is an ancestor of the later one. Query solutions encoded in the hierarchical stacks are enumerated at the final stage. Algorithm *Twig*^{2}*Stack* needs to store all the relevant data in memory for its matching process. As such, it is categorized by [48] as an in-memory evaluation algorithm.

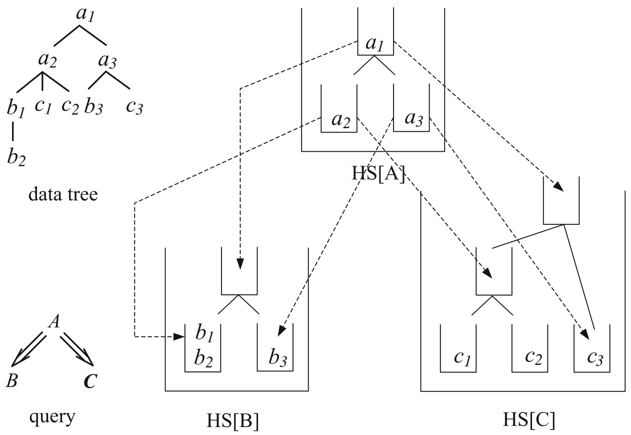


Fig. 17 The hierarchical stack encoding used by Algorithm *Twig*^{2}*Stack* [25]: an example

3.5.3 Streaming algorithms for partial tree-pattern queries

Existing XML streaming algorithms focus almost exclusively on tree-pattern queries (TPQs). Requirements for flexible querying of XML data have motivated recently the consideration of classes of queries that are more general and flexible than TPQs. These queries could not be supported by existing algorithms.

Partial tree-pattern queries. A recent XML streaming paper [101] considers the class of partial tree-pattern queries (PTPQs) which generalizes and strictly contains TPQs, TPQs with reverse axes [14,78] and dag queries [14,79]. PTPQs are not restricted by a total order for the nodes in the paths of a tree pattern. They can restrict nodes to lie on the same path even without specifying structural relationships between them, while permitting some or all of these nodes to be shared by multiple paths. PTPQs can express XPath queries with the

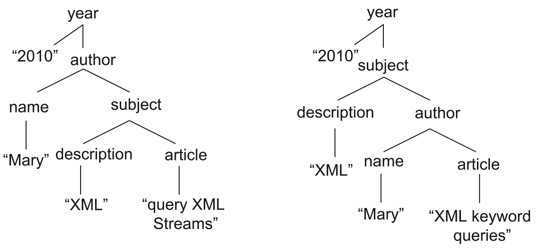


Fig. 18 Fragments of two XML bibliography documents

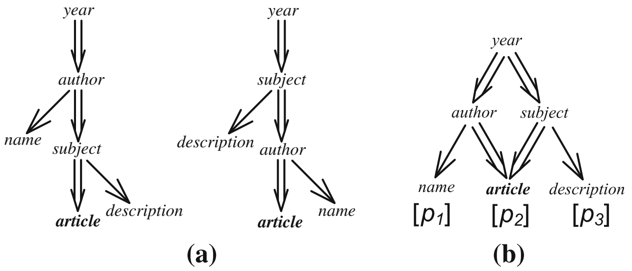


Fig. 19 **a** Two TPQs and **b** a PTPQ for querying the XML bibliography of Fig. 18

reverse axes parent and ancestor along with the *is-same-node* node equality operator of XPath. These constructs are important since they provide a lot of flexibility for querying XML data sources with different structures or for querying a single XML data source without fully knowing its structure.

As an example, consider two streaming XML documents containing bibliographic information from two different data sources that organize articles differently, one grouping them by *year* and *author*, and the other grouping them by *year* and *subject*. Figure 18 shows fragments of these two XML documents. Suppose that we want to find articles on the subject "XML" authored by "Mary" in year "2010." This requirement can be expressed by a single PTPQ which is graphically represented as an annotated directed acyclic graph (dag) shown in Fig. 19b. The annotations [p_1], [p_2] and [p_3] indicate that the annotated nodes and their ancestor nodes belong to paths p_1, p_2 and p_3, respectively. In contrast, two TPQs (shown in Fig. 19a) are needed to retrieve the same information from the two XML documents. It is not possible to retrieve the same information from both documents using a single TPQ. The reason is that TPQs impose *a total order* for the nodes in every one of their paths. It is not possible in a TPQ to indicate that two nodes, say *author* and *subject*, occur in a path without specifying a precedence relationship between them: node *author* has to precede node *subject* or vice versa. It is shown that a PTPQ is equivalent to a *set* of TPQs [101]. However, the number of these TPQs can be exponential on the size of the PTPQ. Clearly, PTPQs cannot be evaluated efficiently using efficient streaming algorithms for TPQs, since an exponential number of them might need to be executed. Therefore, ad hoc algorithms are needed for the efficient evaluation of PTPQs.

PTPQs were initially introduced in [93]. Evaluation algorithms for partial path queries (PTPQs with a single "path") were presented in [90,97,100]. Partial path queries are not a subclass of TPQs but they form a subclass of PTPQs. Evaluation algorithms for PTPQs on persistent XML data were presented in [98,99].

Streaming algorithms for PTPQs. Two efficient streaming algorithms for PTPQs, *PSX* and *EagerPSX*, are presented in [101]. Algorithm *PSX* exploits the annotated dag representation of PTPQs and uses a stack-based technique to compactly encode query matches. It avoids processing redundant matches (i.e., matches of the query dag that do not contribute to new results), and outputs results incrementally (i.e., as soon as they are produced). A theoretical analysis of *PSX* shows polynomial time and space complexity [101]. When the input query is a TPQ, *PSX* has the same time and space complexity as the state of the art streaming algorithms for TPQs [43]. An experimental evaluation shows that, compared to Algorithm X_{aos} [14] which is the only known streaming algorithm that supports TPQs extended with reverse axes, *PSX* performs better by orders of magnitude while consuming a much smaller fraction of memory space.

In order to satisfy stringent requirements of current streaming applications on query response time and memory consumption, a new polynomial time and space algorithm, *EagerPSX*, which extends *PSX*,was designed in [101]. One of its key features is that its evaluation strategy eagerly determines whether node matches should be returned as solutions to the user. An eager strategy is also used to proactively detect redundant matches. Results of an experimental comparison with *PSX* show that *EagerPSX* not only achieves better space performance without compromising time performance, but also greatly improves query response time for both simple and complex queries, in many cases, by orders of magnitude.

Summary. The stack-based algorithms extend the array-based algorithms by exploiting the path stack technique to compactly encode a potentially exponential number of query pattern matches in polynomial space. This way, they avoid the enumeration and explicit storage of query pattern matches during execution. In general, they have performance advantages over the automaton-based and array-based algorithms. This has been confirmed experimentally in [27,43,48]. The table in Fig. 20 summarizes the main features of the algorithms of the XPath streaming evaluation algorithms discussed in this paper.

Among the stack-based algorithms, *LQ* and *EQ* [43] seem to be the most competitive XPath streaming algorithms. They both achieve the best time performance in the streaming environment. They also have better memory performance than other streaming algorithms that use the lazy or eager strategy, respectively. For TPQs involv-

Algorithm	Approach	Query supported	Lazy/ Eager	Query match enumeration?	Incremental output?	Time complexity	Space complexity
XSQ [82,83]	automaton	TPQ	eager	yes	yes	$O(\lvert T\rvert \times 2^{\lvert Q\rvert} \times r^{\lvert Q\rvert}))$	n/a
$SPEX$ [78,79]	automaton	TPQ or single-join dag	lazy	no	yes	$O(\lvert T\rvert \times \lvert Q\rvert \times h)$	$O(\lvert T\rvert \times B \times h + \lvert Q\rvert \times h^2)$
$Layered\ NFA$ [81]	automaton	TPQ	lazy	no	yes	$O(\lvert T\rvert \times \lvert Q\rvert)$	$O(\lvert T\rvert + \lvert Q\rvert \times h)$
$TurboXPath$ [54]	array	TPQ	lazy	yes	no	$O(\lvert T\rvert \times r^H)$	n/a
X_{aos} [14]	array	dag	lazy	yes	no	n/a	n/a
[85]	stack	TPQ	eager	no	yes	$O(\lvert T\rvert \times (\lvert Q\rvert + h + H^2))$	$O(\lvert T\rvert \times H + \lvert Q\rvert \times h)$
$QuickXScan$ [103]	stack	TPQ	lazy	no	yes	$O(\lvert T\rvert \times \lvert Q\rvert \times r)$	$O(\lvert T\rvert + \lvert Q\rvert \times r)$
$TwigM$ [27]	stack	TPQ	lazy	no	yes	$O(\lvert T\rvert \times \lvert Q\rvert \times (\lvert Q\rvert + B \times h))$	$O(\lvert T\rvert \times h + \lvert Q\rvert \times r)$
LQ [43]	stack	TPQ	lazy	no	yes	$O(\lvert T\rvert \times \lvert Q\rvert)$	$O(\lvert T\rvert + \lvert Q\rvert \times r)$
EQ [43]	stack	TPQ	eager	no	yes	$O(\lvert T\rvert \times \lvert Q\rvert)$	$O(\lvert T\rvert + \lvert Q\rvert \times r)$
$StreamTX$ [48]	stack	TPQ	lazy	yes	no	n/a	n/a
$Twig^2Stack$ [25]	stack	TPQ	lazy	yes	no	$O(\lvert T\rvert \times \lvert Q\rvert)$	$O(\lvert T\rvert \times \lvert Q\rvert)$

Fig. 20 The comparative table for XPath streaming evaluation algorithms. Notation summary: $|Q|$ is the size of query Q, $|T|$ is the size of the XML tree T, r is the recursion depth of Q on T, B is the fan-out Q, H is the height of Q, and h is the height of T

ing multiple output nodes, *StreamTX* [48] seems to be the best one. It shows superior performance over the array-based algorithm *TurboXPath* [54] on queries with multiple output nodes.

3.6 Streaming algorithms for XPath queries with ordered axes

Streaming algorithms for processing XPath queries that involve only downward un-ordered XPath axes (*child, descendant*) can be extended for handling XPath queries with ordered axes (*following-sibling*, *following*, *preceding-sibling*, and *preceding*) [75,76,78,81,86].

Computing queries with backward ordered axes *preceding and preceding-sibling* implies retrieving pattern matchings against history stream. For instance, consider evaluating an XPath query `//A/preceding::B` against an XML document T. For each current stream node a matching A, we need to find all the embeddings of pattern $//B$ in T each of which maps B to a node b preceding a in T. Those b nodes preceding a in T are descendants of node a's ancestors in T. The computation can be performed in a bottom-up manner by exploiting properties of stacks to record pattern matches needed for later use. Stack-based streaming algorithms for un-ordered XPath axes can be easily adapted to perform the computation. A stack-based algorithm presented in [86] is capable of processing queries involving backward ordered axes in their predicates (e.g., `//A[preceding::B]`). In [76], a processing technique is presented for computing XPath queries whose backward ordered axes do not appear in predicates (e.g., `//A/preceding::B`). The given query is translated into a TPQ whose nodes are annotated with additional constraints to capture the semantics of backward ordered axes. Figure 21a shows a TPQ representation of `//A/preceding::B`. The leftmost node of the augmented TPQ is the return node of the query. The annotated constraints (denoted by the horizontal arrows among nodes) indicate that nodes of predicate matches for verifying the membership of a candidate match in the result must come after the node of the candidate match in the stream. The augmented TPQ is computed against the input data using a stack-based algorithm.

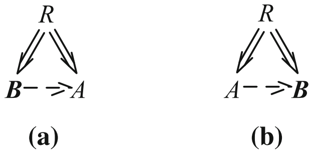


Fig. 21 TPQ representations for XPath queries with ordered axes. **a** A TPQ for //A/preceding::B, **b** A TPQ for //A/following::B

Computing queries with forward ordered axes *following and following-sibling* implies that the computation of pattern matchings refers to nodes which are still in the stream. For instance, consider evaluating an XPath query

`//A/following::B` against an XML document T. For each current stream node a matching A, we need to find all the embeddings of pattern $//B$ in T each of which maps B to a node b in T that has not yet arrived. Those b nodes following a in T are descendants of node a's ancestors in T. Three different streaming algorithms have been developed to compute queries with forward ordered axes.

As discussed in Sect. 3.3, *SPEX* [78] computes an XPath query by mapping it to a network of transducers. Like *child* and *descendant* axes, each forward ordered axis in an XPath query is mapped to a transducer. The transducer corresponding to a forward ordered axis is configured to move annotations of nodes to the next siblings of their ancestors (instead of moving them to the children or the descendants of these nodes).

The Layered NFA method for XPath queries with unordered axes discussed in Sect. 3.3 is also used to compute XPath queries that have forward ordered axes [81]. Recall that the approach uses the concept of axis *scope* to keep track of the execution status and maintain the buffered tree. The scope of an axis for a context node denotes the period during evaluation when the context node may become effective. The only difference of the computation of ordered axes lays in the determination of the scope. The scope of a forward ordered axis depends on the predicate results of the step that matches a context node. When the context node satisfies its predicates, it will remain effective until the end of the stream is reached. Otherwise, it is useless and it is removed from the buffer. In contrast, the scope of a non-ordered axis (*child* or *descendant*) is determined by the open and close events of the context node.

A stack-based algorithm for computing XPath queries with forward ordered axes which do not appear in predicates (e.g., `//A/following::B`) is presented in [75]. As in [76], the input XPath query is translated into a TPQ whose nodes are annotated with additional constraints to capture the semantics of forward ordered axes. Figure 21b shows a TPQ representation of `//A/following::B`. In the augmented TPQ, the rightmost node is the return node of the query. Both algorithms in [75,76] essentially extend *LQ* [43] to compute augmented TPQs.

3.7 Streaming algorithms for evaluating multiple XPath queries

Different applications may require finding concurrently the matching elements of multiple queries against the streaming XML document. This is the XML multiquery streaming evaluation problem. A number of approaches have been developed to handle this problem [23,26,31,62,73]. Most of them use automaton-based methods to retrieve query pattern matches.

Some approaches have focused on a restricted fragment of XPath like path expressions [23,62]. Given a set of path queries and a streaming XML document, mqX-$scan$ [62] buffers in memory a data path from the root of the XML document to the last seen element (*current path*) and computes each input query against the current path. The process continues until the end of the stream is reached. To enable real-time processing, a large number of path queries on the streaming XML data, $AFilter$[23] exploits the sharing opportunities across different queries. Specifically, it indexes the input queries to identify both common subquery prefixes and suffixes. These common subexpressions among queries need to be computed only once and their matches are shared across the queries. To avoid enumerating an exponentially large number of active states during the automata execution, $AFilter$ uses a lazy mechanism as *XPush*[46] to retrieve pattern matches.

One processing scheme for multiple TPQs decomposes each TPQ to its constituent paths, evaluates the resulting path queries, and merge-joins path matches to produce query solutions [26,31]. *YFilter* [31] builds a NFA to represent the set of path queries, and supports shared processing of the common prefixes of all these paths. Unlike *YFilter* which evaluates path queries top-down (i.e., from query root to leave) and exploits the sharing of common prefixes, *GFilter* [26] uses a bottom-up approach which evaluates path queries bottom-up and exploits the sharing of common suffixes. Moreover, *GFilter* exploits the hierarchical stack encoding scheme presented in *Twig*^{2}*Stack* [25] (See Sect. 3.5.2) to encode the path matches in polynomial time and space. This technique resolves the exponential path enumeration problem that can occur with *YFilter* [31].

XTREAM [73] uses a different scheme to evaluate multiple TPQs. It constructs for each TPQ a tree-structured NFA, where each state corresponds to a node in the TPQ. The queries are evaluated by traversing the NFAs with respect to the incoming stream events. Two runtime stacks are used to keep track of active states during evaluation. Unlike *YFilter* [31] and *GFilter* [26], *XTREAM* does not exploit sharing opportunities across different queries.

Index-Filter [20] extends *YFilter* [30] through the path stack technique [21] to compute query matches of multiple path pattern queries on XML document streams. However, *Index-Filter* is not a strict streaming algorithm since it requires the XML document to be pre-processed and indexed. Algorithms for finding query matches of multiple TPQs on XML document streams are presented in [61]. Similar to *Index-Filter*, they are not strict streaming algorithms in that they require scanning the input data in more than one pass.

4 XQuery streaming evaluation and optimization

Query language XQuery [6] has evolved into a powerful and widely accepted query language for XML data

XQuery Engine	Processing Paradigm	XQuery Supported	Optimization Scheme
XSM [66]	transducer-based	non-recursive XQuery	schema-based
Tukwila[52]	automata-algebra	FLWR (no predicates in path expressions)	cost-based
Galax [69]	algebra-based	XQuery 1.0	data pre-filtering
YFilter [31]	automata-algebra	a single FLWR	schema-based
XQRL [36]	pull-based	XQuery 1.0	query rewriting
FluXQuery [58,57]	algebra-based	FLWR (no // and * in path expressions)	data output unblocking
TurboXPath [54]	unclassified	FLW	static buffer releasing
Raindrop [91,92]	automata-algebra	FLWR	early data filtering
XQPull [33]	pull-based	subset of XQuery 1.0	unclassified

Fig. 22 The comparative table for XQuery streaming evaluation engines

over the past years. As mentioned in Sect. 1.1, an XQuery query is based on a FOR-LET-WHERE-RETURN (FLWR) construct. The FOR and LET clauses contain a series of variable names and XPath expressions. The WHERE clause defines selection and join predicates. The RETURN clause creates the output XML structure. Consider, for instance, the following simple XQuery query:

FOR \$x in //*bib*
WHERE \$x/*year* = "2011″
RETURN ⟨result⟩ {\$x//*title*} ⟨/result⟩

The query returns the *title* of articles published in year 2011, on the XML data in Fig. 1b. The query variable x is bound to a sequence of *bib* nodes as computed by the expression //*bib*. In addition, each binding must satisfy the condition specified in the WHERE clause: the value of its child node *year* in the XML data should be 2011. For each such binding of x, the RETURN clause is invoked to construct an XML fragment which consists of its descendant element *title* in the data.

Besides FLWR constructs, an XQuery query can contain an ORDER BY clause to specify the sort order of the results. Aggregate functions such as max(), min(), sum(), count(), avg() and a GROUP BY operator (introduced in XQuery 1.1) are supported by XQuery as well.

XQuery expressions can be nested within FLWR clauses to build hierarchical expressions. The compositional syntax of XQuery makes XQuery much more expressive than XPath. In the streaming context, the rich semantics of XQuery makes the evaluation and optimization problem more challenging than that of XPath since: (1) the nested syntax of XQuery often implies an evaluation plan that uses the nested-loop procedure. Such an evaluation plan requires multiple passes over the data and results in poor evaluation performance. A careful analysis of the semantics of the XQuery query is needed to find an efficient query plan; (2) in processing an XML document stream, XPath expressions are evaluated as node-selecting queries whereas XQuery expressions are used for specifying XML data transformations.

XQuery is useful not only to query XML data stored in databases, but also to process XML data streams. Various XQuery stream query engines have been developed recently, such as *XSM* [66], *Tukwila*[52], *YFilter* [31], *Galax* [36,69], *XQRL* [36], *FluXQuery* [58,57], *TurboXPath* [54], *Raindrop* [91,92], and *XQPull* [33]. Most of these engines support only a subset of the XQuery language.

Unlike relational streaming data, which are flat and consist of attribute-value pairs or tuples, XML streaming data are nested. Also, unlike relational streaming query engines, which focus on grouping and aggregation and window operations [38], XML streaming query engines focus on physically matching query patterns over XML data streams and constructing the results. Even though efficient algorithms are proposed to process grouping operations over persistent XML data [96], it is not known how they can be extended to handle group by queries over XML streaming data. *FlowGraph* [64] performs static data dependency analysis on queries for generating query execution plans that allow computing XQuery aggregations on the fly. Extending XQuery to handle window-based aggregations over XML streams is explored in [17].

Below we classify the existing XQuery stream query engines by their processing paradigms and provide a high level description for each paradigm. Then, we overview the existing XQuery optimization techniques for processing XML stream data. Figure 22 provides a comparative view for the XQuery streaming engines in terms of the supported fragment of XQuery, the adopted processing paradigm and the optimization scheme.

4.1 XQuery streaming evaluation paradigms

The evaluation approaches taken by the existing XQuery stream query engines largely follow four processing paradigms: the transducer-based paradigm, the algebra-based paradigm, the automata-algebra paradigm, and the pull-based paradigm. Note that the processing model of *TurboXPath* [54] does not strictly follow one of the above four paradigms. Rather, it uses the array-based approach to compute a parse tree with multiple output nodes (a query plan compiled from the given query) against the input XML data stream and produces a sequence of tuples of XML fragments matching the output nodes. We have described the computation process of *TurboXPath* in Sect. 3.4.

The transducer-based paradigm. This paradigm adapts traditional transducers [47] augmented with buffers to process XQuery on XML data streams. XSM [66] is one noticeable XQuery engine that follows this processing paradigm.

XSM first compiles a query into a network of transducers. Each transducer corresponds to a subexpression of the query (e.g., a path expression). A transducer takes as input one or more XML data streams and produces one or more output streams. It may be additionally associated with input/output buffers and a set of read/write pointers. Buffers are used to store variable bindings. Then, the network of transducers is reduced to a single transducer by repeatedly composing a pair of transducers linked via buffers into a single transducer. XSM is designed to process non-recursive XML data. It supports a subset of XQuery whose path expressions do not have descendant axes. In order to reduce computation, for instance, to eliminate unnecessary transition states, schema information is exploited.

While the transducer-based paradigm is clean and elegant, it has two potential problems: (1) XQuery engines following this paradigm are difficult to generalize to support full XQuery. The reason is, in this paradigm, every XQuery construct has to be expressed by a transducer. This is very hard even for simple XQuery constructs, such as sequence concatenation and element construction [32]; (2) automata operate at a low level of abstraction which involves all internal details of the computations. As a result, it may be difficult to integrate automata-based approaches with algebraic-based approaches for query optimization.

The algebra-based paradigm. This processing paradigm produces an algebraic query plan for a given XQuery and uses algebra operators to evaluate queries on XML data streams. One of its advantages over the transducer-based paradigm is that it offers more query optimization opportunities, since optimizations can be developed for each operator separately.

A number of XQuery engines adopt the algebra-based processing paradigm. One example is $Galax$[36], where *streaming operators* are used in the physical query algebra. These operators are event-based and are evaluated directly over the events in the input stream rather than over buffered data. A feature of $Galax$ is its blending of stream processing with traditional evaluation techniques (e.g., index-based access, join, and query unnesting) for persistent XML databases. Without introducing dedicated streaming operators, *FluXQuery* [58,57] specifies which parts of the query are evaluated directly on the stream and which are evaluated over buffered data. This is in contrast to $Galax$ [36] which specifies the physical query plans in detail. Finally, the work present in [34] adapts XQuery algebras [35] for conventional query processing on stored XML data to the streaming context.

The automata-algebra paradigm. This paradigm models the query semantics as a combination of automata and algebraic operations. A given XQuery is decomposed into two parts: one part uses automata to process all the path expressions in the query and the other part is processed based on algebraic techniques. Accordingly, the query execution proceeds in two stages: the pattern matching stage and the post-processing stage.

The pattern matching stage converts the input XML stream into a stream of tuples by binding variables to the nodes matched by XPath expressions. Automata are used to compute pattern matchings of XPath expressions against the input XML data stream. The matched nodes are combined to form tuples which are forwarded to the post-processing stage. The pattern matching stage also incrementally maintains a tree which represents some of the parsed XML stream events for later use. The post-processing is performed by an algebraic engine which filters, combines, and restructures the tuples to produce the final query answer. Most of the query operators used in this stage are similar to the standard relational equivalents. In order to accommodate the streaming environment, tuples are incrementally produced in the first stage and are pipelined to the second stage so that query results can be delivered to the user on the fly. As one can see, unlike the transducer-based paradigm which uses stream events as the processing unit throughout the evaluation process, the automata-algebra paradigm uses two processing units (stream events and tuples) in different stages.

The automata-algebra paradigm has the following benefits: (1) it can utilize the automaton-based methods developed for the XPath streaming evaluation, and (2) it can leverage the mature techniques from relational query processing.

A number of XQuery stream querying engines adopt the automata-algebra paradigm including $Tukwila$[52], *YFilter* [31], and *Raindrop* [92]. Besides using automata, *Raindrop* additionally defines a class of stream algebra operators to perform data navigation, variable binding, and structural join operations. These algebra operators allow *Raindrop* to take advantage of algebra-based optimizations for streaming evaluation. We will present the optimization techniques of *Raindrop* later. A unique feature of *YFilter* is the employment of an NFA to represent the full set of navigation paths and the support of shared processing of the common prefixes of all these paths. *YFilter* uses a schema-based optimization technique for reducing computation at the post-processing stage. Unlike *YFilter* and *Raindrop*, the optimization operations in $Tukwila$ are cost-based and take place at the physical level.

The pull-based paradigm. This paradigm has two distinctive features: (1) the construction of a query plan as an operator pipeline, and (2) the use of a pull-based processing model on input streams. The goal of the pull-based paradigm is to minimize unnecessary data buffering between operators. In essence, this is an instance of the classic database principle of pipelining. Both *XQRL* [36] and *XQPull* [33] adopt this processing paradigm.

In this paradigm, a given XQuery is translated into a pipeline of operators, called *iterators*, with each XQuery syntactic structure (e.g., a path expression or a FOR clause) being mapped to an iterator. The iterators are connected in the same way as the corresponding XQuery syntactic structures are composed to form the query. Each iterator is equivalent in functionality with an algebraic operator used by relational query engines. At run-time, the iterators process the stream of events through the pipeline one-event-at-a-time in a pull-based fashion: an iterator delivers a stream event to the output only when requested by the next iterator in the pipeline. To deliver one stream event to the output, an iterator requests from the previous iterator as many events as necessary to produce a single event. The execution continues until the end of the input stream is reached.

The pull-based paradigm has the following benefits: (1) the compositional approach of constructing query plans leads to concise, clean, and extensible implementations [33], and (2) the pipeline-based execution model avoids buffering intermediate results when possible.

Nevertheless, the pull-based paradigm suffers from three potential problems: (1) it incurs the overhead of method calls to handle an event, the number of which is proportional to the size of the pipeline. In contrast, the cost of handling an event in other paradigms is typically constant; (2) a custom wrapper is required for the operators to process the pull-based events; and (3) the execution of iterators needs to be synchronized and this complicates programming considerably.

4.2 XQuery streaming optimization

4.2.1 Problem statement

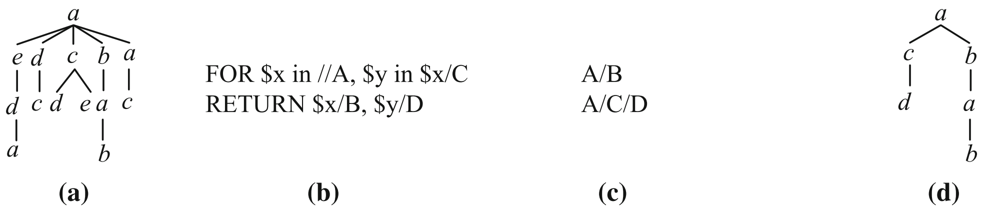
The semantics of XQuery requires a query engine to compute the subexpressions of a query only after the referred nodes have been fully read from the input XML stream and may be assumed available in main memory buffers [58]. In the streaming context, the input cannot be completely loaded in memory prior to query evaluation. Nevertheless, a query engine has to buffer some XML nodes until it can verify that they are not part of the query answer. A large main memory buffer requirement may lead to a significant CPU consumption due to the operation cost on the buffered data. For this reason, a primary optimization target on XQuery streaming evaluation is the minimization of main memory consumption and efficient buffer management. Indeed, considerable work on optimizing XQuery streaming evaluation has focused on developing buffer optimization techniques for dealing with and reducing the amount of data buffered in main memory [58,64,69,87,91]. A more recent work [71] proposed a solution which bounds the amount of data buffering by building a stream synopsis. In this survey, we focus on the problem of optimizing XQuery streaming evaluation as a buffer optimization problem.

Schmidt et al. [87] list three desiderata for the buffer management of an XML streaming evaluation engine: (1) only data that is relevant for query evaluation is stored in the buffer; (2) data are not buffered longer than necessary; and (3) no multiple copies of data is kept in buffers. To be optimal for (1), a system should be able to check the satisfiability of XQuery expressions, which however is an undecidable problem [15]. Therefore, XQuery engines that target on buffer optimizations have to rely on a best-effort approach.

Buffer optimization techniques often exploit schema constraints to: (1) statically infer the buffers which are necessary to avoid superfluous buffering [58]; (2) predict the non-occurrence of a certain pattern within a context; and (3) detect the failure of predicates early on so as to discard earlier the data which fail on these predicates [91]. Utilizing schema information to optimize XML filtering performance is addressed in [88]. A common assumption here is that XML streams are generated conforming to a pre-defined schema such as *document type definition* (DTD) and XML Schema.

4.2.2 Optimization schemes

We categorize the existing XQuery streaming optimization techniques into the following five optimization schemes: query rewriting, data pre-filtering, early data filtering, data output unblocking, and early buffer releasing. These optimization techniques use either static or dynamic analysis or a combination of the two. The static analysis technique is performed at compile time. Example applications of static analysis include deriving constraints or inferring types from schema information, analyzing queries regarding the relevance of data to query evaluation, and analyzing the data dependencies to transform queries. Many XQuery streaming engines [31,54,58,64,66,69,91] adopt static analysis for query optimization. The dynamic analysis is performed at runtime, where an optimization decision is made based on the current buffer content, the current state of query evaluation, and the part of the input read so far. Eager streaming evaluation techniques introduced in Sect. 3.5 use dynamic analysis. Dynamic analysis is also used in [87] for implementing buffer releasing optimization.



```
FOR $x in //A, $y in $x/C
RETURN $x/B, $y/D
```

```
A/B
A/C/D
```

Fig. 23 An example of the application of the XML document projection technique in *Galax*. **a** An XML tree T, **b** an XQuery query Q, **c** two projection paths of Q, **d** the projected XML tree [69]

Below, we describe the five optimization schemes at the conceptual level. More technical details for each optimization technique can be found in the referenced papers.

Query rewriting. The query rewriting intends to translate a query expression generated by the query parser into an equivalent expression that is cheaper to evaluate. It is generally rule-based and uses static analysis over the input query. A number of XQuery streaming engines adopt this optimization scheme, including *XQRL* [36], *FlowGraph* [64], and *Raindrop* [91,92].

XQRL [36] develops a set of heuristic query rewriting rules. Examples of the rewriting rules include query normalization rules such as the rule of unnesting FLWR expressions in the FOR and RETURN clauses, rules for reducing computation such as the rule of translating the descendant axis to a sequence of child axes based on schema information, and rules for enabling streaming evaluation such as the rule of translating expressions with backward axes (e.g., *parent* and *ancestor*) into expressions with only forward axes whenever possible [80].

FlowGraph [64] rewrites XQuery queries for streaming evaluation by statically analyzing data dependencies between the variables of the input query. Observing that XPath expressions in nested loops often incur the traversal of a data stream multiple times, *FlowGraph* develops techniques that rewrite XQuery queries that involve nested-loop XPath expressions so that they can be executed with only a single scan of the data stream. The techniques include unrolling nested FOR expressions, finding the common prefix of the XPath expressions that traverse a data stream multiple times, and pipelining the processing to remove unnecessary buffering between two query operators.

Data pre-filtering. Document Object Model (DOM) [1] is a data model which represents an XML document as a (ordered) node-labeled tree. Conventional in-memory XPath or XQuery engines [40–42,56] load the entire XML document in memory often by constructing a DOM tree before processing it. These in-memory engines consume main memory in large multiples of the size of the input XML documents and therefore face a scalability issue on larger XML document inputs [69]. In order to reduce memory requirements in XQuery processors, *Galax* [69] develops a data pre-filtering technique called *XML document projection*. This technique resembles promoting projections before a join in relational queries. Based on a static analysis of the structural requirements of the query, it prunes all data that are certain to be irrelevant to query evaluation and stores in memory the projected document. This way, the in-memory evaluation can be conducted on the projected XML document whose size is generally smaller than the original one. Figure 23 shows an example of applying the XML document projection technique for evaluating the XQuery query Q on the XML tree T. As one can see, the resulting projected XML tree (Fig. 23d) is much smaller than the input XML tree T. Figure 23c shows the two projection paths of Q obtained through a static analysis on Q.

The projection technique of *Galax* [69] is further refined in [16,19] to reduce the pruning overhead and improve the pruning precision. The technique presented in [19] utilizes a structural index of the data and performs the pruning at run-time. The work presented in [16] extends the projection technique to support backward axes as well as query predicates by taking advantage of the schema information.

Early data filtering. In the streaming context, a query is computed by sequentially scanning the data only once. Normally, there is no way to jump to a certain portion of the stream. However, by exploiting schema constraints, the sequential scanning can be expedited by skipping computations that do not contribute to the final answer. This is achieved through techniques on early detecting failed query predicates or predicting the non-occurrence of a certain pattern within a bound context.

As an example, consider an XQuery query: `FOR $x in /A[B] WHERE $x/C ="1"RETURN $x//D,$x//E`. For a binding x on an A node in the input, a naive execution strategy would check the existence of its child B node and the satisfaction of the predicate and produce the output after the end event of the bound node has arrived. Five computations have to be performed on each binding x. These are: (1) finding pattern /B, (2) finding pattern /C, (3) evaluating whether a matched C node contains “1”, (4) buffering nodes matching //D, and (5) buffering nodes matching E. Now suppose the following DTD is given: `<!ELEMENT`

A(B?,F,C+,D*,E*)>. From the DTD, we know that element F is a mandatory child of element A. When a start event of F is encountered but no B node has been located, we can determine that no results can be produced from the subtree rooted at the current binding on A in the input XML document. We can then skip all the remaining computations (2)–(5). This can lead to significant performance improvement when there are a large number of C, D, and E nodes in the subtree.

Raindrop [91,92] realizes the early filtering optimization scheme by transforming the query plan based on a set of semantic optimization rules. The rules utilize the constraints of ordering, occurrence, and type choice that are derived from a given DTD. For instance, in the previous example, the evaluation of the query can be optimized using an order rule which indicates that the open event of F is an *ending mark* of the pattern /B in the binding on A.

The early data filtering optimization technique can reduce the memory usage since: (1) buffered data failed on the predicates can be purged from the buffer early, and (2) data buffering can be avoided due to skipped computations.

Besides using schema knowledge, another way of realizing the data filtering strategy for optimizing queries is to use indices on XML streams [45]. The streaming index SIX [45] provides the start and end positions of each element in the stream. During evaluation, irrelevant elements are skipped by jumping to their end positions.

Data output unblocking. During query execution, data may need to be buffered before being delivered to the output in two cases: (1) pattern matching results need to be produced according to a specified order, and (2) some predicates for determining the membership of a pattern matching result in the output may have not yet be satisfied. However, if it is known that the order of the data considered for output coincides with the order of the data in the input (for Case 1), or that, through an early detection, predicates are known to be satisfied (for Case 2), results can be delivered on the fly without incurring unnecessary data buffering. This way, the main memory consumption can be reduced significantly. Usually, an order constraint among matched query patterns for output can be obtained through static analysis of data dependencies and with schema knowledge.

For example, consider the following XQuery query: FOR $x in /A/B, $y in $x/C, $z in $x/DRETURN ⟨result⟩ {$y} {$z} ⟨/result⟩. Without schema information, for each binding of x on a path /A/B, an XQuery engine would normally have to buffer all stream D nodes matching the query node D before delivering the results. However, given the DTD <!ELEMENT A(B*)> <!ELEMENT B(C*,D*)>, we know that the order of nodes considered for output coincides with the order of the data in the input. Therefore, the C and D nodes can be output the moment their open events are encountered. As a result, no data buffering is needed.

This optimization technique is adopted by both *FluXQuery* [57,58] (Case 1 only) and *R-SOX* [94] (both cases). *R-SOX* enhances the static optimization techniques of *Raindrop* [91,92] to support runtime optimization. Both *FluXQuery* and *Raindrop* use static analysis and exploit schema information to rewrite query plans. While the goal of *FluXQuery* is to minimize the buffer size, *Raindrop* focuses on reducing unnecessary computations. These two goals are complementary since skipping unnecessary computations reduces the buffer size.

Early buffer releasing. In order to successfully reduce the main memory consumption, an effective technique is needed to timely remove data from the buffer once they are no longer relevant to query evaluation. The key to realize the early buffer releasing is to identify the moments when the evaluation of certain subexpressions has finished.

In [87], an approach called *active garbage collection* is presented to realize the early buffer releasing. The approach centers around the notion of relevance of data to query evaluation. It combines the technique of XML document projection for pre-filtering data irrelevant to query evaluation [69] and the technique of garbage collection for automatic memory management in programming languages [95]. Briefly, the approach proceeds as follows. Based on a static analysis of the structural requirements of the query, it extracts projection paths [69] from the query. While reading the XML input, it prefilters it based on the extracted projection paths, and assigns *roles* to each XML node copied into the buffer. The roles of a node capture the situation where the node is matched by multiple projection paths and even multiple times. During query evaluation, the roles of nodes are reduced. The moments for reducing the roles of nodes are determined at compile time. A node becomes a candidate for removal from the buffer at runtime once it has lost all its roles.

In contrast to the early buffer releasing scheme, the static buffer releasing scheme determines when to purge buffers at compile time. In the static scheme, the lifetime of a buffer is associated with the scope of an XQuery variable. Buffers are purged once the scope of the associated variable ends. A potential problem of the static scheme is the duplicate buffering, which can happen when an XML node is bound by different variables (e.g., by a condition checking variable and by a return variable) or when an XML node participates in more than one pattern match (and this is more likely when the query involves descendant axes and wildcards) [87]. The duplicate buffering problem is resolved by the active garbage collection approach through dynamic analysis [87]. By proactively releasing buffers, the early buffer releasing scheme is more effective on reducing main memory consumption than the static one. A number of XQuery engines, including *TurboXPath* [54], *FluXQuery* [57,58] and *FlowGraph* [64], use the static buffer releasing scheme.

5 Conclusion

We have reviewed the state of the art of XPath and XQuery evaluation techniques against XML data streams. As this survey has shown, most current methods for processing queries over XML streams have focused on physically matching query patterns over XML data streams and constructing query results. Among the three major types of XPath streaming evaluation techniques, the stack-based approach seems the most efficient. Algorithms in this approach evaluate TPQs against XML streams in polynomial time and space, which is a significant improvement over automaton-based and array-based algorithms. Also, as shown in [101], stack-based evaluation techniques on TPQs can be extended to support a broad structural fragment of XPath that goes beyond and strictly contains TPQs and dags. Existing XQuery stream query engines, though, have mainly employed automata-based techniques for computing XPath expressions. We have identified four processing paradigms implemented by existing XQuery stream query engines. Among them, the automata-algebra paradigm seems to enjoy more benefits as it is able to leverage both the automaton-based methods developed for the XPath streaming evaluation and the mature techniques developed for relational query processing.

We have also reviewed XQuery optimization techniques for processing XML stream data. We identified the XQuery streaming optimization problem as a buffer optimization problem. Five buffer optimization schemes were discussed. An effective optimization scheme such as the scheme of early buffer releasing is promising since it combines both the static analysis technique and the (dynamic) eager evaluation strategy for the streaming optimization task.

Although considerable research has been done in the area of query evaluation on XML streams, there are several issues that remain open in this area. We identify a couple of them below. One issue is the support of windows. The window mechanism is used by relational stream engines for extracting a finite relation from an infinite stream [59]. Most of the existing XQuery stream query engines support XQuery 1.0 (or a subset of it), which lacks support for window queries [17]. As window functions have been included as a new feature to XQuery 3.0, efficient techniques need to be developed to process window queries using XQuery.

Another issue concerns query evaluation and optimization across multiple XML streams. Techniques that are capable of processing queries which correlate multiple input events have been recognized as being highly important for event processing, since they allow detecting complex patterns in real-time [50]. However, most existing techniques have focused on the efficient evaluation of XPath or XQuery queries over a single XML data stream. Although there has been some work on this problem (e.g., [50]), new techniques need to be developed to support the evaluation of complex XQuery queries over multiple XML streams.

References

1. Document Object Model (DOM). World Wide Web Consortium site, W3C. http://www.w3.org/DOM/
2. NewsML: News Markup Language. http://www.newsml.org
3. NITF: News Industry Text Format. http://www.nitf.org
4. World Wide Web Consortium site, W3C. http://www.w3.org/
5. XML Path Language (XPath). World Wide Web Consortium site, W3C. http://www.w3.org/TR/xpath20
6. XML Query Language (XQuery). World Wide Web Consortium site, W3C. http://www.w3.org/XML/Query
7. Abiteboul, S., Marinoiu, B.: Distributed monitoring of peer to peer systems. In: WIDM, pp. 41–48 (2007)
8. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Comput. Netw. **38**(4), 393–422 (2002)
9. Altinel, M., Franklin, M.J.: Efficient filtering of XML documents for selective dissemination of information. In: VLDB, pp. 53–64 (2000)
10. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom J.: Models and issues in data stream systems. In: PODS, pp. 1–16 (2002)
11. Bar-Yossef, Z., Fontoura, M., Josifovski V. (2004) On the memory requirements of XPath evaluation over XML streams. In: PODS, pp. 177–188 (2004)
12. Bar-Yossef, Z., Fontoura, M., Josifovski V.: Buffering in query evaluation over XML streams. In: PODS, pp. 216–227 (2005)
13. Bar-Yossef, Z., Fontoura, M., Josifovski, V.: On the memory requirements of XPath evaluation over XML streams. J. Comput. Syst. Sci. **73**(3), 391–441 (2007)
14. Barton, C., Charles, P., Goyal, D., Raghavachari, M., Fontoura, M., Josifovski, V.: Streaming XPath processing with forward and backward axes. In: ICDE, pp. 455–466 (2003)
15. Benedikt, M., Fan, W., Geerts, F.: XPath satisfiability in the presence of DTDs. J. ACM 55(2), 8:1–8:79 (2008)
16. Benzaken, V., Castagna, G., Colazzo, D., Nguyen K.: Type-based XML projection. In: VLDB, pp. 271–282 (2006)
17. Botan, I., Fischer, P.M., Florescu, D., Kossmann, D., Kraska, T., Tamosevicius, R.: Extending XQuery with window functions. In: VLDB, pp. 75–86 (2007)
18. Böttcher, S., Steinmetz, R.: Evaluating xpath queries on XML data streams. In: BNCOD, pp. 101–113 (2007)
19. Bressan, S., Catania, B., Lacroix, Z., Li, Y.G., Maddalena, A.: Accelerating queries by pruning XML documents. Data Knowl. Eng. **54**(2), 211–240 (2005)
20. Bruno, N., Gravano, L., Koudas, N., Srivastava, D.: Navigation- vs. index-based XML multi-query processing. In: ICDE, pp. 139–150 (2003)
21. Bruno, N., Koudas N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: 310
22. Candan, K.S., Hsiung, W.-P., Chen, S., Tatemura, J., Agrawal, D.: AFilter: Adaptable XML filtering with prefix-caching and suffix-clustering. In: VLDB, pp. 559–570 (2006)

23. Candan, K.S., Hsiung, W.-P., Chen, S., Tatemura, J., Agrawal, D.: Afilter: adaptable XML filtering with prefix-caching suffix-clustering. In: VLDB, pp. 559–570. VLDB Endowment (2006)
24. Chan, C.Y., Felber, P., Garofalakis, M.N., Rastogi, R.: Efficient filtering of XML documents with XPath expressions. In: ICDE, pp. 235–244 (2002)
25. Chen, S., Li, H.-G., Tatemura, J., Hsiung, W.-P., Agrawal, D., Candan, K.S.: Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In: VLDB (2006)
26. Chen, S., Li, H.-G., Tatemura, J., Hsiung, W.-P., Agrawal, D., Candan, K.S.: Scalable filtering of multiple generalized-tree-pattern queries over XML streams. IEEE Trans. Knowl. Data Eng. **20**(12), 1627–1640 (2008)
27. Chen, Y., Davidson, S.B., Zheng, Y.: An efficient XPath query processor for XML streams. In: ICDE, p. 79 (2006)
28. Cortes, C., Fisher, K., Pregibon, D., Rogers, A.: Hancock: a language for extracting signatures from data streams. In: KDD, pp. 9–17 (2000)
29. Cranor, C.D., Johnson, T., Spatscheck, O., Shkapenyuk, V.: Gigascope: a stream database for network applications. In: SIGMOD, pp. 647–651 (2003)
30. Diao, Y., Altinel, M., Franklin, M.J., Zhang, H., Fischer, P.M.: Path sharing and predicate evaluation for high-performance XML filtering. ACM Trans. Database Syst. **28**(4), 467–516 (2003)
31. Diao, Y., Franklin, M.J.: High-performance XML filtering: an overview of YFilter. IEEE Data Eng. Bull. **26**(1), 41–48 (2003)
32. Fegaras, L.: Efficient processing of XML update streams. In: ICDE, pp. 616–625 (2008)
33. Fegaras, L., Dash, R.K., Wang, Y.: A fully pipelined XQuery processor. In: XIME-P (2006)
34. Fegaras, L., Levine, D., Bose, S., Chaluvadi, V.: Query processing of streamed XML data. In: CIKM, pp. 126–133 (2002)
35. Fernández, M.F., Siméon, J., Wadler, P.: A semi-monad for semi-structured data. In: ICDT, pp. 263–300 (2001)
36. Florescu, D., Hillery, C., Kossmann, D., Lucas, P., Riccardi, F., Westmann,T., Carey, M.J., Sundararajan, A., Agrawal, G.: The BEA/XQRL streaming XQuery processor. In: VLDB, pp. 997–1008 (2003)
37. Foster, I.T., Kesselman, C., Nick, J.M., Tuecke, S.: Grid services for distributed system integration. IEEE Comput. **35**(6), 37–46 (2002)
38. Golab, L., Özsu, M.T.: Issues in data stream management. SIGMOD Record **32**(2), 5–14 (2003)
39. Gong, X., Yan, Y., Qian, W., Zhou, A.: Bloom filter-based XML packets filtering for millions of path queries. In: ICDE, pp. 890–901 (2005)
40. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. In: VLDB, pp. 95–106 (2002)
41. Gottlob, G., Koch, C., Pichler, R.: XPath query evaluation: Improving time and space efficiency. In: ICDE, pp. 379–390 (2003)
42. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. ACM Trans. Database Syst. **30**(2), 444–491 (2005)
43. Gou, G., Chirkova, R.: Efficient algorithms for evaluating XPath over streams. In: SIGMOD, pp. 269–280 (2007)
44. Gou, G., Chirkova, R.: Efficiently querying large XML data repositories: a survey. IEEE Trans. Knowl. Data Eng. **19**(10), 1381–1403 (2007)
45. Green, T.J., Gupta, A., Miklau, G., Onizuka, M., Suciu, D.: Processing XML streams with deterministic automata and stream indexes. ACM Trans. Database Syst. **29**(4), 752–788 (2004)
46. Gupta, A.K., Suciu, D.: Stream processing of XPath queries with predicates. In: SIGMOD, pp. 419–430 (2003)
47. Gurari, E.M.: Introduction to the Theory of Computation. Computer Science Press, Rockville (1989)
48. Han, W.-S., Jiang, H., Ho, H., Li, Q.: StreamTX: extracting tuples from streaming XML data. Proc. VLDB Endow. **1**(1), 289–300 (2008)
49. Hoeller, N., Reinke, C., Neumann, J., Groppe, S., Werner, C., Linnemann, V.: XML data management and xpath evaluation in wireless sensor networks. In: MoMM, pp. 218–230 (2009)
50. Hong, M., Demers, A.J., Gehrke, J., Koch, C., Riedewald, M., White, W.M.: Massively multi-query join processing in publish/subscribe systems. In: SIGMOD Conference, pp. 761–772 (2007)
51. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (2nd edn). Addison-Wesley-Longman, Boston (2001)
52. Ives, Z.G., Halevy, A.Y., Weld, D.S.: An XML query engine for network-bound data. VLDB J. **11**(4), 380–402 (2001)
53. Jeffery, S.R., Garofalakis, M.N., Franklin, M.J.: Adaptive cleaning for RFID data streams. In: VLDB, pp. 163–174 (2006)
54. Josifovski, V., Fontoura, M., Barta, A.: Querying XML streams. VLDB J. **14**(2), 197–210 (2002)
55. Kabisch, S., Peintner, D., Heuer, J., Kosch, H.: Efficient and flexible XML-based data-exchange in microcontroller-based sensor actor networks. In: AINA Workshops, pp. 508–513 (2010)
56. Kay, M.: SAXON: The XSLT and XQuery Processor. http://saxon.sourceforge.net/
57. Koch, C., Scherzinger, S., Schweikardt, N., Stegmaier, B.: FluX-Query: an optimizing XQuery processor for streaming XML data. In: VLDB, pp. 1309–1312 (2004)
58. Koch, C., Scherzinger, S., Schweikardt, N., Stegmaier, B.: Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In: VLDB, pp. 228–239 (2004)
59. Koudas, N., Srivastava, D.: Data stream query processing: a tutorial. In: VLDB, p. 1149 (2003)
60. Kwon, J., Rao, P., Moon, B., Lee, S.: Fist: Scalable XML document filtering by sequencing twig patterns. In: VLDB, pp. 217–228 (2005)
61. Lakshmanan, L.V.S., Parthasarathy, S.: On efficient matching of streaming XML documents and queries. In: EDBT (2002)
62. Lee, M.L., Chua, B.C., Hsu, W., Tan, K.-L.: Efficient evaluation of multiple queries on streaming XML data. In: CIKM, pp. 118–125 (2002)
63. Li, M., Mani, M., Rundensteiner, E.A.: Efficiently loading and processing XML streams. In: IDEAS, pp. 59–67 (2008)
64. Li, X., Agrawal, G.: Efficient evaluation of XQuery over streaming data. In: VLDB, pp. 265–276 (2005)
65. Liu, J., Roantree, M.: Precomputing queries for personal health sensor environments. In: MEDES, pp. 49–56 (2009)
66. Ludäscher, B., Mukhopadhyay, P., Papakonstantinou, Y.: A transducer-based XML query processor. In: VLDB, pp. 227–238 (2002)
67. Madden, S., Franklin, M.J.: Fjording the stream: An architecture for queries over streaming sensor data. In: ICDE, pp. 555–566 (2002)
68. Mainwaring, A.M., Culler, D.E., Polastre, J., Szewczyk, R., Anderson, J.: Wireless sensor networks for habitat monitoring. In: WSNA, pp. 88–97 (2002)
69. Marian, A., Siméon, J.: Projecting XML documents. In: VLDB, pp. 213–224 (2003)
70. Massie, M.L., Chun, B.N., Culler, D.E.: The ganglia distributed monitoring system: design, implementation, and experience. Parallel Comput. **30**(5–6), 817–840 (2004)
71. Mayorga, V., Polyzotis, N.: Sketch-based summarization of ordered XML streams. In: ICDE, pp. 541–552 (2009)
72. Megginson, D., et al.: Simple API for XML. http://www.saxproject.org/

73. Min, J.-K., Park, M.-J., Chung, C.-W.: XTREAM: an efficient multi-query evaluation on streaming xml data. Inf. Sci. **177**(17), 3519–3538 (2007)
74. Moro, M.M., Bakalov, P., Tsotras, V.J.: Early profile pruning on XML-aware publish/subscribe systems. In: VLDB, pp. 866–877 (2007)
75. Nizar, A., Kumar, P.S.: Efficient evaluation of forward xpath axes over XML streams. In: COMAD, pp. 222–233 (2008)
76. Nizar, A., Kumar, P.S.: Ordered backward XPath axis processing against XML streams. In: XSym, pp. 1–16 (2009)
77. Olteanu, D.: Evaluation of XPath queries against XML streams. In: PhD Thesis, University of Munich (2005)
78. Olteanu, D.: Spex: streamed and progressive evaluation of XPath. IEEE Trans. Knowl. Data Eng. **19**(7), 934–949 (2007)
79. Olteanu, D., Furche, T., Bry, F.: Evaluating complex queries against XML streams with polynomial combined complexity. In: BNCOD, pp. 31–44 (2004)
80. Olteanu, D., Meuss, H., Furche, T., Bry, F.: XPath: Looking forward. In: EDBT, pp. 109–127 (2002)
81. Onizuka, M.: Processing XPath queries with forward and downward axes over XML streams. In: EDBT, pp. 27–38 (2010)
82. Peng, F., Chawathe, S.S.: XPath queries on streaming data. In: SIGMOD, pp. 431–442 (2003)
83. Peng, F., Chawathe, S.S.: XSQ: a streaming XPath engine. ACM Trans. Database Syst. **30**(2), 577–623 (2005)
84. Raj, A., Kumar, P.S.: Branch sequencing based XML message broker architecture. In: ICDE, pp. 656–665 (2007)
85. Ramanan, P.: Evaluating an XPath query on a streaming XML document. In: ICMD (2005)
86. Ramanan, P.: Worst-case optimal algorithm for XPath evaluation over XML streams. J. Comput. Syst. Sci. **75**(8), 465–485 (2009)
87. Schmidt, M., Scherzinger, S., Koch, C.: Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation. In: ICDE, pp. 236–245 (2007)
88. Silvasti, P., Sippu, S., Soisalon-Soininen, E.: Schema-conscious filtering of XML documents. In: EDBT, pp. 970–981 (2009)
89. Snoeren, A.C., Conley, K., Gifford, D.K.: Mesh based content routing using XML. In: SOSP, pp. 160–173 (2001)
90. Souldatos, S., Wu, X., Theodoratos, D., Dalamagas, T., Sellis, T.K.: Evaluation of partial path queries on XML data. In: CIKM, pp. 21–30 (2007)
91. Su, H., Rundensteiner, E.A., Mani, M.: Semantic query optimization for XQuery over XML streams. In: VLDB, pp. 277–288 (2005)
92. Su, H., Rundensteiner, E.A., Mani, M.: Automaton meets algebra: a hybrid paradigm for XML stream processing. Data Knowl. Eng. **59**(3), 576–602 (2006)
93. Theodoratos, D., Dalamagas, T., Koufopoulos, A., Gehani, N.: Semantic querying of tree-structured data sources using partially specified tree patterns. In: CIKM (2005)
94. Wang, S., Su, H., Li, M., Wei, M., Yang, S., Ditto, D., Rundensteiner, E.A., Mani, M.: R-sox: Runtime semantic query optimization over XML streams. In: VLDB, pp. 1207–1210 (2006)
95. Wilson, P.R.: Uniprocessor garbage collection techniques. In: IWMM, pp. 1–42 (1992)
96. Wu, H., Ling, T.W., Xu, L., Bao, Z.: Performing grouping and aggregate functions in XML queries. In: WWW, pp. 1001–1010 (2009)
97. Wu, X., Souldatos, S., Theodoratos, D., Dalamagas, T., Sellis, T.K.: Efficient evaluation of generalized path pattern queries on XML data. In: WWW, pp. 835–844 (2008)
98. Wu, X., Souldatos, S., Theodoratos, D., Dalamagas, T., Vassiliou, Y., Sellis, T.: Processing and evaluating partial tree queries on XML data. IEEE TKDE (to appear) Electronic TKDE. doi:10.1109/TKDE.2011.137
99. Wu, X., Theodoratos, D., Souldatos, S., Dalamagas, T., Sellis, T.K.: Efficient evaluation of generalized tree-pattern queries with same-path constraints. In: SSDBM, pp. 361–379 (2009)
100. Wu, X., Theodoratos, D., Souldatos, S., Dalamagas, T., Sellis, T.K.: Evaluation techniques for generalized path pattern queries on XML data. World Wide Web **13**(4), 441–474 (2010)
101. Wu, X., Theodoratos, D., Zuzarte, C.: Efficient evaluation of generalized tree-pattern queries on XML streams. VLDB J. **19**(5), 661–686 (2010)
102. Yoo, D.-S., Tan, V.V., Yi, M.-J.: A universal data access server for distributed data acquisition and monitoring systems. In: ICIC (1) (2009)
103. Zhang, G., Zou, Q.: QuickXScan: Efficient streaming XPath evaluation. In: International Conference on Internet Computing, pp. 249–255 (2006)
104. Zhu,Y., Shasha, D.: Efficient elastic burst detection in data streams. In: KDD, pp. 336–345 (2003)