The VLDB Journal (2005) 14: 197–210 / Digital Object Identifier (DOI) 10.1007/s00778-004-0123-7

Querying XML streams

Vanja Josifovski[1], **Marcus Fontoura**[1], **Attila Barta**[2]

[1] IBM Almaden Research Center, San Jose, CA 95120, USA
e-mail: {vanja,fontoura}@us.ibm.com
[2] Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 3H5
e-mail: atibarta@cs.toronto.edu

Edited by R. Baeza-Yates. Received: January 30, 2003 / Accepted: February 4, 2004
Published online: April 8, 2004 –

Abstract. Efficient querying of XML streams will be one of the fundamental features of next-generation information systems. In this paper we propose the TurboXPath path processor, which accepts a language equivalent to a subset of the **for-let-where** constructs of XQuery over a single document. TurboXPath can be extended to provide full XQuery support or used to augment federated database engines for efficient handling of queries over XML data streams produced by external sources. Internally, TurboXPath uses a tree-shaped path expression with multiple outputs to drive the execution. The result of a query execution is a sequence of tuples of XML fragments matching the output nodes. Based on a streamed execution model, TurboXPath scales up to large documents and has limited memory consumption for increased concurrency. Experimental evaluation of a prototype demonstrates performance gains compared to other state-of-the-art path processors.

1 Introduction

Modern relational database engines are currently adding support for a new XML data type and a set of new XML-related operators [12]. These systems already have efficient and well-tuned implementation of the relational operators that can be reapplied to querying XML streams. Nevertheless, they lack support for XPath [8] expressions, which are used to navigate through XML documents in most XML query mechanisms such as **XQuery** [5] and **SQL/XML** [12].

This paper discusses the design and implementation of TurboXPath, a prototype system that processes path queries over XML streams. TurboXPath accepts a subset of the XQuery **for-let-where** construct with some of the variables marked to be bound out. The result is a sequence of tuples of variable bindings representing document fragments or atomic values. In this paper we focus on describing how TurboXPath is applied over XML streams; however, most of the techniques discussed here are also applicable to prestored XML. Streaming algorithms, as the one presented in this paper, are well suited for processing queries specified in order-preserving languages such as XQuery. As opposed to relational queries where, in the absence of explicit ordering statements, the plan can produce the result in any order, XQuery requires the order of nodes to be preserved based on the order in which they appear in the input document.

While there are several implementations of XPath/XSLT that can be adapted for path processing in an XQuery engine, these are inadequate for the task. One obstacle in using the current XPath/XSLT technology in conjunction with an XQuery engine is the mismatch between the tuple-oriented model of the XQuery **for-let-where** construct and the node set model of the XPath processors. The same discrepancy is present when comparing the node set model with the relational algebra used in relational database engines. Retrieving multiple values from an XML document corresponds to retrieving multiple columns from a relational table and is very often needed. Achieving this goal for XML streams using the available XPath processors requires either materialization of the whole input document or multiple passes over the stream, which is not always possible, as in the case of data streamed from Web services.

Recently an XQuery implementation [13] and some XPath implementations [14,15,18–20] that operate over XML streams have been reported. However, none of these approaches addresses the issue of extraction of multiple bindings in tuples and adapting the processing model to relational database engines. We survey these approaches in the related work section and compare TurboXPath with some of them in the experimental evaluation section.

This paper presents a design description and experimental evaluation of the TurboXPath system. The distinguishing features of TurboXPath are:

- Queries are evaluated in a single, document-order pass over the input document stream.
- Tuples of correlated document fragments (bindings) are produced as results. The result is in document order (as defined by the XQuery standard) with duplicates removed.
- The input XQuery query is translated into a parse tree with multiple output nodes that, in conjunction with an event stack, is used to process the query. TurboXPath avoids the translation of the query into a finite state automaton (FSA), a technique used in some of the related approaches [16, 18,19], and the creation of automata states at runtime [14, 15], which is very expensive computationally and might

degrade query performance in scenarios where several independent queries are executed in parallel over different XML streams.
- TurboXPath supports a complex set of XQuery features including **for**, **let**, and **where** clauses with XPath paths composed of steps using **child**, **descendant**, **self**, **parent**, and **ancestor** axes; any node test ('*'); functions, and arithmetic and structural predicates. In conjunction with a new XML data type and some new XML-related operators, TurboXPath can provide the remaining functionality for full XQuery runtime in relational database engines.
- TurboXPath operates over any document, including cases when the document is recursive with regard to the query. We define those cases and discuss how they impose harder processing requirements on streamed XQuery processors.

The main contributions of this paper include:

1. The description of an XQuery processing system that works as a runtime operator in a relational engine (Sect. 2).
2. A new algorithm for XQuery processing over streams that avoids the translation to finite-state automata and the creation of automata states at runtime (Sect. 2.2).
3. The definition of when a document is recursive with respect to a query and the impact of recursive documents in the processing of XML streams (Sect. 2.2).
4. The description of optimization techniques for predicate evaluation in XQuery (Sect. 2.3).
5. Experimental results that show orders of magnitude improvement when comparing TurboXPath with other XML processing systems (Sect. 3).

The rest of this paper is organized as follows. Section 2 contains the technical core of the paper describing the design of the TurboXPath processor. Section 3 presents an experimental evaluation of our prototype implementation. Section 4 discusses related research projects and alternative approaches. Section 5 concludes the paper and presents our future research directions.

2 TurboXPath architecture and implementation

To provide context for the subsequent discussion, we first illustrate how TurboXPath fits into the architecture of an XML-enabled database engine. In such database engines, XQuery queries are translated into execution plans where the path processing operators have the same role as table accesses in traditional query evaluation pipelines. The fragments extracted by the path processing operators can then be transformed and combined by traditional relational operators such as joins, grouping, correlation, sorting, etc. and by XML-specific operators, like the XML generation operators used to process the **return** clause of XQuery. Figure 1 illustrates an execution plan for the multidocument XQuery query:

```
for $c in doc("d1")//customer
for $p in
 doc("d2")//profiles[cid/text() = $c/cid/text()]
for $o in $c/order[date = '12/12/01']
return
  <result>
    {$c/name} {$p/status} {$o/amount}
  </result>
```

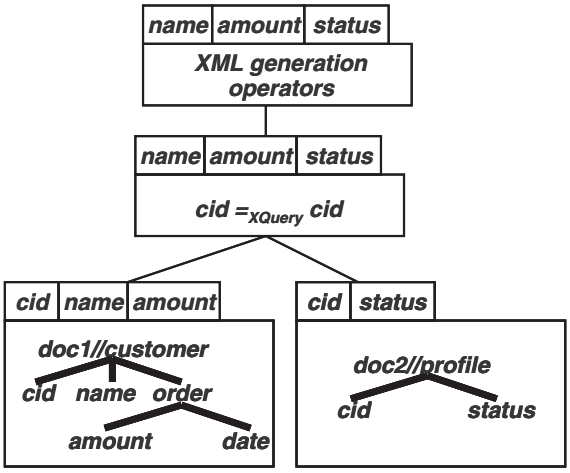


Fig. 1. Example query evaluation plan

In this plan, the query is decomposed into two single document accesses that produce tuples of XML fragments from "d1" and "d2". These tuples are then combined by a join on "cid", using the XQuery equality operator. Finally, XML generation operators are used to format the result as specified by the **return** clause.

The documents against which queries are evaluated can either be stored locally in the database or obtained from other XML-enabled data sources over the network. The TurboXPath component takes an XML document (stream) identifier and an XQuery fragment passed by the compiler. It generates an intermediate result with n XML data type columns as output, where n is the number of variables that must be extracted from the stream in order for the other components to process the query. In Fig. 1 the operator on the bottom left represents one invocation of TurboXPath with the query fragment:

```
for $c in
 doc("d1.xml")//customer[order/date="12/12/01"]
let $cid := $c/cid/text()
let $name := $c/name
for $o in $c/order
let $a in $o/amount
return-tuple $cid, $name, $a;
```

This query fragment is derived from the sample query and retrieves all the variables from "d1" that are required to fully evaluate the query. A correspondent XQuery fragment is used for "d2". The processing of the "d2" stream is represented in the query plan by the TurboXPath operator on the bottom right of Fig. 1.

The **return-tuple** clause in the query fragment example is used to identify the variables to be bound out. It does not conform to the XQuery standard and is used here only to describe the internal tuple-based interface between the runtime operators in the system. In the example, the TurboXPath operator for "d1" returns tuples with three bindings (document fragments), one for each bound-out variable.

TurboXPath can process both **for** and **let** clauses. **Let** clauses are processed in a manner similar to that of the **for** clauses, except that multiple results are grouped together and returned as a single sequence of XML fragments. Such pushing of grouping into the TurboXPath operator is simple to perform while it reduces the number of tuples flowing through the

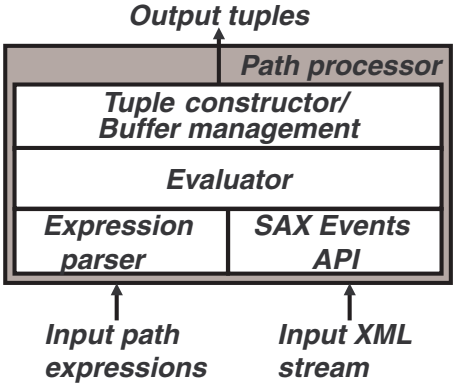


Fig. 2. System architecture

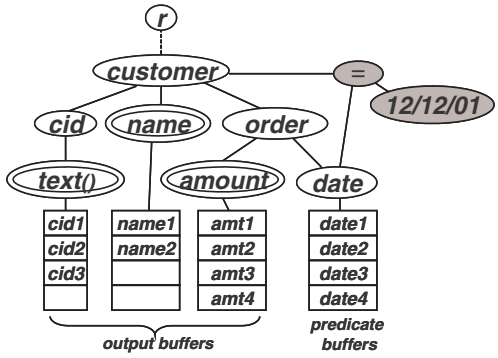


Fig. 3. Parse tree example

engine. Since the processing of **let** and **for** clauses is similar, in the rest of the paper we only describe the processing of the **for** clause queries.

The main components of TurboXPath are the *expression parser*, the *evaluator*, and the *tuple constructor/buffer manager*, as illustrated in Fig. 2. The input path expressions are parsed and connected into a single parse tree (PT) with multiple output nodes. Intermediate results representing XML fragments retrieved from the input document are stored in buffers associated with the output and predicate nodes. During document processing, a SAX parser generates events from the input XML stream. The evaluator uses these events to perform the state transitions and populate the buffers. It is also responsible for triggering the tuple construction module when the output buffers contain enough information to output result tuples. The following sections detail each of these components.

2.1 Expression parser

The expression parser is responsible for parsing the set of path expressions and producing a single parse tree (PT). Nodes in the PT correspond to node tests in the input path expressions while edges correspond to the relationship between node tests in the query. In the case when a node test in the query is followed by a predicate containing multiple branches, or when several expressions are rooted in the same variable, the corresponding PT node has multiple children. Figure 3 illustrates the PT generated by parsing the query of the bottom left operator in Fig. 1. Each PT has a special root node at the top, represented by "r" in Fig. 3. Each nonroot node is annotated with an axis indicator, indicating one of the XQuery step axes. There are several axes defined in XQuery (some optional), allowing each document node to be reached at any point of the processing. The discussion in this paper concentrates on the processing of the **child** and **descendant** axes. The other forward axes – **attribute**, **(descendant-or-)self**, and **following(-sibling)** – are supported with minor additions to the presented mechanism. For example, node tests with the attribute axis are represented in the PT by separate nodes and are handled in a similar fashion as element node tests; steps with self and descendant-or-self axes are supported by a slight addition to the matching mechanism described in Sect. 2.2. In [3] we described query tree rewrites for handling of the **ancestor** and **parent** axes.

The PT in Fig. 3 contains nodes with child and descendant axes. We use dotted lines to represent descendant axes and solid lines to represent child axes. The "customer" node has three children. The nodes corresponding to the variables listed in the **return-tuple** clause are called *output* nodes. Any PT node, including an internal node, can be an *output* node. Output nodes can also be descendants of other output nodes. In Fig. 3 the output nodes "name", "amount", and the text node are distinguished from the other nodes by double circles.

A PT node may also have a set of associated predicate trees. Each predicate tree is *anchored* at a PT node, called the *context node* of that predicate. In the example, "customer" is the context node for the predicate on the order date. Predicate tree nodes are shown in gray in the figure. Predicate trees are composed of leafs that are either constants or pointers to nodes in the PT subtree rooted at the context node. Internal predicate nodes are operators as defined in the XQuery/XPath standard specifications.

2.2 Evaluator

The evaluator is the central component of TurboXPath. It uses the PT to process the stream of SAX events generated from the input document to identify the fragments that are to be extracted and returned to the database engine. The PT is *static*, meaning that it does not change during processing, and can be reused over several documents. Besides the PT, the evaluator uses three *dynamic* structures that change during query evaluation depending on the input document:

- *Output buffers*: store the intermediate results that can be part of the result tuple.
- *Predicate buffers*: store the content of nodes participating in predicate expressions.
- *Work array* (WA): used to match the document nodes with query steps and to support existential predicate evaluation.

Figure 3 shows predicate and output buffers for the example query. The WA represents an inlined tree structure and can be compared in function to the DOM tree of the traditional XPath processors. An important difference is that the WA represents only the "interesting" portions of the tree, based on the already seen input. Furthermore, the WA is specifically designed for efficient processing of the query as opposed to the dual (traversal and query processing) purpose of the DOM representations in XPath/XSLT processors. During document

processing, the WA changes depending on the input. Each WA entry has four fields:

- Pointer to the corresponding PT node,
- Document level at which the entry was added to the array,
- References between parent-child WA entries,
- Status flag, used during the processing to indicate if the corresponding document node has satisfied the query conditions.

The SAX events produced by parsing the input document are transformed into evaluator events of the form $(name, type, document\ level)$, where $name$ is the node test name and $type$ is the event type, which can be OPEN, CLOSE, ATTRIBUTE, COMMENT, or PI. The document level is maintained by the SAX event handler by simply counting the OPEN and CLOSE events. By convention, the document root appears at level 0. The processing of a document starts with a (ROOT, OPEN, 0) event and ends with a corresponding CLOSE event.

The evaluator works by trying to match incoming events to all the WA entries. A *match* occurs when both the document levels and the names of the event and the WA entry are the same. A match also occurs when only the names match if the entry corresponds to a descendant path step (in this case the document level is ignored). On the other hand, when an entry corresponds to an *any node* test PT node (represented by "*" in XQuery), the name comparison always returns TRUE and only the document level is considered. WA entries corresponding to attributes, comments, and processing instruction (PI) node tests match only events of type ATTRIBUTE, COMMENT, and PI, respectively. The ATTRIBUTE events for the evaluator are produced by iterating over the attributes in the StartElement SAX handler. ATTRIBUTE, COMMENT, and PI events are handled in a manner similar to two consecutive OPEN and CLOSE events and are not discussed further in the paper.

The following actions are performed by the evaluator when a match is found for OPEN and CLOSE events:

- **OPEN:** For each child of the PT node corresponding to the matched WA entry, a new child WA entry is added, carrying the current document level incremented by one. The children added for the same WA match compose a *sibling group*.
  When the matched WA entry corresponds to a leaf node in the PT, no new entries are added to the WA. In this case, if the PT node is not an output node, the status flag of the matched WA entry is set to TRUE, indicating that all the conditions for this node have been satisfied. For each matched entry corresponding to an output node in the PT (either leaf or not), a buffer is created to save its content. This buffer is then added to a list of active buffers. During processing, every SAX event is forwarded to all the active buffers. In our current prototype implementation we use UTF16 textual representation for the buffered document fragments. When parsing a portion of the input stream that is to be buffered, the content of each event is translated from its original encoding into UTF16 and added to all active buffers.
- **CLOSE:** For every output node (either leaf or not), the CLOSE event removes the buffers associated with the matched node from the list of active buffers. For the leaf output node the CLOSE event also sets its status in the WA to TRUE. This change of the status indicates that the matching was satisfied and that the results are available in the corresponding output buffers. CLOSE events have no effect on leaf entries that are not output nodes since their status can be updated on OPEN events.
  If the matched node is an intermediate PT node (nonleaf), the WA must contain a sibling group that was added when the same node was matched with a corresponding OPEN event. During the CLOSE event, the status of the node must be evaluated by checking the status of its sibling group entries. For simplicity, let us consider that there are no predicates involved (predicates will be discussed in detail in Sect. 2.3). In this case the status flags of the node is computed by AND-ing the status flags of its sibling group. At this point, the sibling group entries are removed from the WA. The matched WA entry, however, remains to be used when its parent node is closed. Furthermore, if the status of the matched WA entry was previously set to TRUE, it remains so even if the evaluation of its status returned FALSE. This allows for the existential semantics of XQuery where a path is satisfied over a set of children nodes if any of the nodes satisfy all the conditions (and not necessarily the last one).
  Note that if the status of the node evaluates to FALSE, the buffers added between the matching OPEN and CLOSE events need to be purged from the queues. The mechanism to identify these buffers is presented in Sect. 2.5.

State transitions in the evaluator are represented by changes of the content of the WA. To illustrate the processing, we use a simple query and a small document stream, shown in Fig. 4. The state of the evaluator after each event is represented in the figure by a snapshot of the WA. The event leading to a snapshot is given at the top. In each entry, a node test name is used to represent pointers to the corresponding PT nodes. The document level is shown in the lower right corner, and the evaluation status (TRUE/FALSE) is in the upper right corner. Entries matching node tests that are performed over the descendant axis have "*" instead of a document level number. A link on the left side of the WA entries is used to relate multiple entries from a single sibling group. The references between parent/child WA entries are omitted for clarity.

The array grows with events matching nonleaf PT nodes. For each occurrence of the two consecutive "a" elements in the document, one sibling group consisting of entry "c", and "b" is added. These sibling groups are removed when the corresponding "a" nodes are closed. Note that an entry for "a" is added to the array before the first "a" in the document is seen and persists after the last "a" is closed. This is due to the fact that the WA represents not only the important nodes that have already been seen in the document, but also the nodes that we are looking for. The status of the "a" entry is set when the first "a" is closed. Once set to TRUE, the status is unchanged until the entry is removed from the array. In the example, this is apparent when the second "a" node does not satisfy the condition (there is no "c" child), and the status of the "a" entry remains TRUE. This principle allows the same data structure (WA) to be used for keeping track of which conditions have been satisfied so far as well as for detecting relevant document

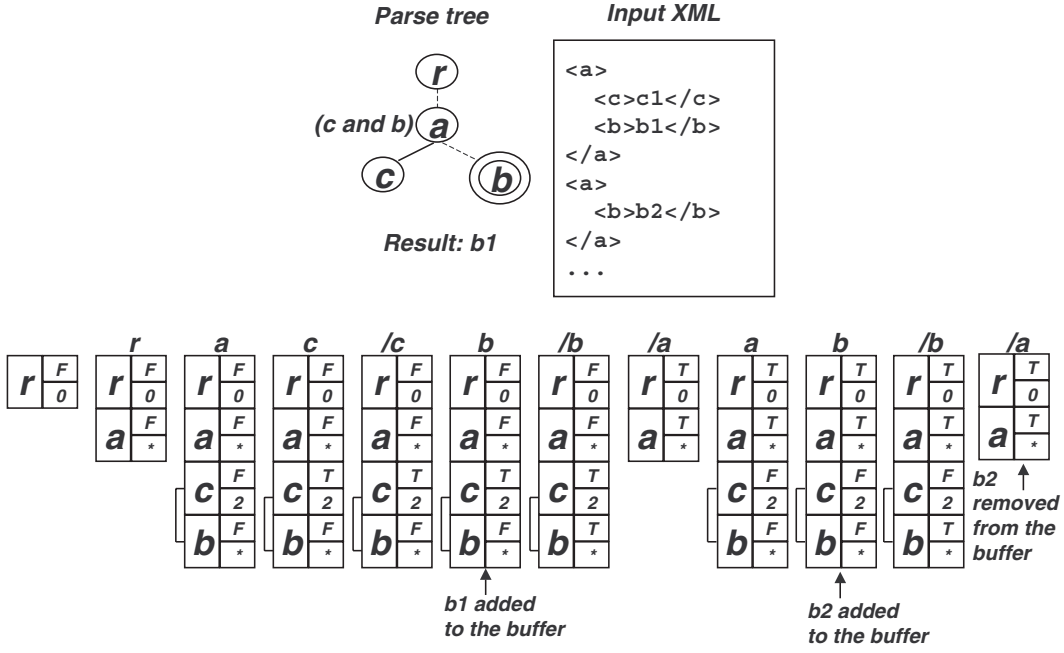


Fig. 4. Processing of the query //a[c]b

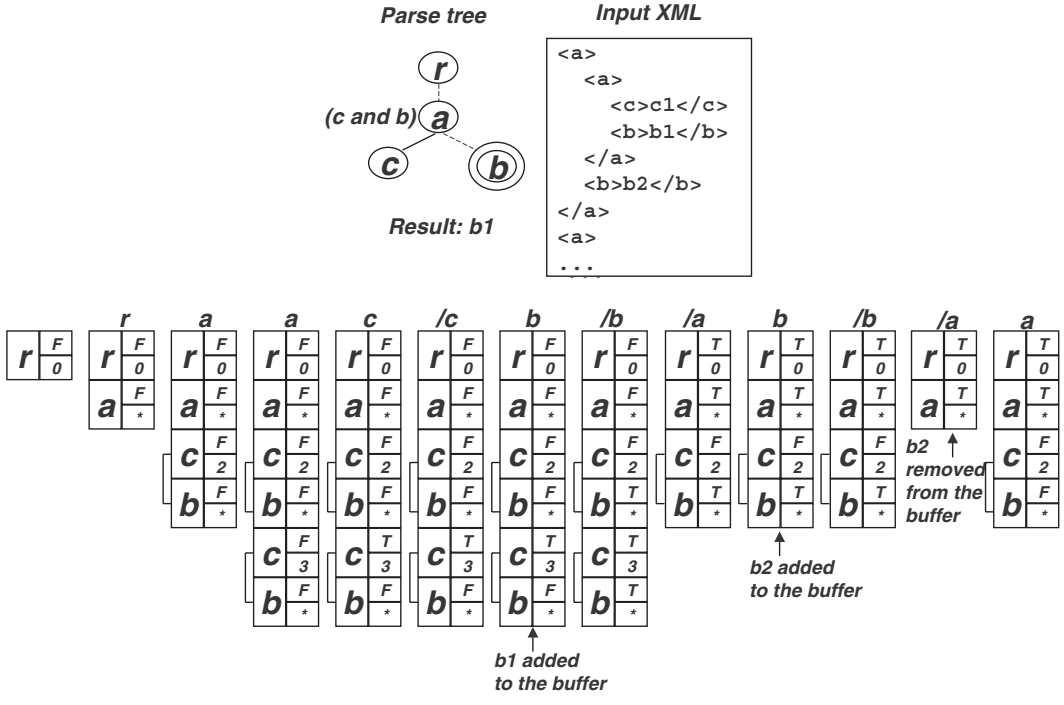


Fig. 5. Processing of the query //a[c]b over a document recursive w.r.t. the query

nodes. The status of the root "r" entry mirrors the status of its only child. TurboXPath uses this optimization to be able to use the root node status as an indicator in containment queries.

The status of the "c" entry is set to TRUE when the "c" element is open. The "c" node is not an output node, and therefore its condition is satisfied when we encounter (OPEN) the first "c" under the current "a" node. As "b" is an output node, the status of "b" WA entries is changed when a "b" element is closed, since only then are the output buffers complete and able to be used to generate the tuples.

One difficulty in designing a streamed XML path processor is to provide correct behavior when the input document is *recursive* with regard to the query.

Definition. A document D is recursive with regard to a query Q iff there exist two document nodes $n_1,\ n_2 \in D$ such that n_1 is an ancestor of n_2 and both n_1 and n_2 match the same node q in the query tree.

The document in Fig. 5 is recursive with regard to the query "//a[c]b" since the nested "a" nodes match the "a" node in the query tree. The same document is not recursive with regard to

the query "/a/a" since each "a" node of the document matches a different node in the query tree. A document does not need to have a recursive structure (DTD or Schema) to be recursive with a query: every document is recursive w.r.t. to the query "//*". For a document to be recursive w.r.t. a query, the query must contain at least one step with a descendant axis. In the following discussion we refer to processing documents that are recursive w.r.t. the query as "recursive cases".

Due to the nature of the streaming data, retrieving fragments and constructing tuples in recursive cases requires the processor to evaluate conditions for several elements simultaneously and calls for more elaborate solutions. While not common, correct handling of recursive cases is important to ensure correct evaluation over any input document.

In nonrecursive cases the WA has at most one entry for each PT node, limiting the size of the array to the size of the query. The WA is therefore preallocated to this size to limit the memory management calls to the operating system. In recursive cases the WA size can increase over this limit due to multiple WA entries corresponding to the same PT node. This is necessary since WA entries for recursive elements will be parents of multiple sibling groups, each representing an independent thread of control evaluating one of the recursive occurrences of the element. In the general case, the maximum size of the WA is therefore proportional to the product of the degree of recursion and the maximum fanout (number of children) in the PT. Recursive cases are not common, so WA entries for the recursive elements are allocated dynamically from the memory heap. An important point is that the memory required for the WA does not grow exponentially with the query size, which is an improvement over techniques based on automata [16,18, 19], as will be discussed in Sect. 4.

Figure 5 shows a recursive case where the document from the previous example is modified so that instead of two consecutive "a" elements the second "a" element is nested within the first. In this modified example, after the second "a" element is opened, there are two "a" sibling groups composed of "b" and "c" entries, one for each "a" element. The "c" event matches only the WA entry with the appropriate document level. This reflects the fact the "c" entry is a child of only one of the enclosing "a" elements. The "b" events, on the other hand, match both "b" entries since the "b" node test is specified using the descendent axis. Upon closing of the inner "a" element, the flags in the WA entries of its sibling group are AND-ed. The status of the "a" entry is changed to TRUE since both entries have status TRUE. This is not the case when the outer "a" element closes since there is no "c" child in this case. While this does not have an effect on the "a" status flag, which remains TRUE, it results in a dropped buffer for the second "b" element since it does not participate in any output tuple (buffer management is further discussed in Sect. 2.5). Note that, although the second "b" element is evaluated to TRUE, it does not change the status of the flags already set to TRUE by the first "b" element, reflecting the fact that a descendant "b" had already been found.

2.3 Predicate evaluation

The predicates are evaluated when the document node matching the anchor PT node is closed. Terms of boolean predicates

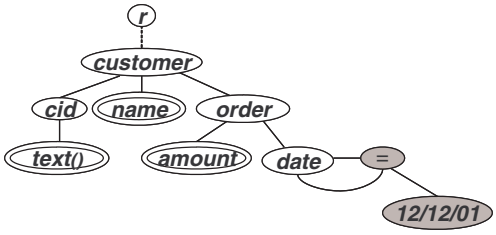


Fig. 6. Predicate pushdown example

that are simple paths are evaluated by using the values of the status flags in the WA entries of the sibling group corresponding to the matched entry. In the example in Fig. 5, when an "a" node is closed, the status flags of the "b" and "c" entries are AND-ed by the predicate anchored at the "a" PT node. However, in the general case, predicate evaluation may require non-boolean values stored in *predicate buffers*. In the example in Fig. 3 order dates are stored in predicate buffers during the processing of each customer node. When the "customer" element is closed, the predicate is evaluated over the order dates.

As with output buffers, during expression parsing, all the nodes that need to be buffered in predicate buffers are marked with a flag in the PT and the system allocates the predicate buffers for storing their content. In addition, both predicate and output buffer queues preserve document order. This is a requirement for predicate buffers in order to support positional predicates and XPath 1.0 compatibility mode casting rules that are order dependent.

As a strongly typed language based on the Query Data Model [5], XQuery requires data types to evaluate the operators. For example, the type of the "date" element in the example in Fig. 3 determines if the generic equality operator is interpreted as an equality between integers, strings, or date type items. The data types in our prototype are added to the SAX events by a modification of the Xerces validating parser and the SAX interface. While there are several technical challenges in providing correct type information in a SAX parser, these are beyond the scope of this paper, and we will ignore the typing issues in the rest of the discussion. Each buffer entry is then annotated with its type for use in the operator evaluation.

In general, a predicate can be completely evaluated only when its anchor node is closed. The predicate in the expression customer[evenings_phone ! = daytime_phone], for example, can only be evaluated when the "customer" element is closed since the XQuery existential semantics requires that every combination of "evening_phone" and "daytime_phone" is tested for a match. However, predicates that refer to only one PT node can be eagerly evaluated, increasing the system performance and reducing the buffering requirements. In these cases, TurboXPath applies a rewrite named *predicate pushdown*. To illustrate this rewrite, we look back at the example based on Fig. 3 where we had to accumulate all order dates under a customer before applying the predicate. Figure 6 shows an alternative representation of the same query where the predicate is anchored at the "date" node, instead of being anchored at the "customer" node as in the nonoptimized PT. In this case the predicate is evaluated for each "date" node without having to accumulate several values before evaluation.

2.4 Tuple construction

As the tuple processing model of the database engine requires matching (joining) the retrieved XML fragments with tuples, if several fragments for one or more of the tuple variables appear before the tuple is completed, these fragments must be buffered. For example, when returning tuples <a, b>, TurboXPath must buffer all "a"s that appear in the document before the first "b" appears (or vice versa). Furthermore, only tuples where the fragments satisfy the structural constraints of the query should be emitted. The goal of the tuple construction phase is to construct the correct tuples, out of all possible tuples in the cross product of the output buffers.

Since in the streaming model the document is unavailable at tuple construction time, each buffer must be annotated with enough information to complete the tuple construction. In order to preserve the position of the buffer content in the input document tree, a unique node ID (NID) is assigned to each matched node in the input document. Document nodes that do not match query nodes are ignored. Each buffer is annotated with its ancestor NIDs grouped into sets corresponding to the matched query nodes. We refer to these sets as *ancestor sets* (ASs). Each buffer has one ancestor set per ancestor in the query tree. Each set contains the NIDs of the document nodes matching the corresponding query node. An AS contains more than one NID only in recursive cases, where the same query node is matched by more than one document node.

In the following discussion we first present a simple nested loop join (NLJ) algorithm for the tuple construction. Then we show how simple improvements to this algorithm allow for an execution model similar to a merge join, which is worst-case optimal in time (the execution time is linear with the size of the result). We illustrate the process using the following query over the document fragment in Fig. 7. The example document represents a customer-order hierarchy and is close in structure to the running example used in the rest of the paper. In order to demonstrate some technical issues with recursive cases we assume that the "order" elements can be nested within each other. To simplify the presentation we also assume that the NIDs assigned to the elements in the example are the numbers given after each start element tag.

```
for $customer in document("...")/customer
  for $name in $customer/name/text()
    for $order in $customer//order
      for $date in $order/date
        for $amount in $order/amount
          return-tuple $name, $date, $amount;
```

Since there are three variables in the output tuple, the system keeps three buffer queues. After processing the example document, the queues contain all together 7 buffers. The result, however, contains only 3 tuples as opposed to 12 produced by the cross product of all the buffer queues. Some of the tuples are pruned from the result since they do not satisfy the structural constraints of the document. In the example, "amount" and "date" elements appear in the same tuple only if both were found under the same "order" element. This reasoning can be applied recursively up the parse tree: "name" buffers join with <amount, date> tuples that appear within the same "customer" element.

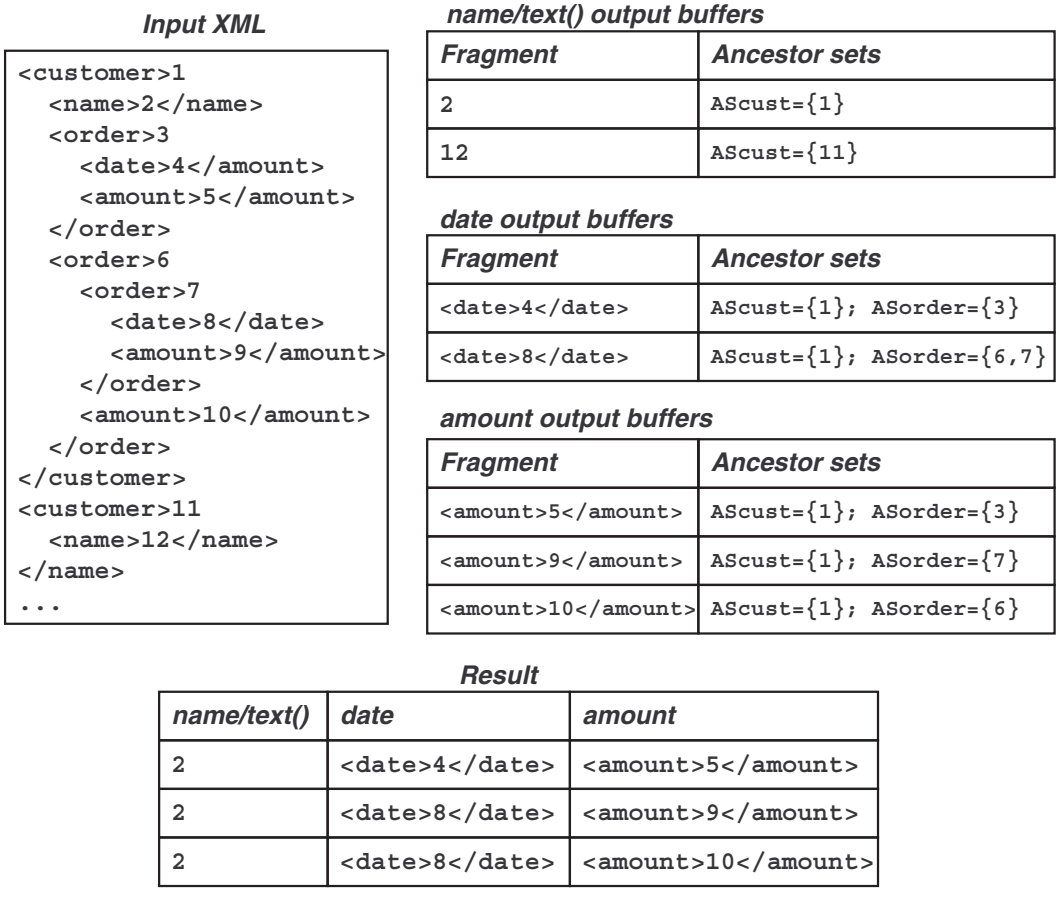
The NLJ tuple construction algorithm builds new tuples by starting from an empty tuple without any fragments (buffers) and adding buffers to it. The algorithm iterates over the buffers, and for each buffer it checks if the addition of that buffer to the tuple would generate a valid (partial) tuple. A tuple is valid if the intersection of each of all nonempty ancestor sets associated with that buffer and the corresponding nonempty ancestor sets of the tuple are not empty. The tuple ancestor set is an intersection of the corresponding ancestor sets for all the buffers in the tuple so far. When all variables are bound in the tuple, the tuple is complete and is emitted.

In this example, the algorithm would start by adding the first buffer (NID = 2) of the first buffer queue ("name") to the initially empty tuple. This would change the tuple ancestor set corresponding to the "customer" query node $TupleAS_{cust}$, which would be initialized from the buffer ancestor set $TupleAS_{cust} = AS_{cust} = \{1\}$. Next, the algorithm would try to bind values to the remaining two tuple columns by selecting the first buffer (NID = 4) from the second buffer queue (date). In this case, since the intersection of $TupleAS_{cust}$ and AS_{cust} for NID = 4 is nonempty, the binding is added to the tuple. In addition, since NID = 4 has an ancestor set corresponding to the query node "order", the set $TupleAS_{order} = \{3\}$. The algorithm would then proceed for the first "amount" fragment (NID = 5). The correctness of the tuple would be verified by checking the intersection of $TupleAS_{cust}$ and AS_{cust} for NID = 5 and the intersection of $TupleAS_{order}$ and AS_{order} for NID = 5, which are both nonempty. Therefore, the first tuple would be constructed with buffers representing the elements with NIDs 2, 4, and 5. Following the nested-loop pattern, the "amount" fragment with NID = 5 would be replaced in the tuple by the next one in the same queue, which has NID = 9. In this case the intersection of AS_{order} and $TupleAS_{order}$ would be empty, and the tuple for NIDs 2, 4, and 9 would not be emitted. The algorithm then proceeds for the remaining combinations of output buffers, generating the tuples in the $Result$ table of Fig. 7.

The tuple construction algorithm emits each tuple once, independently of the cardinality of the intersections of the ancestor sets of the participating buffers. This eliminates the need for duplicate removal operators on the top of the TurboXPath operator.

There are a couple ways to bring the performance of the algorithm above closer to the optimal. We first observe that intersection of only one ancestor set is needed to check if a buffer fits into a tuple. The intuition is that the path of a buffer intersects with the paths of the other buffers at least in one query node. In the example above this means that when the "amount" binding is added, we need to check the intersection only of the set AS_{order} since the ancestor paths of "amount" and "date" intersect at "order". If the intersection of the AS_{order} and $TupleAS_{order}$ is not empty, then the intersections of the sets corresponding to all the ancestors of "order" are not empty as well (in the example AS_{cust}).

Single ancestor set intersection reduces the complexity of the tuple construction per buffer queue entry. Nevertheless, performing NLJ when the buffer queues are long might still be inefficient. The "wasted effort" in the NLJ algorithm is due to the iteration over buffer entries that do not fit in the partially built tuple. For example, if the "customer" element in the example above contained a large number of "order" elements, each with several "date" and "amount" subelements, then the NLJ would try to match every "date" with every "amount",



Input XML

```
<customer>1
  <name>2</name>
  <order>3
    <date>4</amount>
    <amount>5</amount>
  </order>
  <order>6
    <order>7
      <date>8</date>
      <amount>9</amount>
    </order>
    <amount>10</amount>
  </order>
</customer>
<customer>11
  <name>12</name>
</name>
...
```

name/text() output buffers

Fragment	Ancestor sets
2	AScust={1}
12	AScust={11}

date output buffers

Fragment	Ancestor sets
<date>4</date>	AScust={1}; ASorder={3}
<date>8</date>	AScust={1}; ASorder={6,7}

amount output buffers

Fragment	Ancestor sets
<amount>5</amount>	AScust={1}; ASorder={3}
<amount>9</amount>	AScust={1}; ASorder={7}
<amount>10</amount>	AScust={1}; ASorder={6}

Result

name/text()	date	amount
2	<date>4</date>	<amount>5</amount>
2	<date>8</date>	<amount>9</amount>
2	<date>8</date>	<amount>10</amount>

Fig. 7. Tuple construction output buffers example

while only the "date" and "amount" elements within the same "order" parent would match.

To improve on this we observe that in nonrecursive cases all ancestor sets contain only one NID. Furthermore, the NIDs in the same ancestor set are monotonically increasing. This property stems from the preorder traversal of the document during the matching phase, when NIDs are assigned to XML fragments and the fragments added to the output buffers. Based on these observations, for nonrecursive cases the NLJ algorithm presented above can be modified into a multiway merge algorithm. The merge join algorithm requires very small modification to the NLJ algorithm. The difference is only in how the merge algorithm moves through the buffers when a buffer does not fit into the tuple. The NLJ continues to the end of the queue, while the merge join algorithm backtracks to the first queue position with the current NID. Furthermore, when fitting a new element after the backtracking, the merge join algorithm continues at the first buffer with an NID larger than the current one, as opposed to the beginning of the queue as in the NLJ algorithm.

We note here that in a recursive case the merge algorithm is not applicable. For example, in Fig. 7 the AS_{order} of fragment 9 contains NID 7, while the following fragment 10 has an AS_{order} with NID 6, violating the assumptions that the NIDs increase monotonically. This inversion is caused by the inner "order" element eclipsing the outer, as a parent of the element "<amount> 9 </amount>". Nevertheless, in the majority of cases the algorithm is applicable and its execution time is proportional to the number of the returned tuples. The class of the recursive cases for which NLJ is needed is detected by the evaluator, and the appropriate tuple construction algorithm is applied.

2.5 Buffer management

In most scenarios the tuples returned are composed of bindings from small *compartments* within a document. As tuples are generated by combining data within each compartment, the buffer queues can be cleaned between compartments. In this section we describe how to determine the boundaries of the compartments and the circumstances when a buffer can be discarded in order to release memory and lower the memory requirements of the processor.

Buffer elimination due to failed condition

The tuple construction algorithm described above requires that all the buffers in the output buffer queues satisfy the query conditions. Therefore, buffers that do not satisfy these conditions must be eliminated before the tuple construction starts. As described above, the status of each node is evaluated in its close event. Therefore, when an element is closed and not all the query conditions are satisfied, its NID is removed from all the ancestor sets of all buffers in system. If, for some buffer, this

removal results in an empty set for at least one of the ancestor sets, the buffer does not satisfy the query conditions and it is discarded. As described in the previous section, since in nonrecursive cases the NIDs in the buffer queue are monotonically increasing, all buffers to be removed are at the top of the queue. In such cases, hence, it is not necessary to scan the whole buffer queue to eliminate buffers but only up to the first buffer that is not removed.

An example of such a case is shown in Fig. 5 presented in Sect. 2.2. At the point when the outer "a" element is closed, the output queue of the "b" node contains two buffers, one for fragment "b1" and another one for fragment "b2". The first of these buffers has $AS_a = \{1, 2\}$, for both "a" nodes, while the later has $AS_a = \{2\}$. Since the conditions under the outer "a" (NID = 2) are not satisfied (no "c" child), the NID of the outer "a" is deleted from the ancestor sets of both "b" buffers. This leaves the buffer for "b2" with no valid "a" parent and the buffer is dropped from the queue. The buffer containing "b1" is not removed since its ancestor set for "a" still contains the inner "a" (NID = 1).

Buffer elimination due to expiration

As presented above, the tuple construction process assumes that all the fragments that participate in result tuples are extracted from the document and stored in buffer queues before the first tuple is emitted. For most documents this is not necessary. Often, fragments that participate in the result do not need to be kept in buffers until the end of the document. A buffer can be discarded after all the possible tuples that use this buffer have been constructed. To establish this point of *expiration* we define the concept of lowest common ancestor node (LCAN) as a parse tree node such that:

1. It is an ancestor of all the output nodes;
2. It is not a parent of any other node that satisfies 1.

Since each query has at least one output node, each query must have an LCAN. In fact there is exactly one LCAN for every query since if there existed two LCAN nodes they could not be related as child and parent due to condition 2 defined above.

During processing, the evaluator keeps track of the document level at which the outermost match to the LCAN is made (due to recursion there might exist several properly nested document subtrees with roots matching the LCAN node). At closing element action, if the closed level matches the outermost LCAN document level and all the conditions in the closing node are verified, the tuples can be emitted. In this case, after emitting the tuples, all the buffers are deleted and all the counters used to generate IDs for the document nodes are reset. In other words, the LCAN query node allows for identifying independent compartments of the document. Data within the compartments are processed independently of data within other compartments.

In the example shown in Fig. 7, the LCAN node is "customer". By the time the "customer" (NID = 1) closes, all the conditions are verified, the tuple construction process is trigged, the correct tuples are emitted, all the buffers are deleted, and all the counters are reset. Please note that since all fragments under the "customer" (NID = 1) are removed when it is closed, the second buffer for "name" (NID = 12) would not coexist with the remaining buffers shown in Fig. 7. In fact, by the time the second "name" is buffered it would be the only fragment in the system buffers and it would get NID = 2 since the counters would have been reset. As another example, let us consider the query

```
for $s in document(``...'')//store
  let $sn := $s/name
  let $cn := $s/customers/customer/name
  let $ca := $s/customers/customer/address
    return-tuple $sn, $cn
```

retrieving the store and the customer names applied to the document:

```
<store>
  <name>Sears</name>
  <customers>
    <customer>
      <name>Jim</name>
      <address>Jim's str.</address>
    </customer>
    <customer>
      <name>John</name>
      <address>John's str.</address>
    </customer>
    ...
  </customers>
</store>
```

The LCAN node is "store" since it is an ancestor of both output nodes. The names of the customers are kept until the end of the "store" element because another store name might appear at any point within this element, producing one more tuple for each customer. While possible, such document organization is very unlikely. To avoid unnecessarily large buffer queues, the processor needs to know either that there is only one store name or that they are all given before the customer entries. This is an example of where schema information about the input document can lower the memory consumption. If in the query the store name was not an output node, the LCAN would be lowered to "customer" and buffering time would be reduced.

Clearly the document organization can impact the performance of some queries. In the case where the user can influence the organization of the input document, the elements should be grouped by their tag names to allow for use of schema information as above. Another document organization rule that might decrease the memory consumption is that the bulkier extract elements should appear as late in the document as possible since the elements found earlier in the document are buffered for a longer time than those found later.

3 Prototype evaluation

The TurboXPath prototype was developed on Windows 2000 using Microsoft Visual C++ and has been tested both on Windows 2000 and AIX 4.3.3 platforms. To parse the input XML documents, TurboXPath uses the SAX API of the Xerces C++ [23] parser. Xerces is shareware software based on IBM's XML parser, which is distributed freely by the Apache Foundation. In the evaluation presented in this section we compare query completion time and peak memory consumption of the TurboXPath processor with the Apache Xalan XPath/XSLT

processor [22]. Xalan also uses Xerces internally, but it is based on the DOM API where the entire document is parsed and a tree representation is built into the memory. The processing of XPath queries is then performed over the DOM tree in memory. In order to support similar functionality as the TurboXPath processor, we have build a simple C++ module over the Xalan interfaces to stringify a DOM tree whenever an intermediate node is retrieved.

To test TurboXPath in conjunction with a relational database, we used DB2/UDB and encapsulated TurboXPath into user-defined functions (UDFs) for the three most common SQL/XML [12] functions: XMLContains, XMLExtract, and XMLTable. Each function was implemented using both TurboXPath and Xalan as the core path processor. XMLContains takes an XPath expression and an XML document as inputs and returns a boolean value indicating if the XPath is satisfied by the document. XMLExtract has the same inputs but it returns an XML fragment as output. Both XMLContains and XMLExtract are scalar functions. XMLTable, on the other hand, is a table function that takes a set of XPath expressions (or an XQuery expression with multiple outputs) and an XML document as inputs and generates a virtual table in which the columns correspond to the input expressions.

We also compare TurboXPath against two other publicly available XML streaming systems: XMLGrep and XSQ [20]. XMLGrep supports most of XPath 1.0; however, its design and implementation details have not been published. We are unaware, for instance, which XML parser is used in XMLGrep. XSQ is one of the most complete XML stream processing systems described in the literature. The evaluation section in [20] shows that XSQ delivers superior performance as compared with several other engines. XSQ was developed in Java, and it uses the Xerces Java SAX API for parsing. TurboXPath was compared to these two systems using a standalone version not based on UDFs.

We used four different data sets for running the experiments. The first one is a set of files from the DBLP publication database, varying from 900 KB to 7000 KB. The second data set is product reviews collected at an online product review site. The individual reviews were aggregated into files of up to 400 MB. The experiments on this data set were measured in the standalone version since Xalan could not complete the execution over the large files as a DB2 UDF due to memory exhaustion. The third data set is a set of randomly generated files that have been generated from a set of also randomly generated XPath expressions using the XML generation tools described in [3]. These files vary from 400 KB to 8 MB. Finally, for the comparison against XMLGrep and XSQ we used XMark, which is a standard data set used for XML benchmarks. XMark provides an XML generator and a set of suggested queries. The XMark files we used vary from 30 MB to 120 MB.

The comparative measurements of the execution times and memory consumption presented in Figs. 8 and 9 were obtained on an IBM ThinkPad T20 with a 700-MHz processor and 256 MB of RAM and Windows 2000. All experiments were performed with warm cache. Table 1 presents the queries used in each graph. The queries were chosen in the way most of the TurboXPath features are exploited. The queries used for XMark are based on the suggested queries but have been modified to accommodate the features not supported by XMLGrep and XSQ.

Figures 8a and b illustrate the performance of the XMLContains boolean UDF in both DBLP and the randomly generated files. In both cases TurboXPath outperforms Xalan by several orders of magnitude. An interesting point regarding XMLContains is the fact that, since TurboXPath uses incremental parsing, it does not need to go over the entire XML document to produce the result in case of positive queries. Figure 9a shows a "good" case in which the requested location path is probably at the beginning of the document and the response time is extremely fast. Figure 8b shows the average case in which the response time is bound by the XML parsing time.

Figures 8c and d present similar results for the XMLExtract UDFs. In this case, however, the parsing time is the actual lower bound since the documents have to be completely parsed in order to generate all the possible matching XML fragments. TurboXPath is on average only 50% slower than the parsing time, while Xalan can be up to seven times slower (however, note that Xalan uses DOM and not SAX). Figure 8e shows how XMLExtract performs on extremely large files. After the limit of 60 MB Xalan performance starts to decay exponentially, while TurboXPath continues to show good performance. The measurements indicate that TurboXPath clearly performs better. The difference is most notable when the main memory is exhausted and the operating system starts using the swap file for virtual memory pages. Figure 8f shows the memory usage for the execution in Fig. 8e. TurboXPath executable uses around a 1.7 MB image independent of the input file size, while the size of the memory used by Xalan's DOM tree grows to over 300MB.

Figures 8g and h illustrate the impact of selecting several outputs while at the same time using a TurboXPath implementation of the XMLTable UDF. Figure 8g is the result of a query in which the number of output tuples is independent of the width of the tuples (number of selected columns). Figure 8h, on the other hand, shows the result of a query in which the number of output tuples grows exponentially with the number of columns. Both queries are similar, the only difference being that in Fig. 8h we made the LCAN node to be "dblp" instead of "phdthesis", which made all the combinations of "title", "author", and "year" valid results. Note that in both cases the time to select one, two, or three columns simultaneously is almost the same, even though the number of result tuples varied from 20 to 8000 in the second case. This confirms that since TurboXPath uses a streamed execution model, visiting each document node only once, the additional cost of selecting more than one XML fragment is negligible.

XQuery processing often requires the extraction of many fragments from a document (or stream). As the experiments have shown, this can be done with no extra cost in TurboXPath. Therefore, the optimizer should generate query plans that maximize the number of XML fragments extracted by each invocation of TurboXPath. This is exemplified in Fig. 1, where the query plan generated by the optimizer makes only two TurboXPath invocations, one extracting three fragments from "doc1" (cid, name, and amount) and the other extracting two fragments from "doc2" (cid, and status). Since there the experiments have shown that there is no additional cost for selecting multiple fragments, this is a more efficient plan than

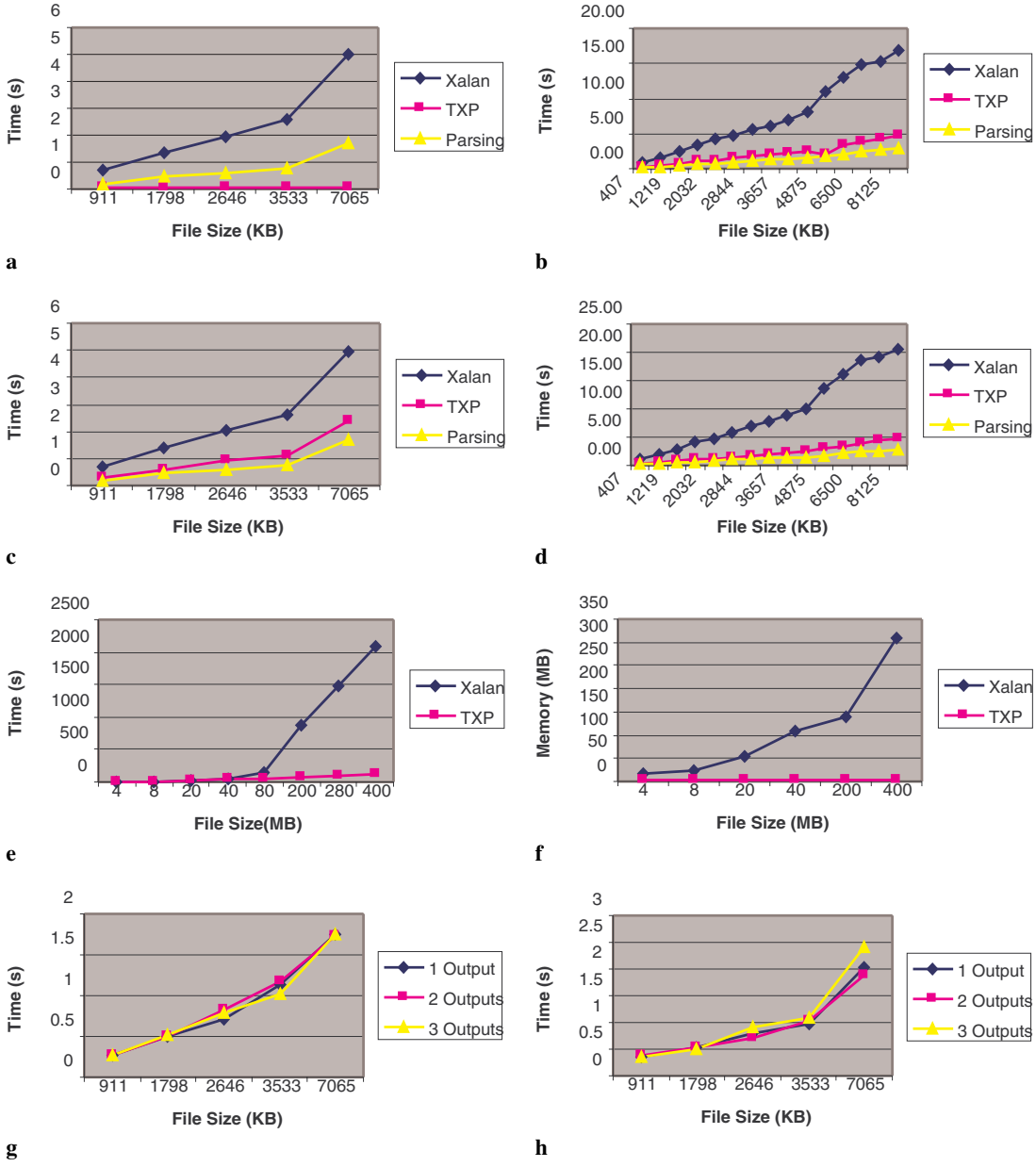


Fig. 8a–h. Experimental results. **a** XMLContains (DBLP). **b** XMLContains (random). **c** XML Extract (DBLP); **d** XMLExtract (random). **e** XMLExtract (review). **f** Memory usage (review). **g** Number of outputs (DBLP). **h** Number of outputs (DBLP)

Table 1. Queries used in the experiments

Figure number	Query
8a, 8c	/dblp/*[@key="conf/vldb/2001" and editor and title]/title
8b, 8d	//g0//l2[.//s2//d0]//t0[.//u0//b1//q1]//l3//c2
8e, 8f	//REVIEW[URL/text() and SEQNUM/text() and EXTRACTOR/text()]
8g	for $thesis in /dblp/phdthesis[year < "1994" and author and title] return-tuple $thesis/title, $thesis/author, $thesis/year;
8h	for $thesis in /dblp[phdthesis/year < "1994" and phdthesis/author and phdthesis/title] return-tuple $thesis/title, $thesis/author, $thesis/year;
9a, 9b	/site/people/person[@id='person0']/name/text()
9c, 9d	//closed_auction/price/text()

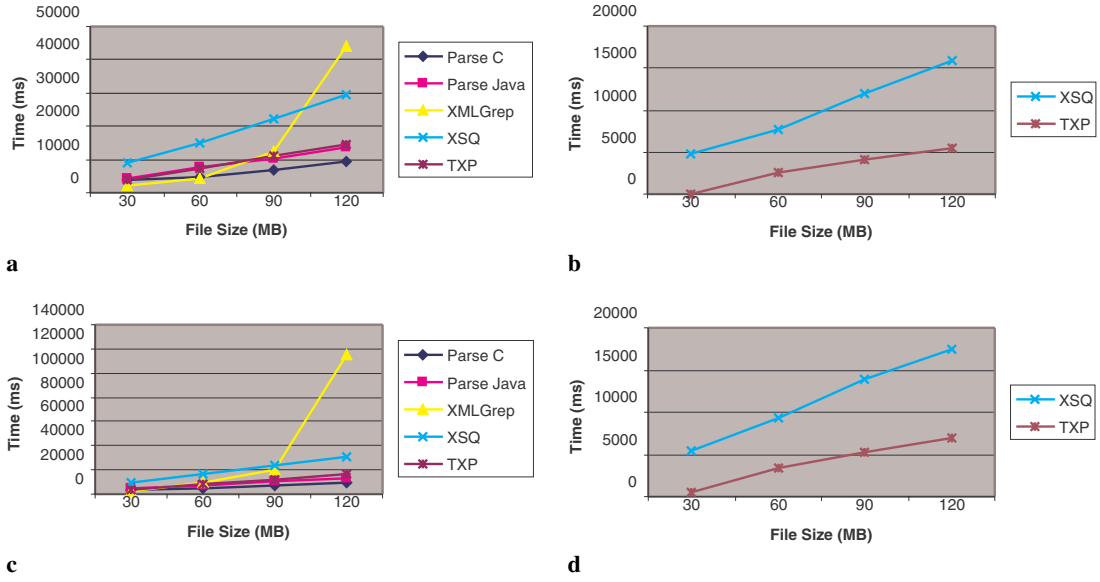


Fig. 9a–d. Experimental results. **a** Execution time (XMark Query 1). **b** Evaluation time (XMark Query 1). **c** Execution time (XMark Query 2). **d** Evaluation time (XMark Query 2)

one requiring five invocations of TurboXPath, each returning one fragment.

The graphs in Fig. 9 compare TurboXPath against XMLGrep and XSQ. Figure 9a shows how the performance of these systems varies with respect to the size of the input data in an extraction query that has predicates and returns only one XML element. This graph also shows the Xerces parsing time in C and Java, since XSQ is developed in Java. XMLGrep shows very good performance for small files, but the performance degrades dramatically for larger files. XSQ scales well, but it is consistently slower than TurboXPath. Figure 9b factors out the parsing time for both XSQ and TurboXPath. Note that we could not do that for XMLGrep since we are unaware of which parser it uses. Figure 9b shows that even factoring out the parsing time (parsing in Java is slower than parsing in C++), TurboXPath is still much faster than XSQ, and the slope of the curves show that it scales better than XSQ. Figures 9c and d show similar results for a different XMark query that involves a descendent step ("//").

In addition to scaling better than the other systems, TurboXPath is also more complete. Several of the queries proposed by the XMark benchmark cannot be executed in XSQ and XMLGrep. XQuery expressions that require multiple outputs, for instance, would require several invocations of XMLGrep or XSQ, which would result in even bigger processing times.

4 Related work

Most of today's commercial XPath processors are incorporated into XSLT platforms for document transformation. In this respect, their implementation assumes that the XML documents are entirely available at query time. They construct an internal representation of the documents (most commonly DOM trees) in order to efficiently evaluate the operators. This approach does not apply when the documents are streamed, that is, when only certain fragments of the documents are available at query time.

However, query processing in streaming environments is not a new topic. Extensive research in streamed query processing was conducted in areas such as information dissemination (ID). ID systems register thousands of user profiles and monitor several information sources for matches of these profiles. XFilter [1], for example, is an ID system based on XPath. It uses an automaton to process large numbers of simple boolean XPath queries over large numbers of documents. YFilter extends XFilter to use a nondeterministic automaton (NFA) in which the state transitions for multiple queries are precomputed instead of advancing an automaton for each query separately [10]. In contrast, the evaluator of TurboXPath was designed to avoid the translation to finite-state automata (FA) since such translation is computationally expensive and may generate an exponentially large number of states [14]. Since TurboXPath was designed to work in an environment with several concurrent users, memory consumption is an issue and the approaches based on automata may be unfeasible. On the other hand, ID systems are designed to handle workloads consisting of large numbers of Xpath expressions and it is not clear how TurboXPath would apply to that setting.

The evaluator presented in this paper has its roots in the FA theory. However, XPath queries involving branching require parentheses in order to verify the structural constraints and the matching and cannot be processed using pure FA. TurboXPath uses the document level and the WA to perform the state transition in a manner similar to the transitions of the nondeterministic FA (NFA), while being able to match the corresponding OPEN and CLOSE events. There is no notion of a single state in the evaluator as in FA; instead, a combination of flags and the content of the WA reflect the current state. In contrast with the FA approaches, the TurboXPath parse tree need not be transformed before use. The size of the tree is proportional to the query, and it is always smaller or equal to the corresponding automata, which in recursive cases is exponen-

tial in the size of the query [14]. Finally, the PT provides a natural context for the placement of predicates, a feature that has not been addressed by most of the other streamed XPath evaluation frameworks such as [16,18,19].

Another related ID system is XTrie [6]. XTrie differs from XFilter in that its index is based on decomposing the tree patterns into sequences of element names and indexing them with a trie. In addition, XTrie supports a bigger XPath subset when compared to XFilter, supporting queries with AND branches.

In the NiagaraCQ system [7,21] user profiles are expressed using continuous queries expressed in XML-QL [9]. Continuous queries allow users to get new results whenever an update of interest occurs in certain information sources. The NiagaraCQ project differs from our work in that it focuses on solutions for grouping large numbers of queries together and on developing efficient relational operators for variable-rate streaming environments, as opposed to the efficient evaluation of the path queries.

Tukwila [16,17] has been developed concurrently with TurboXPath and provides a similar set of features. The core of the Tukwila system is the X-scan operator for path processing. During streaming, Tukwila builds an intermediate tree representation that is then traversed by the X-scan operator to perform the state transitions. Portions of the tree no longer needed are discarded, with only the relevant pieces of the document being kept in memory. The tree manager has a function similar to that of the WA and the output buffers in TurboXPath. Since processing paths over an XML document requires parenthesis matching, which is not supported by regular expression automata, Tukwila uses a stack and a meta-automaton to enable and disable several deterministic finite automata (DFA) that represent linear subpaths in the original query. The automata are generated by first mapping a linear XPath subexpression to a DFA and then translating the DFA to a nondeterministic automaton (NFA), possibly producing an exponential number of states. Furthermore, the Tukwila buffer management as reported in [16,17] cannot handle cases where documents are recursive w.r.t. the query. Finally, in Tukwila predicates are not supported in the X-scan operator and are delegated to the relational engine. On the other hand, Tukwila supports IDREFS, a feature not yet supported by TurboXPath.

Recently an XQuery implementation [13] and some XPath implementations [11,14,15,18–20] that operate over XML streams have been reported. In [13], Florescu et al. describe an XQuery engine designed for data transformations on XML streams. Their system is targeted for small XML documents and is a standalone XQuery processor that does not rely on any database system. The system proposed in this paper, on the other hand, is designed to efficiently support any type of XML document, both streamed and persisted, and can be used in conjunction with a relational engine. The XPath processors proposed in [18–20] are based on connecting a set of FA in a network that represents the query. All these approaches suffer from the same problem of any implementation based on FA, which may require memory that grows exponentially with the size of the query. Of these implementations the one reported in [20] is the most complete, supporting branching and predicate evaluation. Performance comparisons between this system and TurboXPath have been described in Sect. 3. The systems reported in [14,15] are in the area of ID, and as such they support only boolean queries. Some theoretical bounds in the size of the automata are presented in [14], while [15] describes how predicates can be shared by different queries. The predicates supported by [15] are only boolean predicates for which no buffering is required. Furthermore, the approach of eliminating redundant computation among several queries proposed in [15] is not applicable to the database scenario, where several queries are executed concurrently over different XML input streams. An extension to YFilter that supports a limited subset of the FLWR clause in XQuery (the **let** construct is not supported) is described in [11] .

Several projects have explored processing XPath queries over preprocessed documents. Zhang et al. [24] proposed a region algebralike representation for the XML documents in which a document is stored in two relations with the following schema: {Term, DocID, StartPos, EndPos, LevelNum} for elements and {Term, DocID, StartPos, LevelNum} for document tree leafs. According to this paradigm, computation of a path expression is performed by joining these two structures. Complex path queries are divided into a sequence of one-step queries, and the steps are computed using joins. Al-Khalifa et al. extended this work with tree-merge and stack-tree joins [2]. The latter was extended to a new family of join algorithms called holistic twig joins [4]. None of these approaches was developed specifically for streaming.

5 Conclusions and future work

This paper presented the TurboXPath system, a streamed processor for processing **for-let-where** constructs of XQuery. The features supported by TurboXPath were chosen by examining the emerging XML query standards and identifying a common core. A distinguishing feature of the TurboXPath system is its capability to process query trees constructed of several concatenated paths returning tuples as results. The tuple-based interface is suitable for integration in an XQuery database engine using existing virtual table interfaces. The paper also presented an experimental evaluation of a prototype implementation of the TurboXPath processor, where it demonstrated orders of magnitude lower execution times and memory consumption compared to a DOM-based XPath processor and substantial performance improvement and richer set of features compared to other streamed XPath processors. Finally, the paper presented a novel XPath processing technique that does not require the translation of queries to FSAs. This new technique may require exponentially less memory to process queries when compared to the previous streaming algorithms based on FSA.

In our current work we explore expanding the use of TurboXPath over various binary XML formats using a custom SAX interface. The component producing the SAX events can skip irrelevant pieces of the document, allowing for improved performance. As opposed to an iterator-based interface between the XPath processor and the database storage, the SAX event-based interface allows for use of the same processing model for both streams and stored XML documents and requires no additional duplicate elimination operations.

References

1. Altinel M, Franklin M (2000) Efficient filtering of XML documents for selective dissemination of information. In: Proceedings of the 26th international conference on very large databases, Cairo, Egypt, 10–14 September 2000
2. Al-Khalifa S, Jagadish HV, Koudas N, Patel JM, Srivastava D, Wu Y (2002) Structural joins: a primitive for efficient XML query pattern matching. In: Proceedings of the 18th international conference on data engineering, San Jose, CA, 26 February–1 March 2002
3. Barton C, Charles P, Goyal D, Raghavachari M, Fontoura M, Josifovski V (2003) Streaming XPath processing with forward and backward axes. In: Proceedings of the 19th international conference on data engineering, Bangalore, India, 5–8 March 2003
4. Bruno N, Srivastava D, Koudas N (2002) Holistic twig joins: optimal XML pattern matching. In: Proceedings of SIGMOD, Madison, WI, 3–6 June 2002
5. Chamberlin D, Clark J, Florescu D, Robie J, Simeon J, Stefanescu M (2003) XQuery 1.0: An XML query language, W3C Working Draft, August 2003. http://www.w3.org/TR/xquery
6. Chan C-Y, Felber P, Garofalakis M, Rastogi R (2002) Efficient filtering of XML documents with XPath expressions. In: Proceedings of the 18th international conference on data engineering, San Jose, CA, 26 February–1 March 2002
7. Chen J, DeWitt DJ, Tian F, Wang Y (2000) NiagaraCQ: a scalable continuous query system for Internet databases. In: Proceedings of ACM SIGMOD, Dallas, TX, 15–18 May 2000
8. Clark J, DeRose S (1999) XML Path Language (XPath) Version 1.0. W3C Recommendation 16 November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116
9. Deutsch A, Fernandez M, Florescu D, Levy A, Suciu D (1999) XML-QL: A query language for XML. In: Proceedings of the WWW conference, Toronto
10. Diao Y, Fisher P, Franklin M, To R (2002) YFilter: efficient and scalable filtering of XML documents. In: Proceedings of the 18th international conference on data engineering, San Jose, CA, 26 February–1 March 2002
11. Diao Y, Franklin MJ (2003) Query processing for high-volume XML message brokering. In: Proceedings of the 29th international conference on very large databases, Berlin, 9–12 September 2003
12. Eisenberg A, Melton J (2002) SQL/XML is making good progress. SIGMOD Rec 31(2):101–108
13. Florescu D, Hillary C, Kossmann D, Lucas P, Riccardi F, Westmann T, Carey M, Sundararajan A, Agrawal G (2003) The BEA/XQRL streaming XQuery processor. In: Proceedings of the 29th international conference on very large data bases, Berlin, 9–12 September 2003
14. Green T, Miklau G, Onizuka M, Suciu D (2003) Processing XML streams with deterministic automata. In: Proceedings of the 19th international conference on data engineering, Bangalore, India, 5–8 March 2003
15. Gupta A, Suciu D (2003) Stream processing of XPath queries with predicates. In: Proceedings of ACM SIGMOD, San Jose, CA, 23–26 May 2003
16. Ives Z, Levy AY, Weld DS (2000) Efficient evaluation of regular path expressions on streaming XML data. Technical Report UW-CSE-2000-05-02, University of Washington
17. Ives Z, Halevy AY, Weld DS (2002) An XML query engine for network-bound data. J Very Large Databases 11(4):380–402
18. Ludäscher B, Mukhopadhyay P, Papakonstantinou Y (2002) A transducer-based XML query processor. In: Proceedings of the 28th international conference on very large data bases, Hong Kong, September 2002
19. Olteanu D, Kiesling T, Bry F (2003) An evaluation of regular path expressions with qualifiers against XML streams. In: Proceedings of the 19th international conference on data engineering, Bangalore, India, 5–8 March 2003
20. Peng F, Chawathe S (2003) XPath queries on streaming data. In: Proceedings of ACM SIGMOD, San Jose, CA, 23–26 May 2003
21. Viglas E, Naughton JF (2002) Rate-based query optimization for streaming information sources. In: Proceedings of ACM SIGMOD, Madison, WI, 3–6 June 2002
22. Xalan-C++, an XSLT processor. Apache XML project. http://xml.apache.org/xalan-c/index.html
23. Xerces-C++, a validating XML parser. Apache XML Project. http://xml.apache.org/xerces-c/index.html
24. Zhang C, Naughton JF, DeWitt DJ, Luo Q (2001) On supporting containment queries in relational database management systems. In: Lohman GM (ed) Proceedings of ACM SIGMOD, Santa Barbara, CA, 21–24 May 2001