

# QuickXScan: Efficient Streaming XPath Evaluation

**Guogen Zhang**  
IBM Silicon Valley Lab  
San Jose, CA 95141, USA

**Qinghua Zou**  
Microsoft Corporation  
Redmond, WA 98052, USA

**Abstract** - Many XML applications over the Internet favor high-performance single-pass streaming XPath evaluation. Finite automata-based algorithms suffer from potentially combinatorial explosion of dynamic states for matching descendant axes. We present QuickXScan for streaming evaluation of XPath queries containing child and descendant axes with complex predicates. Using a tree representation for an XPath query, it employs a matching grid, a compact tree of interrelated stacks as in the holistic twig join algorithms, to represent the matches and their relationships. QuickXScan fully utilizes transitivity for matching, thus reduces the number of active states to the query size in the worst case. It also evaluates expressions incrementally using propagation and iterative rules, and produces result sequences without the need for duplicate removal. QuickXScan is practical and highly efficient.

**Keywords:** XPath, streaming, transitivity, propagation.

## 1 Introduction

We describe QuickXScan, an industrial strength streaming XPath algorithm implemented for native XML support in DB2 for z/OS [6,30]. QuickXScan evaluates XPath expressions with predicates by one sequential scan of XML data with high efficiency. It supports the following five forward axes: child, attribute, descendant, self, and descendant-or-self, and the parent axis by transformation into forward axes [23]. It also supports aggregate functions and arbitrarily complex predicates. In addition, it can support skipping of subtrees for many queries with significant performance benefit. As QuickXScan does not rely on relational techniques [29,3,7,14,15], the algorithm is well suited for applications over the Internet that favor an efficient streaming algorithm [12].

Some of the main issues in streaming XPath evaluation include: (1) how to model and represent an XPath query with complex predicates and aggregates for efficient evaluation; (2) how to represent matches efficiently for transitive steps to avoid combinatorial explosion; (3) how to evaluate expressions (including the result sequence) using least amount of buffers without duplicates; (4) how to generate partial results early to reduce the buffer requirement and response time (i.e. streaming output).

In QuickXScan, based on XPath semantics [26,27], we apply principles similar to that of attribute grammars and syntax-directed translation [1] to the streaming XPath evaluation and optimization. First, we model an XPath

query with a query tree. Each query node is associated with variables and their evaluation rules for expressions. As a result, we are able to handle complex queries like the following:

```
(Q1) /a//b[./c="XML" and d/e > 300 or  
        if sum(./f/g)>10 then count(d)<5].
```

Existing tree pattern-based query models [7] need to represent relationships among query tree branches, such as AND or OR [18,19,20], and will have difficulties to support complex predicates and aggregates. Automata-based models either do not support predicates, such as those for filtering [2,8,10,16], or isolate each instantiation of the matches and are less efficient in handling transitive steps (i.e. “//”) against recursive documents [5,17,20,21,22,24]. In QuickXScan, a query tree model is used without assuming any relationships between branches.

Second, it is well known that the transitive steps involving the descendant axis are the expensive ones to evaluate, since an incoming node can match many existing matches for a recursive document. In structural join based algorithms, there is a nice solution of using compact stacks to represent a possibly combinatorial explosive number of matching path instantiations with linear complexity [7,9,18,19]. However, most known automata-based algorithms [20,24] do not exploit such transitivity in matches, but have to track each instantiation of automata or transducers independently, leading to exponential complexity [24]. QuickXScan extends the idea of compact stacks, called a *matching grid*, to implicitly represent many match combinations, similar to what is used in Ramanan’s recent algorithm [25].

Third, QuickXScan uses iterative rules to incrementally evaluate expressions for complex predicates and aggregates. It uses transitivity properties for the variables and propagates values among matches for expressions and generating results without duplicates. Ramanan’s algorithm [25] has some similar features for simple predicate evaluation.

In this paper, we do not address the issue of streaming output, which was one of the focuses of XSQ [24].

The rest of the paper is organized as follows. In Section 2, we present a simple model for XML stream data. In Section 3, we describe XPath query representation and how to compile an XPath query into this representation. In Section 4, we present the representation of matches and matching process. Section 5 covers value propagation for expression

evaluation. Section 6 outlines the runtime algorithm. In the following section, we analyze some properties of the matching grid, the key representation of matches used in QuickXScan, and space and time complexity of the algorithm. We conclude the paper in Section 8.

## 2 XML document streams

To ease the exhibition of the algorithm, we focus on simplified XML documents with elements and texts only. An XML stream can be modeled by an event stream generated by visiting nodes of the XML data tree in a depth-first order.

An XML stream supports three primitives: (1) *read()*, which returns one of the two event types: *OPEN*, an event when encountering a start tag (visiting a node); and *CLOSE*, an event when encountering an end tag (leaving a node); (2) *skip()*, which skips the children and descendants, and continues traversal on the right sibling; and (3) *EOF()*, which tests the end of a document. Skipping provides significant savings even for SAX parsers as the construction of SAX events can be skipped.

Associated with each *OPEN* or *CLOSE* event is a node being positioned with the following information:

*node* (*nid*, *kind*, *name*, *level*, *value*),

where *nid* is a node identity used for document order; *kind* is simplified to *element* only, as document is handled at the initialization time only and *text* becomes the value of an element node; *name* is a QName for an element; *level* starts with 0 for the root and increments by 1 for each level of nesting; and *value* is a string value extracted from the node content.

## 3 Modeling and compiling XPath queries

The subset of XPath queries of interest is path expressions satisfying the following EBNF:

```

path ::= / rpath | // rpath
rpath ::= step | rpath / step | rpath // step
step ::= ( . | nametest ) pred*
nametest ::= QName
pred ::= '[' expr ']'
expr ::= expr and expr | expr or expr | not(expr)
        | prim | prim op prim | if expr then expr [else expr]
prim ::= rpath | nLiteral | sLiteral | string() | number() |
        count(rpath) | sum(rpath)
op ::= < | <= | = | != | > | >=

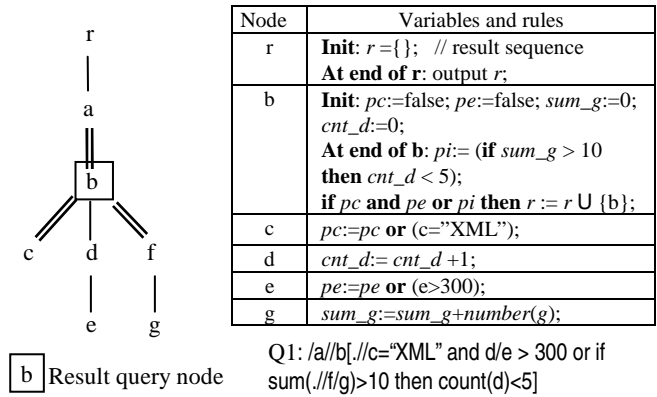
```

An XPath query, *Q*, is an absolute path expression starting with a root step, followed by steps, each with either a child or descendant axis (after the equivalence transformation of “//”), a name test, and an optional predicate list.

A path expression is compiled into a query tree. Each query node corresponds to a step and is labeled with a QName for the name test. Each edge represents a child or descendant relationship from one step to the previous in correspondence to the axis of the step. Graphically we use a

single line to represent the child axis and a double line the descendant axis. The *result query node* for a path expression is squared in the graphical representation. A *candidate result sequence* is the sequence of nodes that matches the result query node but for which not all predicates have been evaluated to true.

Each query node contains variables and evaluation rules needed for the path expression and other intermediate expressions, including the predicates associated with the step. Note that relative path expressions in predicates are merged into the query tree, and in predicates, they are replaced with references to relevant variables. For example, Query Q1 can be represented by a query tree shown in Figure 1. Also shown in Figure 1 are the variables and the rules to evaluate the expressions (omitting the propagation rules for the time being).



**Figure 1 Example of a query tree and associated rules**

In the example, variables with prefix “*p*” are for predicates; “*cnt<sub>d</sub>*” and “*sum<sub>g</sub>*” are for the two aggregates on *d* and *g*, respectively. Note that all the variables except *r* are scoped within a *b* instance. For example, when a *b* is matched, *sum<sub>g</sub>* is initialized to 0. Then when a *g* match is encountered, *sum<sub>g</sub>* is accumulated with a new *g* value. The final *sum<sub>g</sub>* value is used to evaluate the predicate  $sum(./f/g) > 10$  for the associated *b* instance. Also note that the rules for nodes *a* and *f* are trivial and optimized away.

Compilation of an XPath query can be performed stepwise by converting axis steps into query nodes and connecting them together, and converting functions, predicates, and candidate result sequences into variables and evaluation rules, which are small iterative programs. In general, we use the following framework to evaluate expressions for a match from the values of child or descendant matches:

1. *Init*: on a match, instantiate and initialize the variables;
2. *Extract*: for each matched child or descendant node, apply an extract function or evaluate expressions;
3. *Accumulate*: evaluate the new cumulative values based on the current cumulative values and new extracted values;

4. *Final*: at the end, perform final evaluation of variables, each of which may be based on a set of variables at the associated node.

In addition to predicate pushdown and incremental evaluation of aggregates, some other optimization can be performed. For example, predicate  $a < b$  is equivalent to  $\min(a) < \max(b)$ , and we only need to keep two scalar values for  $\min(a)$  and  $\max(b)$ , instead of two lists of objects.

It is worth mentioning again that any complex predicates can be modeled easily as the relationships among query tree branches are captured in variables and their rules.

## 4 Matching grid and matching process

The main data structure used at runtime is the query tree and the corresponding *matching grid*, a set of interrelated stacks containing *matching units* for matches. Each stack corresponds to one query node, and keeps track of matches to the query node. Matching units cross stacks are linked based on their matching relationship. As a document tree is traversed, matches for each query node are pushed onto the corresponding stack as part of the *OPEN* processing, and popped off when the *CLOSE* processing is finished. Multiple matches in a stack are graphically represented horizontally. Note that at any point, all the matches in the stacks have the ancestor-descendant relationship.

Figure 2 shows an example of a matching grid for a somewhat extreme case, which is for query `//a//a//b//c//d` and document `<a><a><a><a><b><a><b><c><b><c><d><d/></d></c></b></c></b></a></b></a></a></a></a>`. The superscript on each match symbol is the number of paths from the root to the match node. For example, there are 20 paths for  $d_{11}$ . However, all these paths contribute a single node to the result sequence.

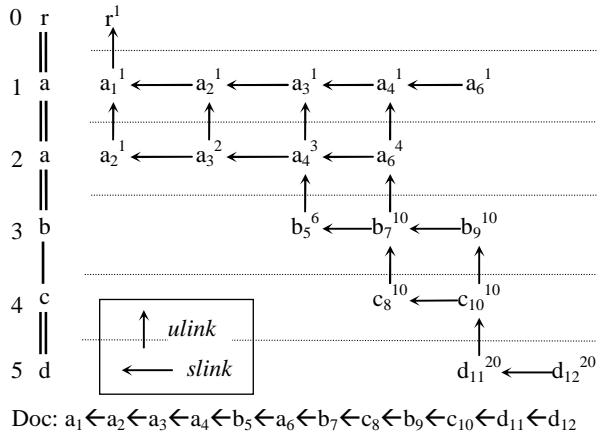


Figure 2 Example of a matching grid

The reason that we can use this compact representation is due to transitivity of the ancestor-descendant relationship. Intuitively, let a path expression be `//A//B`, and let  $a1$  and  $a2$  match with  $A$ ,  $b1$  match with  $B$ , and  $a2$  be a descendant of  $a1$ . Then, if  $b1$  is a descendant of  $a2$ ,  $b1$  is also a descendant of  $a1$ .

Except for the root step that can only match with the document root node, the basic matching conditions<sup>1</sup> between a query node  $q$  and a data node  $n$  are as follows:

1. The stack for the previous step of  $q$  is not empty; say  $n_1$  is at the stack top;
2.  $n$  satisfies the name test of  $q$ ;
3.  $n.level = n_1.level + 1$  if  $q.axis = child$ , or  $n.level \geq n_1.level + 1$  if  $q.axis = descendant$ .

Note that we only need to check conditions with the match at the stack top of the previous step. When a match is found, the relationship between  $n_1$  and  $n$  is kept in  $n$  with an upward link, *ulink*, to node  $n_1$ . Since we will use *ulink* for upward propagation of values, for document order, we will set *ulink* to *nil* if this match and the previous match of the same step share the same  $n_1$  (see  $b_9$  in Fig. 2). The matches of a step have a sideways link, *slink*, to form a logical stack.

A query node is *active* if it can potentially have a new match. The simplest way to see whether or not a query node is active is to check stack emptiness of its previous step.

To precisely track active query nodes without checking all the stacks, we need to analyze the query tree also. Once a query node  $q$  is matched, all its next steps become active. A step with the `"/` axis is called *m-transitive* as it can match with nodes at different levels. Once an m-transitive step becomes active, it will stay active until stack pop-off changes its status. Note that a non m-transitive query node is active only if its previous step has a match with the largest level.

The active query nodes can be easily maintained for push and pop by keeping two separate sub-lists for active query nodes, one for m-transitive steps and the other for non m-transitive steps. Details are omitted here. A document subtree can be skipped if there is no m-transitive step active and the current node does not match a query node.

## 5 Value propagation

The rules described in Section 3 are about accumulation of individual values. There is one more dimension to the rules, i.e. propagation and further accumulation of the cumulative values for an expression. Intuitively, if we have XPath steps such as `//A//B`, if  $a1$  is an ancestor of  $a2$  and both match with  $A$ , and  $b1$  matches  $B$  and is a descendant of  $a2$ , then  $b1$  is also a descendant of  $a1$ . Thus, if  $s1$  is a sequence of  $B$  descendants of  $a2$ , then  $s1$  is also a sub-sequence of  $B$  descendants of  $a1$ . This means the sequence of  $B$  descendants of  $a2$  can be propagated to  $a1$  to calculate the sequence of  $B$  descendants of  $a1$ . During the accumulation of  $B$  descendants of  $a2$ , there is no need to repeat the same for  $a1$  until  $a2$  is finished and its value is propagated to  $a1$ . This illustrates another key transitivity

<sup>1</sup> The conditions will need to be adjusted if the two steps have the same name test or the current step contains the self axis.

property that provides foundation for efficient incremental evaluation of expressions. We call this *p-transitivity* of a step for value propagation.

There are three basic types of values: Boolean values for predicates, scalar values for aggregates, and sequence values for path expressions. Let us continue with example Q1 for propagation rules on query node *b* (Figure 3). If we have two matches *b1* and *b2* on the stack of query node *b*, the *slink* of *b2* points to *b1*. Then at the end of *b2*, we can propagate two variable values, *pc* and *sum\_g* from *b2* to *b1*. The reason is that the step for *b* is *p-transitive* with respect to *pc* and *sum\_g*, but not *pe* and *cnt\_d*. The key to *p-transitivity* is in the next step. The *m-transitivity* of the next step makes the step *p-transitive* with respect to the corresponding variable from the next step.

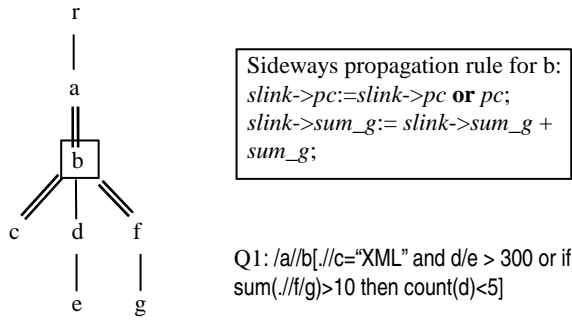


Figure 3 Example of a query tree and propagation rules

### 5.1 Propagating candidate result sequence

A candidate result sequence (*crs*) is a sequence of nodes matching a result query node with pending predicate filters from a node to the root. A *crs* that finds a path to the root with all the predicates being true becomes part of the result sequence. To get checked against predicates along the paths to the root, we use propagation. We differentiate among *upward*, *sideways*, and *backtracking* propagations. *Upward propagation* is to pass values from a match of a step to an upper step. When a *crs* reaches a certain step, any predicates below that step have been evaluated to true, and it only needs to get through predicates along a path to the root. Therefore we use upward propagation for a *crs* when a predicate is true. In case a predicate is false, some alternative paths have to be tried. One of the alternatives is to use sideways propagation.

*Sideways propagation* is to pass values between two matches of the same step, as the above example shows. While upward propagation is always valid, sideways propagation requires certain conditions. That is, a step has to be *p-transitive* with respect to a *crs*. By induction, we can prove that an *m-transitive* leaf query node is *p-transitive* for a *crs*, and a non-leaf step is *p-transitive* for a *crs* if its next step is *m-transitive*. Now if the predicate associated with a match is false, and the step is *p-transitive*, we can propagate the current *crs* sideways to its ancestor in the same stack for the query node.

What if the predicate is false and the step is not *p-transitive*? We could propagate a *crs* both upward and sideways as long as it is legal before reaching such a point. However, this will lead to duplicates when there are multiple paths with predicates being true. The reason we use upward propagation only when a predicate is true is to avoid such duplicates. This may cause us to miss the sideways propagation opportunities at the steps below the current step. To make up for this, we use *backtracking propagation*. It is to propagate a *crs* to the first *p-transitive* step that is below the current step. If there is no such step or the stack for such step is empty, then the *crs* is dropped.

We show an example in Figure 4, where the path expression is *//a[u]/b[v]/c[w]//d* (where *u*, *v*, and *w* are predicates), and a matching grid example is shown with the predicate truth values marked as superscript on each matching unit. The *crs*  $\{d_1, d_2\}$  is propagated to  $c_3$  first. As the predicate is true, it is propagated upward to  $b_3$ . Because the predicate for  $b_3$  is false and step *b* is not *p-transitive*, the *crs* will feed to  $c_2$ , as the dotted line shows. The reason that  $c_2$  remains available when *crs*  $\{d_1, d_2\}$  needs to be backtracked is that there is a parent-child relationship that guarantees  $c_2$  is an ancestor of  $b_3$  in this example. The *crs*  $\{d_1, d_2\}$  will eventually reach the root and become part of the result. The existence of  $c_2$  with a predicate being true is critical.

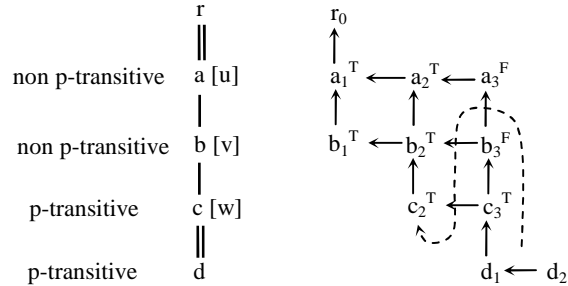


Figure 4 Example of propagation with predicates

Under these propagation rules, simple concatenation or merge of subsequences for accumulation can guarantee the uniqueness of a *crs* when it is accumulated from sub *crs*'s, and also the document order under certain conditions.

### 5.2 Propagating predicate and aggregate values

A primitive predicate truth value or aggregate value is derived from the *crs* of a relative path expression when it reaches the top node of the relative path, thus inherits the *p-transitivity* from the *crs*. If the step with the predicate or aggregate is *p-transitive*, then a truth value or aggregate value can be propagated sideways to its ancestor, as the example in Figure 3 shows (step *b* is *p-transitive* for *pc* and *sum\_g*). However, if there is predicate nesting, such as *a[b[c=5]/d="x"]*, a truth value or aggregate value will also need to be checked by other predicates and subject to the same propagation rules outlined above, except for some simplification. Since logical operators *and* and *or* are idempotent, we can get rid of duplicate concerns by always propagating them both ways when they are legal without

using backtracking propagation. Other simple aggregates, such as *min* and *max*, share the same simplicity.

However, *sum* and *count* need to take care of duplicates, although document order is insignificant. They can be handled similarly to the *crs* propagation, although the propagated value is a number instead of a sequence.

## 6 Putting it all together

We outline QuickXScan runtime algorithm in this section. In the pseudo code below, we only focus on the main features of the algorithm.

### 6.1 Data structures

A query tree  $Q$  can be represented as an array:  $qNode[qLen]$ , where  $qLen = |Q|$ , the query size, i.e. the total number of nodes in  $Q$ . Each  $qNode$  contains description for variables and rules needed for a match.

A match with a  $qNode$  is represented using  $mUnit$ . It contains instantiation of variables for the query node. Among them are  $qid$ ,  $nid$ ,  $level$ ,  $pred$ ,  $output$ ,  $ulink$ , and  $slink$ , where  $qid$  is the index of matching  $qNode$ ,  $nid$  is the node id,  $level$  is the level of the node,  $pred$  and  $output$  are variables for the match as the result of evaluating the definitions in  $qNode[qid]$ , and  $ulink$  and  $slink$  are the upward link and sideways link.

A *grid* consists of an array  $mUnit[]$ ,  $gTop$  and  $sTop[qLen]$ . Array  $mUnit[]$  is for all the matches and operates as a global stack with  $gTop$  as its stack top. Array  $sTop[qLen]$  keeps stack tops for logical stacks associated with  $qNode[qLen]$ . The entries in each logical stack are linked by  $slink$  in each  $mUnit$ . Note that push and pop of the global stack are consistent with that of the logical stacks.

### 6.2 The main procedure

The main procedure, as shown in Figure 5, traverses the document using *read()* and *skip()*. The *init()* also handles the root node, so the main loop focuses on two event types: *OPEN* and *CLOSE*. It keeps traversing a tree in document order and calling *Open()* for each new node unless it was told to skip a subtree. At the end of each node, it calls *Close()*. At the end of document, the result sequence is produced from the root matching unit.

```

Procedure QuickXScan(Document  $d$ , Sequence  $r$ )
1: begin
2:   $init()$ ;  $n := d.read()$ ;
3:  while ( $!d.EOF()$ )
4:     $skip := false$ ;
5:    switch ( $n.EventType$ )
6:      case OPEN:  $skip := Open(n)$ ;
7:      case CLOSE:  $skip := Close(n)$ ;
8:      if ( $skip$ ) {  $d.skip()$ ; continue }
9:       $n := d.read()$ ;
10: end // while
11:  $r := mUnit[0].output$ ; // result at the root
12: end;

```

Figure 5 The QuickXScan main procedure

### 6.3 Matching a node

Procedure *Open( $n$ )* as shown in Figure 6 is called for each node  $n$ . It loops through each active query node to check for matching (lines 3-10). If the match condition holds, a new  $mUnit$  is created and put into the matching grid. Otherwise, if  $n$  does not match any query node, and there is no  $m$ -transitive query node active, then its subtree is skipped. Otherwise, *activeQnodes* needs to be maintained.

```

// Match a node and return true if to skip the subtree
Procedure Open(node  $n$ )
1: begin
2:   $matched := false$ ;
3:  for each  $qid$  in  $activeQnodes$ 
4:     $q := qNode[qid]$ ;
5:    if (match condition holds) {
6:       $m := new MUnit(q, n)$ ; // initialize variables also
7:      Add  $m$  into  $mUnit[]$  and update  $gTop$  and
         $sTop[qid]$ , and set  $ulink$  and  $slink$ ;
8:       $matched := true$ ;
9:    }
10: end // for
11: if (not  $matched$  and no  $m$ -transitive  $qNode$  active)
12:  return true;
13: else { maintain  $activeQnodes$ ; return false }
14: end;

```

Figure 6 Procedure Open for matching a node

```

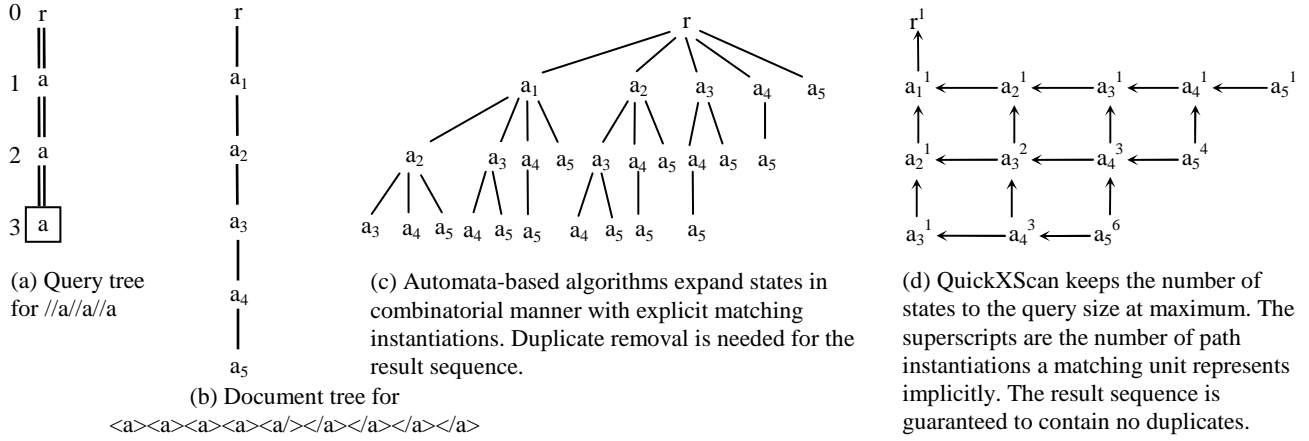
// Clear matches from matching grid whose level  $\geq n.level$ 
// and perform end-of-node processing
Procedure Close(node  $n$ )
1: begin
2:  while ( $gTop \geq 0$ )
3:     $m := mUnit[gTop]$ ;
4:    if ( $m.level < n.level$ ) return false;
5:     $q = qNode[m.qid]$ ;
6:     $mu := m.ulink$ ;  $ms := m.slink$ ;
7:    evaluate  $m.pred$ ; // true by default
8:    propagate values in  $output$  based on  $m.pred$  truth
      value,  $mu$  and  $ms$ , and query node  $q$ 's properties;
9:    for each propagated value, evaluate its cumulative
      value in destination matching unit;
10:   handle error and finish accordingly;
11:   maintain  $activeQnodes$ ;
12:    $gTop--$ ;
13:   if (early finish and no  $m$ -transitive  $qNode$  active)
      return true;
14: end; // while
15: end;

```

Figure 7 Procedure Close for processing end of node

### 6.4 Closing matching units

Procedure *Close( $n$ )* illustrated in Figure 7 is used to clear  $mUnits$  that the traversal has passed, i.e. those with level  $\geq n.level$ . It starts from the top of  $mUnit[]$ . The major work before an  $mUnit$  can be popped off is to execute its rules, and propagate values depending on the properties of the corresponding query node. As values are propagated to the destination  $mUnit$ , the cumulative values are evaluated.



**Figure 8 A worst-case scenario comparison of QuickXScan and automata-based algorithms**

As  $mUnits$  are popped off,  $activeQnodes$  needs to be maintained. Note that this procedure can handle a *CLOSE* event that corresponds to multiple consecutive *CLOSE* events.

We have simplified the algorithm description by ignoring some details for error and early finish. Early finish means that the predicate associated with a node is already true and there is no need to perform further matching.

## 7 Analysis

In this section, we present some space and time complexity properties of QuickXScan.

If the query nodes are viewed as states in an automata-based algorithm, we can compare the number of states of QuickXScan with automata-based algorithms. To understand the fundamental difference of our approach from existing finite automata algorithms, we use an example of the worst case scenarios to illustrate the point. Figure 8(a) shows a query tree for XPath query //a//a/a. Figure 8(b) shows a document tree for a simple 5 nested-elements document. Figure 8(c) shows all possible matches between the XPath query and the document, which are explicitly tracked by automata-based algorithms. It can be shown that

the total of match paths is  $\sum_{i=1}^k C_n^i$  where  $k$  is the number of

steps in a query and  $n$  is the document depth. In Figure 8(d), we show how QuickXScan keeps track of the matches using stacks. Only stack tops need to be checked for matching, thus reducing the number of states to the query size at maximum. Furthermore, since multiple matches to a query node are implicitly represented using a single matching unit, with the propagation rules, there are no duplicates for the result, while in automata-based algorithms, duplicate removal may be necessary. This scenario is the worst case because it has the maximum number of possible matches.

*Theorem 1.* At any time, the number of active states in QuickXScan is the number of query nodes in a query at maximum.

Now we analyze the upper bound of the number of matching units in a matching grid. Since all the nodes of the matching units in a matching grid at any time are from one path from the root to a current node being visited, the maximum number of distinct nodes are bounded by the height of the document tree. Since a data node can match with multiple query nodes, the maximum number of matching units is approximately  $(|Q| * h)$ , where  $|Q|$  is the number of query nodes in  $Q$  and  $h$  is the height of a document tree. In general, the number of matching units within a matching grid is  $O(|Q|^*r)$ , where  $r$  is the *recursion degree* of the document, or how many nodes with the same name are nested within each other at most.

*Theorem 2.* QuickXScan needs to maintain  $O(|Q|^*r)$  matching units at maximum at any time, where  $r$  is the recursion degree of a document.

Theorem 2 gives the estimates of memory requirement for matching units. In addition, memory is required for variables in each matching unit, especially for those sequence-valued variables. A matching unit may contain a variable for a node sequence of all descendants with a certain name, at certain time. Sequence propagation adds to the time complexity of QuickXScan. In the above worst case, the sequence contains only one node to start with. It will be propagated through all the matching units at maximum, therefore the time complexity is also  $O(|Q| * |D|)$ . In general, we have the following.

*Theorem 3.* The time complexity of QuickXScan is  $O(|Q|^*r * |D|)$ .

*Proof sketch.* Each node will match at most  $|Q|$  query nodes, and propagate through at most  $(|Q|^*r)$  matching units. We have at most  $|D|$  nodes to go through, therefore the time complexity is  $O(|Q|^*r * |D|)$ .  $\square$

The above estimate is loose and may be possibly improved. With minimization of XPath queries [11] as a phase in compilation, the complexity of QuickXScan is approaching the theoretical limit [4]. In practice, QuickXScan has linear time complexity of the document size.

We have implemented QuickXScan in a commercial database product with native XML support [6,30] and also in a Windows environment for the performance comparison purpose. Our performance measurement results using XMark [28] show that QuickXScan consistently outperforms systems such as TurboXPath [20], XSQ [24], and Galax [13] with respect to elapsed time, throughput, and memory consumption. Due to its skip feature, some queries can run faster than merely parsing a document and generating the SAX events.

## 8 Conclusions

We have presented QuickXScan, a practical and highly efficient streaming XPath evaluation algorithm. QuickXScan is based on the principles similar to that of attribute grammars. It fully exploits the transitivity of ancestor-descendant relationships in matching and evaluating variables for XPath expressions.

One of the unique features of QuickXScan is the query representation. It represents complex queries using a query tree, together with a set of variables and evaluation rules associated with each query node. Another feature of QuickXScan is at runtime, there is a set of interrelated stacks, one for each query node to keep XML data nodes that match with the query node. Active query nodes can be precisely tracked with maximum up to the query size. Yet another unique feature is the propagation of values and candidate result sequences without duplicates, which reduces significant bookkeeping overhead during the evaluation. Although it was implemented in a native XML database environment, QuickXScan is well-suited for other Internet applications.

## References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [2] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination. In *Proceedings of VLDB*, 2000.
- [3] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of ICDE*, 2002.
- [4] Z. Bar-Yossef, M. Fontoura and V. Josifovski. On the memory requirements of XPath evaluation over XML streams, In *Proceedings of PODS*, 2004.
- [5] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura and V. Josifovski. An Algorithm for Streaming XPath Processing with Forward and Backward Axes. In *Proceedings of ICDE*, 2003.
- [6] K. Beyer et al. DB2 goes hybrid: Integrating native XML and XQuery with relational data and SQL, *IBM Systems Journal*, Vol. 45, No. 2, 2006.
- [7] N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of SIGMOD 2002*, pages 310-321, 2002.
- [8] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of ICDE*, 2002.
- [9] B. Choi, M. Mahoui, and D. Wood. On the optimality of holistic algorithms for twig queries. In *Proc. 14<sup>th</sup> DEXA*, pages 28-37, 2003.
- [10] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *Proceedings of ICDE*, 2002.
- [11] S. Flesca, F. Furfaro, and E. Masciari, "On the minimization of XPath queries," In *Proceedings of the 29th VLDB Conference*, Berlin, Germany, 2003
- [12] D. Florescu, C. Hillary, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. Carey, A. Sundararajan, G. Agrawal. The BEA/XQRL Streaming XQuery Processor. In *Proceedings of VLDB*, 2003.
- [13] Galax. <http://www.galaxquery.org/research.html>.
- [14] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proceedings of VLDB*, 2002.
- [15] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: improving time and space efficiency. In *Proceedings of ICDE'03*, Bangalore, India, Mar. 2003.
- [16] T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proceedings of ICDT*, 2003.
- [17] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *Proceedings of SIGMOD*, 2003.
- [18] H. Jiang, W. Huang, H. Lu, and J. X. Yu. Holistic Twig Joins on Indexed XML Documents. In *Proceedings of the 29th VLDB Conference*, Berlin, Germany, 2003.
- [19] H. Jiang, H. Lu, and W. Wang. Efficient Processing of XML Twig Queries with OR-Predicates, In *Proceedings of SIGMOD 2004*.
- [20] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *The VLDB Journal*, 2005.
- [21] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of VLDB*, 2002.
- [22] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against XML streams. In *Proceedings of ICDE*, 2003.
- [23] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Workshop on XML-Based Data Management*, 2002. Springer LNCS 2490.
- [24] F. Peng, and S. Chawathe. XSQ: A streaming XPath Engine. *ACM Transactions on Database Systems*, Vol. 30, No. 2, June 2005, Pages 577–623.
- [25] P. Ramanan. Evaluating an XPath Query on a Streaming XML Document. COMAD 2005b, India, 2005.
- [26] P. Wadler. A formal semantics of patterns in XML, *Markup Technologies* 99, <http://homepages.inf.ed.ac.uk/wadler/papers/xsl-semantics/xsl-semantics.pdf>
- [27] P. Wadler. Two Semantics for XPath, <http://homepages.inf.ed.ac.uk/wadler/papers/xpath-semantics/xpath-semantics.pdf>
- [28] XMARK. <http://monetdb.cwi.nl/xml/index.html>
- [29] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD*, pages 425-436, 2001.
- [30] G. Zhang. Building a scalable native XML database on infrastructure for relational databases. *International Workshop XIMEP'05*. Available at <http://www.geocities.com/zhanggene/pub/ScalableNativeXMLDB.pdf>