XPath Queries on Streaming Data[*]

Feng Peng and Sudarshan S. Chawathe
Department of Computer Science, University of Maryland, College Park
College Park, Maryland, 20742, USA
{pengfeng,chaw}@cs.umd.edu

ABSTRACT

We present the design and implementation of the XSQ system for querying streaming XML data using XPath 1.0. Using a clean design based on a hierarchical arrangement of pushdown transducers augmented with buffers, XSQ supports features such as multiple predicates, closures, and aggregation. XSQ not only provides high throughput, but is also memory ef£cient: It buffers only data that must be buffered by any streaming XPath processor. We also present an empirical study of the performance characteristics of XPath features, as embodied by XSQ and several other systems.

1. INTRODUCTION

XML is becoming the de facto standard for information exchange and the amount of XML data is growing rapidly. Some of the data is accessible only in *streaming* form. That is, data items are presented in a £xed serialization; the application cannot seek forward or backward in the data, nor can it revisit a data item encountered earlier unless it is explicitly buffered. In addition to data that occurs natively in streaming form (e.g., stock market updates, real-time news feeds, network statistics), it is useful to process large XML datasets in streaming form because of the greater ef£ciency of streaming systems (which use a sequential scan instead of non-sequential data access on disk). In the sequel, we use the term **streaming XML** to refer to both data that occurs naturally in streaming form and data that is best accessed in streaming form.

We address the problem of evaluating XPath queries over streaming XML [23]. XPath is a well-accepted language for addressing parts of an XML document. It is often used in a host language such as XQuery and XSLT. However, it also serves a stand-alone query language for XML. Methods for ef£cient evaluation of XPath queries bene£t not only XPath query engines, but also systems for more powerful languages (e.g., XQuery) which incorporate XPath.

An **XPath query** consists a **location path** and an **output expression**. The location path is a sequence of **location steps** that specify the path from the document root to a desired element. The output expression speci£es the portions or functions of a matching element that form the results. Each location step has an **axis**, a **node test**, and an optional **predicate**. For example, the location path of the query `//book[year>2000]/name/text()` is `//book[year>2000]/name`. The location path matches the elements reachable from the document root using a path consisting of zero or more elements with arbitrary labels, followed by a book element, in turn followed by a name element. The output expression, `text()`, indicates that only the text content of the matching name appears in the result. In the £rst location step, `//book[year>2000]`, `//` is the **closure axis** denoting descendant-or-self, `book` is the **node test**, and `year>2000` is the **predicate**. The predicate restricts the results to the name subelements of books that have a year subelement whose content has a value greater than 2000.

Automaton-based methods for processing streaming data are attractive due to their ef£ciency and clean design. A challenging task in building automaton-based systems for XPath queries is the generation of the automaton from the query. The dif£culties (explained further by the examples below) are due to XPath features such as closures and predicates in conjunction with the read-once nature the streaming data. Brie¤y, when the automaton encounters an item in the stream, the data required to determine whether this item is in the query result may be unavailable. The unavoidable buffering introduces complexities of buffer management (¤agging buffered data based on subsequent satisfaction or falsi£cation of predicates, duplicate avoidance, etc.).

Much of the previous work [1, 12, 7] using this paradigm focuses on *£ltering* a collection of XML documents using restricted XPath expressions. Since XPath expressions without predicates are essentially regular expressions, they can be transformed into £nite state automata (FSA) that accept exactly the documents that satisfy the expressions. If the FSA accepts the document, the £ltering system returns the identi£er of the current document to the user. Thus, such systems do not need to buffer individual elements of the documents. However, as we shall explain shortly, general XPath queries cannot be evaluated in a streaming system that lacks buffering capabilities. The XMLTK system [2] is a closer match to our work, because it supports XPath expressions that retrieve only parts of a document. However, XMLTK does not support predicates in XPath expressions. Therefore, whenever it encounters an element that matches the path expression in a query, it can write it to output. In contrast, if the query includes predicates, the membership of an element in the query result cannot be decided immediately in general. The XSM system [19] handles predicates in the query but it does not handle the closures and aggregations. (It assumes that the query does not contain the closure axis //). As we describe below, closures pose signi£cant challenges to query evaluation.

We note that XPath features such as (multiple) predicates, clo-

[*]This material is based upon work supported by the National Science Foundation under grants IIS-9984296 (CAREER) and IIS-0081860 (ITR).

```
<root>
<pub>
   <book id="1">
      <price> 12.00 </price>
      <name> First </name>
      <author> A </author>
      <price type="discount"> 10.00 </price>
   </book>
   <book id="2">
      <price> 14.00 </price>
      <name> Second </name>
      <author> A </author>
      <author> B </author>
      <price type="discount"> 12.00 </price>
   </book>
   <year> 2002 </year>
</pub>
</root>
```

Figure 1: Example 1

```
<root>
<pub>
   <book>
      <name> X </name>
      <author> A </author>
   </book>
   <book>
      <name> Y </name>
      <pub>
         <book>
            <name> Z </name>
            <author> B </author>
         </book>
         <year> 1999 </year>
      </pub>
   </book>
   <year> 2002 </year>
</pub>
</root>
```

Figure 2: Example 2

$$
\begin{array}{rcl}
\mathbf{Q} & ::= & \mathbf{N}^{+}\Big[/\mathbf{O}\Big] \\
\mathbf{N} & ::= & \Big[/\Big|//\Big]\,\mathbf{tag}\,\Big[\mathbf{F}\Big] \\
\mathbf{F} & ::= & [\,\mathbf{FO}\Big[\mathbf{OP\,constant}\Big]\,] \\
\mathbf{FO} & ::= & \mathbf{@attribute}\Big|\mathbf{tag}\Big[\mathbf{@attribute}\Big]\Big|\,text() \\
\mathbf{O} & ::= & \mathbf{@attribute}\Big|\,text()\Big|count()\Big|sum() \\
\mathbf{OP} & ::= & >\Big|\geq\Big|=\Big|<\Big|\geq\Big|\neq\Big|contains
\end{array}
$$

Figure 3: BCNF for a Subset of XPath

sures, and aggregations are important usability advantages, especially if the data is semistructured or has a structure unknown to the query formulator. It is dif£cult to write a useful query on data whose structure is (partly) unknown without using closure. Similarly, predicates permit a more accurate delineation of the data of interest, leading to smaller, and more usable, results. The challenges posed by these features are exacerbated by data that has a recursive structure, as explained below. (A survey of 60 real datasets found 35 to be recursive [10].)

This paper makes the following **contributions**:

- To the best of our knowledge, our method for evaluating XPath queries over streaming data is the £rst one that handles closures, aggregations, and multiple predicates. As the examples below illustrate, these features, especially in conjunction, pose signi£cant implementation challenges.
- Our methods use a very clean design based on a hierarchical arrangement of pushdown transducers augmented with buffers. The system is easy to understand, implement, and expand to more complex queries.
- We present a detailed empirical study of XSQ and several related systems (Section 6). Our study illustrates the costs and bene£ts of different XPath features and implementation trade-offs as embodied by these systems.
- All the methods described in this paper are fully implemented in the XSQ system, which will be released under the GNU GPL license. In addition to serving as a testbed for further work on this topic, our system should be useful to anyone building systems for languages that include XPath (e.g., XQuery, XSLT).

The rest of this **paper is organized as follows**. In the rest of this section, we use examples to highlight some of the dif£culties in evaluating XPath queries over XML streams. Some preliminaries, including the SAX data model and the XPath language, are covered in Section 2. The design of a basic pushdown transducer (BPDT), which corresponds to an XPath location step, is presented in Section 3. Section 4 describes our method for composing BPDTs to generate the hierarchical pushdown automaton (HPDT) corresponding to an XPath query. Related work is summarized Section 5. Section 6 presents some results from our empirical study of XSQ and related systems. We conclude in Section 7.

EXAMPLE 1. *Consider the following query for the XML data in Figure 1: `/pub[year=2002]/book[price<11]/author`. When we encounter the £rst `author` element in the stream, we know that it satis£es the path `/pub/book/author`. However, the predicate in the £rst location step, `[year=2002]`, cannot be evaluated yet, since we have not encountered all the `year` subelements. We have encountered the £rst `price` subelement of the `book` element. However, we cannot determine whether the `book` fails the predicate `[price<11]`, since there may be more `price` subelements. Therefore we need to buffer the `book` element. When we encounter the second `price` element of the `book`, the second predicate evaluates to true. Since we still do not know the `year` of the `pub` element, the `author A` must continue to be buffered. When we encounter the two `author` subelements of the second `book`, we need to buffer the `authors A` and `B` as well. Now there are two `A`s and one `B` in the buffer. Next we encounter the second `price` element of the second `book`, and it does not satisfy the predicate. When we reach the end of the second `book` element, we know that the predicate `[price<11]` evaluates to false, since there are no more `price` subelements. Thus, the two `author` elements of the second `book` should be removed from the buffer. Note that one `author`, `A`, is still in the buffer since it belongs to the £rst `book`. Later, we determine that the `year` element of the `pub` element satis£es the £rst predicate. By noting that the `author A` in the buffer has already satis£ed the other predicate, we determine that the `author A` should be sent to the output immediately.*

As suggested by the example, we need to solve the following problems in order to evaluate even this relatively simple query. First, we may encounter data that is potentially in the result before we encounter the items required to evaluate the predicates to decide its membership. We need to buffer the potential result items. Second, items in the buffer have to be marked separately so that, after the evaluation of a predicate, we can process only the items that are affected by the predicate. Third, we have to encode the logic of the predicates in the automaton. In the above example, only when all the `price` children fail to satisfy the predicate (and we reach the end of the `book` element) does the `book` element fail to satisfy the predicate. In the mean time, if one of the children satis£es the predicate, we should know that the predicate is true and perform the operations accordingly. Finally, predicates access different portions of the data. Some should be evaluated when the begin tag is encountered, while others should be evaluated upon encountering the text content. There are other forms of predicates, which will be discussed in detail later.

Let us now consider a more complex example, using a query with closures, and data with recursive structure. Figure 2 suggests data with recursive structure: the `pub` element in line 2 has a grandchild named `pub` in line 9.

EXAMPLE 2. *Consider the following query for the XML data in Figure 2: `//pub[year=2002]//book[author]//name`. This example introduces some new problems, in addition to those discussed in the previous example. Since the closure axis // is used in the query, a node and its descendants may match the same location step at the same time. For instance, the `pub` elements in both line 1 and line 9 match the node test in the £rst location step. Consider the `name` element in line 11. There are three ways it can match the query, and each of the matches gives different results of the predicates:*

pub	*[year=2002]*	*book*	*[author]*	*name*
line 2	*true*	*line 7*	*false*	*line 11*
line 2	*true*	*line 10*	*true*	*line 11*
line 9	*false*	*line 10*	*true*	*line 11*

As indicated by the table, only the match in the second row results in both predicates evaluating to true. When we encounter the end tag of the `pub` element in line 15, we know that the `pub` element in line 9 fails the predicate `[year=2002]`. However, we cannot remove the `name` Z from the buffer since it is still possible that this item satis£es the query. The same situation occurs again when we encounter the end tag of the `book` element in line 16. Only when all the possible matches have evaluated the predicates to false can we remove the item from the buffer. We also have to be careful with the other cases where multiple matches evaluate all predicates to true. For example, if we add an `author` element between line 8 and line 9 for the `book` element in line 7, the match in the £rst row would also evaluate both predicates to true. In such cases, we have to avoid duplicates (outputting the same element twice) as well.

These examples illustrate the dif£culties encountered in designing an automaton for evaluating XPath queries systematically. Dif£culties arise due to the fact that elements in an XML stream may come in an order that does not match the order of the corresponding predicates in the query, and due to recursive structure in the data. When the query contains the closure axis and multiple predicates, it is even more dif£cult to keep track of all the information needed for proper buffer management.

2. PRELIMINARIES

2.1 Data Model for XML Streams

Parsers based on the SAX API process an XML document and generate a sequence of SAX events. For each opening (and closing) tag of an element, the SAX parser generates a *begin* (respectively, *end*) event. The begin event of an element comes with a list of (attribute name, attribute value) pairs with the attribute name as the key. For text contents enclosed by the opening and closing tag, the SAX parser generates a *text* event.

The streaming XML data is modeled as a sequence of SAX events, extended with the depth of the event. That is, an XML stream is a sequence $\{e_1, e_2, ...e_i, ...\}$ where $e_i \in B \cup T \cup E$:

- B = { (a,attrs,d) }. (a,attrs,d) is the begin event of an element with the tag "a" that is at depth d in the XML data and *attrs* is a list of the attribute name-value pairs.

- E = { (/a,d) }. (/a,d) is the end event of an element with tag "a" at depth d.

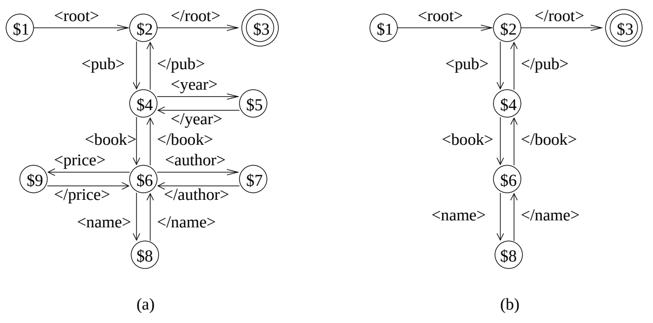


Figure 4: A simple PDA and a simple PDT for the XML stream in Figure 1

- T = { (a,text(),d) }. (a,text(),d) is the text event in the element with tag "a" at depth d. The content of the text event can be retrieved using text().

2.2 XPath

As noted earlier, XSQ implements all of XPath 1.0 [23] (including closures, aggregations, and multiple predicates) except reverse axes (such as `preceding-sibling`) and position functions (such as `pos()` and `last()`). For the rest of this paper, we will focus on the core subset of XPath described by the grammar shown in Figure 3. An XPath query is in the form of $N_1 N_2 \ldots N_n / O$, which consists of a location path, $N_1 N_2 \ldots N_n$, and an output expression O. An element matches the location path if the path from the document root to that element matches the sequence of labels in the location path, and satis£es all predicates (speci£ed syntactically using square brackets). For each matching element, the result of applying the output function to the element is added to the query result. The output expression can specify an attribute of the element, or its text value. It may also be an aggregation function (e.g., sum()) applied to the element's content. If no output expression is speci£ed in the query, the query returns all the elements in the result set.

3. BASIC PUSHDOWN TRANSDUCER

A pushdown transducer (PDT) is a pushdown automaton (PDA) with actions de£ned along with the transition arcs on the automaton. It has a £nite set of states which includes a start state and a set of £nal states, a set of input symbols, and a set of stack symbols. At each step, it fetches an input symbol from the input sequence. Based on the input symbol and the symbols in the stack, it changes the current state and operates the stack according to the transition function. Besides the state transition and stack operation, the transition function also de£nes an output operation which could generate some output during the transition. Note that traditional PDTs do not have an extra buffer and the operations for the buffer. However, as discussed in Section 1, evaluating XPath queries over XML streams requires buffering potential results.

3.1 A Simple PDA for XML Streams

First we introduce a PDA that accepts XML streams that have certain string. Figure 4(a) shows the state transition diagram of a PDA that accepts the XML stream in Figure 1. Text events that are not shown in the diagram map to self-transitions.

For each of the SAX events generated for the XML stream in Figure 1, the PDA in Figure 4(a) makes a state transition according to the state transition diagram. For each *begin* event, it also puts the tag of the element into the stack. For each *end* event, it will match the tag of the current element and the tag at the top of the stack. If these two tags match, it pops the tag from the stack. Otherwise the XML stream is not well-formed. After the PDA has processed all the events generated from the stream, the PDA should be in the £nal state *$3* and the stack should be empty. In the following discussion, we assume the XML stream is always well-formed.

Such a PDA can be adapted to a £ltering system for XML documents using the following method. Suppose we want to £nd all the documents that contain some elements that have the pattern `//pub//book//name`. We can just remove all the branches in Figure 4(a) to make it a £lter PDA shown Figure 4(b). Note that if the state transition of the next event is not de£ned in the £lter PDA, the £lter PDA just stays in the same state. Whenever the £lter PDA reaches state *$8*, we know that the current XML document contains an element that satis£es the £lter expression and we can return the document to the user. Moreover, if we put output functions in state *$8* such that it can output the content of *name* element, the £lter PDA becomes a PDT that can answer the XPath query.

However, it is not straightforward to extend this simple PDA to a PDT that answers XPath queries. The main reason is that the PDA has no memory for the previously processed data (the stack of the PDA is used exclusively to keep track of matching the begin and end tags). However, we need the results for all the predicates, which may come in any combinations of sequences, to determine state transitions and the actions. A direct solution is to remember the current results for every predicate, and mark every item with a ¤ag that indicates which predicates are satis£ed and which are not yet. Such methods signi£cantly degrade the performance. For instance, every time we evaluate a predicate, such a method would need to go through the whole buffer to check if some items are affected by its result. Further, the system becomes complex and uses ad-hoc methods to keep track of all the information needed. In Example 1, if the £rst `year` element has satis£ed the predicate `[year=2002]`, the other `year` element of the same `pub` element should not be tested anymore. Then we need to explicitly set a ¤ag for this predicate and reset it appropriately. Considering different semantics of the predicates, we need to set all these operations (set or reset, check or do not check, etc.) separately. If there are closures in the query and the data is recursive, the naive approach becomes even more complex.

3.2 Building the BPDT

Our solution to this problem is based on our observations on the following example.

EXAMPLE 3. *For the following XPath query, consider the second location step (`/book[author]`):*

Q: `/pub[year>2000]/book[author]/name/text()`

In a PDT for this query, we need to perform at least three tasks for this location step: 1. If the `book` element does have an `author` subelement, we need to remember the fact for future use.
2. If the `book` element does not have an `author` subelement, we need to make sure that if the `name` of the current `book` element has been in the buffer, it is deleted from the buffer.
3. If the `book` element does have an `author` subelement, we need to make sure that if the `name` of the current `book` has been in the buffer, it is sent to the output if all the predicates have evaluated to true. If some of the predicates have not been evaluated, we should hold the content in the buffer and handle it later.
The event upon which we can perform the £rst task is the the begin event of the `author` element. The event upon which we can perform the second task is the end event of the `book` element since until then we cannot be sure that the `book` element does not have an `author` subelement. At the begin event of the `author` element, we also need to perform the third task since we know now the predicate in the current location step is true.
Intuitively, these observations suggest associating a PDT similar to the one suggested by Figure 8 with a location step of this form. (The buffer operations will be explained in detail shortly.)

Using similar analysis to the above example for the predicates used in XPath , the location steps in any XPath queries can be categorized into the following classes based on the events upon which the predicates are evaluated.
1. Test whether the current element has a speci£ed attribute, or whether the attribute satis£es some condition, (e.g., `/book[@id]`, `/book[@id` $\leq$ `10]`).
2. Test whether the current element contains some text, or whether the text value satis£es some condition, (e.g., `/year[text() = 2000]`).
3. Test whether the current element has a speci£ed type of child, (e.g., `/book[author]`).
4. Test whether the the current element's speci£ed child contains an attribute, or whether the value of the attribute satis£es some condition, (e.g., `/pub[book@id` $\leq$ `10]`).
5. Test whether the speci£ed child of the current element has a value that satis£es some condition, (e.g., `/book[year` $\leq$ `2000]`).

Based on the above categorization, we design a template for each category of location steps. Figure 5 to Figure 9 summarize these templates. In each template, there is a START state, a TRUE state that indicates the predicate in this location step has evaluated to true, and an NA state that indicates the predicate has not yet been evaluated. The PDT generated from a location step using the template is called a basic pushdown automaton (BPDT). The BPDT has two important features:
1. The result of the predicate is encoded in the states. It is easy to show that whenever the BPDT is in the TRUE state, the predicate has evaluated to true; whenever the BPDT is in the NA state, the predicate has not yet been evaluated.
2. The logic of the predicate is encoded in the BPDT. For example, in Figure 9, we can see the exact logic we want for location steps such as `/pub[year=2002]` in Example 1. If one of the children satis£es the criterion, the BPDT will move to the TRUE state. Only if all the children fail the predicate does the BPDT return to the START state from the NA state, signifying that the predicate has evaluated to false.

3.3 Buffer operations in BPDT

In contrast to the simple PDA, each BPDT has a buffer of its own that is organized as a queue. The operations on the buffer are as follows:
1. *Q*.enqueue(*v*): add *v* to the end of the queue;
2. *Q*.clear(): remove all the items in the queue;
3. *Q*.¤ush(): send all items in the queue to the output in FIFO order;
4. *Q*.upload(): move all the items in the queue to the end of the queue of the BPDT that is the parent of this BPDT in the HPDT network, as explained further in Section 4.

Note that we do not have the *dequeue* operation for the queue since all the items in the queue will be operated on together: either to be *cleared* or to be *¤ushed* to output.

3.4 State transitions in BPDT

When we need to process closures and multiple predicates in the XPath queries, the BPDT is non-deterministic. (Recall Example 2 from Section 1.1.) At runtime, it has to keep a current state set *S*.

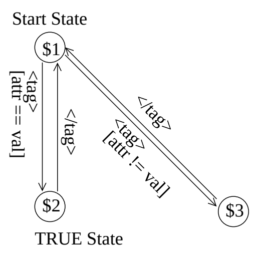


Figure 5: Template BPDT for: /tag[@attr=val]

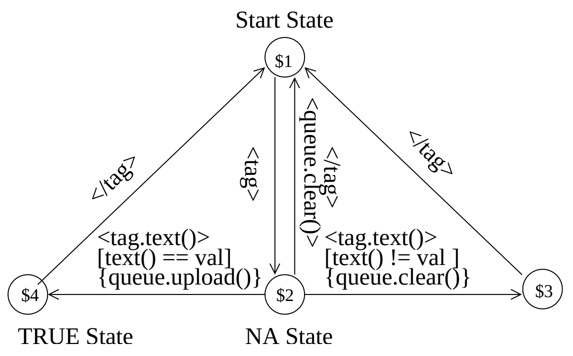


Figure 6: Template BPDT for: /tag[text()=val]

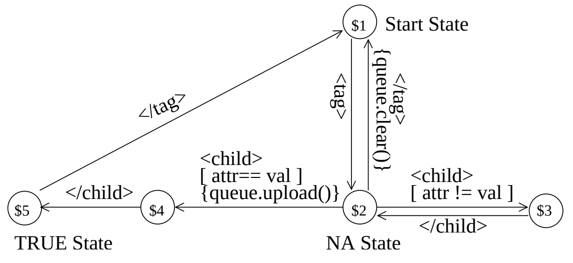


Figure 7: Template BPDT for: /tag[child@attr=val]

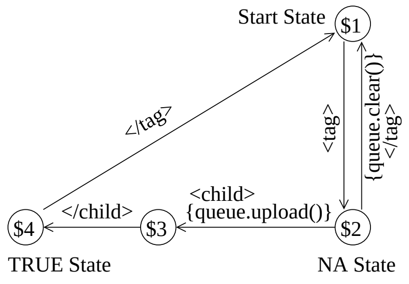


Figure 8: Template BPDT for: /tag[child]

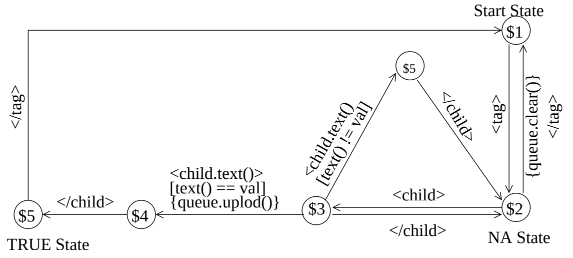


Figure 9: Template BPDT for: /tag[child=val]

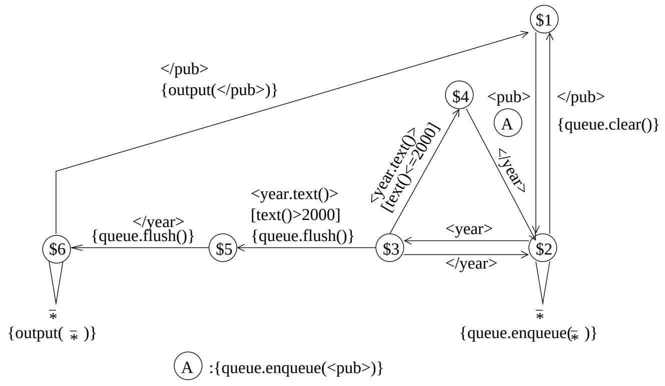


Figure 10: BPDT for query: `/pub[year>2000]`

At each step, the BPDT makes the transition based on each current state $s \in S$, the input symbol e that is a SAX event, and the predicate f. Note that the predicates, the operations, and the new states are stored with the transition arcs.

The BPDT £rst matches e with the labels on all the transition arcs. If it does not £nd a match, it ignores e. If it £nds a matched arc, it £rst checks the predicate f. If f is not *null*, the BPDT evaluates the f using e. If f evaluates to $false$, it does nothing. Otherwise, it replaces s with a new state s' determined by the transition arc. In addition to the state transition, it may also operate on the buffer and produce output.

If we do not need to process closures, the BPDT is deterministic. It always has a single current state. Moreover, there is at most one transition arc that matches the current event. Thus, after it £nds one match of the incoming event it can terminate the searching process immediately and process the next incoming event. Moreover, since we do not need to visit every transition arc, we can use a more ef£cient data structure to store the arcs to make the matching process more ef£cient.

In addition to the regular transitions used in PDTs, the BPDT can have the following special transitions denoted by these special labels: $//$, $*$, and $\bar{*}$. $//$ stands for *closure*. It will match any incoming *begin* event and labels a self-transition. $*$ stands for *wildcard*. $<*>$ will match any incoming *begin* event and $</*>$ will match any *end* event. $\bar{*}$ stands for a *catchall* symbol. It is used when there is no output expression speci£ed in the query. In this case, we need to output the whole element if it is in the result set. $\bar{*}$ will match any incoming event that corresponds to a descendant of the current element.

EXAMPLE 4. *The BPDT shown in Figure 10 uses the catchall symbol. It can answer the query with a single location step: `/pub [year>2000]`. We can see in Figure 10 that in state* $2 *and* $6, *there are two catchall transitions that are responsible to get all the descendants of the `pub` elements. The difference is that in state* $2, *we do not know whether the `pub` element is in the result or not, thus we have to put the descendants in the queue £rst. As soon as we know the predicate is true (when the BPDT performs the state transition* $3 → $5*), the items in the queue are ¤ushed to output. Only when all the `year` elements fail the predicate does the BPDT clear the queue. Note that there is an ¤ush operation on the transition arc from* $5 *to* $6, *which takes care of all the items that are enqueued between the text event and end event of the `year` element if there are any.*

4. HIERARCHICAL PDT

The BPDTs are combined into one hierarchical pushdown transducer (HPDT), in the form of a binary tree, to process XPath queries. The key idea is to use the position of the BPDT in the HPDT to encode the results of all predicates. The BPDT can determine whether a predicate has been evaluated or not by its own position, which is £xed and easy to get in a binary tree. Therefore, the buffer operations in the BPDTs can be determined accordingly. Due to space limitation, we only give a brief description of the algorithms in this section[1].

4.1 An example of HPDT

The state transition diagram shown in Figure 11 is generated for the query `//pub[year>2000]//book[author]//name/text()`. However, if we ignore the special transition arcs in the diagram, it can answer a simpler query without closures: `/pub[year>2000] /book[author]/name/text()`. When the *upload* function in the diagram is called, the contents in the buffer of the current BPDT is moved to the end of parent BPDT's buffer. We describe the process that the HPDT evaluates the query over the stream shown in Figure 1. Each box in the £gure denotes a BPDT. The number on the shoulder of the box is the name of the BPDT.

EXAMPLE 5. *The HPDT starts from state $1. It follows the rule as the usual PDT. When it encounters the `name` "£rst", it is in state $14, thus it enqueues the text content "£rst" into the buffer of $bpdt(3,4)$. At the end event of the `name` element, the item is uploaded to the buffer of $bpdt(2,2)$. The next event is begin event of the `author` element, thus the HPDT goes from state $8 to state $9 and uploads the item to the buffer of $bpdt(1,1)$. The same process applies to the item "second", which is the `name` element of the second `book`. Then at the begin event of the `year` element, the HPDT is in state $3 and the buffer of $bpdt(1,1)$ contains two items: "£rst" and "second". When the HPDT encounters the text event of the `year` element, it evaluates the predicate `[year.text()>2000]`. The result is true. Thus the HPDT goes from state $4 to $6 and ¤ush the content in its buffer to the output. Therefore, the HPDT returns the right result for the query. From this example, we see that in each BPDT, the buffer operations can be determined based on its position in the HPDT. For example, for $bpdt(3,4)$, we know it is the right child of $bpdt(2,2)$. This fact indicates it is connected to the NA state of $bpdt(2,2)$. Thus, when the HPDT reaches the $bpdt(3,4)$, the predicate in $bpdt(2,2)$ has not been evaluated yet. Similarly, since $bpdt(2,2)$ is the right child of $bpdt(1,1)$, we know that when the HPDT reaches $bpdt(2,2)$, the predicate in $bpdt(1,1)$ has not been evaluated yet. Combine these facts, when the HPDT is in $bpdt(3,4)$, we know that both predicates have not been evaluated yet. Notice that these information can be obtained solely from the positions of the BPDTs, it is easy to determine the buffer operations in the BPDTs systematically. The details are described below.*

4.2 Building HPDT from XPath Queries

We now describe how to build an HPDT from an XPath query. Since the BPDT decides its own buffer operation based on its position in the HPDT, we denote the position of each BPDT by a unique ID (l,k), where $l \geq 0$ is the depth for the BPDT in the HPDT system and $k \geq 0$ is its sequence number within the layer (right to left). Given an XPath query $N_1N_2...N_n/O$, the BPDTs, together with the IDs, are generated as described as follows. We £rst generate a **root BPDT** as in Figure 12. The root BPDT is used to consume the

[1]For details, please see our longer version of technical report at `www.cs.umd.edu/~pengfeng/xsq`

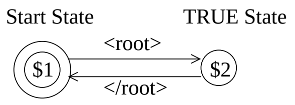


Figure 12: Template for the root BPDT

<root>, *</root>* events, which are generated by the SAX parser for the document root for every XML document. The root BPDT has an ID (0,0). For location step N_i, we go through all the BPDTs $bpdt(i-1,k)$, which are generated from N_{i-1} (N_0 could be thought as $/root$). For each existing $bpdt(i-1,k)$, if it has an NA state, we generate a $bpdt(i,2k)$ as its right child, which use the NA state of $bpdt(i-1,k)$ as its START state. If $bpdt(i-1,k)$ does not have an NA state, we set $bpdt(i,2k)$ to NULL. Similarly, we generate a $bpdt(i,2k+1)$ as the left child of of $bpdt(i-1,k)$, which uses the TRUE state of $bpdt(i-1,k)$ as its START state.

After we connect the BPDTs using the above method, the buffer operation in $bpdt(l,k)$ can be determined as follows. First there is the fact that if $k=(k_0k_2...k_n)_2$, when the HPDT reaches a state (not including the START state) in this BPDT, the i*th* predicate has evaluated to true if and only if $k_i = 1$. We can prove this fact by induction since the left child is connected to the TRUE state of the parent, which means that the predicate in the parent has evaluated to TRUE when the HPDT reaches the states in this BPDT. Therefore, the buffer operations of this BPDT can be determined given the results of the predicates. Note that in $bpdt(i,2^i-1)$, we know that all the predicates in higher layer BPDTs have evaluated to true. Thus, in every $bpdt(i,2^i-1)$ i=$1,...,n$, the BPDT sends the content in the buffer to the output if the predicate in itself evaluates to true.

After generating the new BPDT based on the templates, we also modify the resulting BPDT if the axis is a closure axis `//`. We add a self-transition marked with `//` on its START state. Then the transition arc for *begin* event that come out from the START state and reach a lower layer BPDT are marked with `=`. These arcs are called **closure transitions**. The usage of these two transitions will be described shortly.

We then add the output functions to the lowest layer BPDTs. In $bpdt(n,2^n-1)$, the value is sent to the output directly. In all the other BPDTs in layer n, the output will be sent to the buffer. If the output expression O is speci£ed, the corresponding attribute or function is added to the transitions in the lowest layer BPDTs. Otherwise, a catchall transition is added to the lowest layer BPDTs.

4.3 Running the HPDT

Since XSQ handles XPath queries with closures and multiple predicates, it needs additional mechanisms to ensure that the cases such as in Example 2 are handled correctly. As we show in Example 2, when the HPDT encounters the `name` element on line 11, there are three ways that the path to the element matches the query because of the closures in the query. Each of the matches evaluates the two predicates in the query differently. Although we can get the result of the predicates by the position of the current BPDT as described earlier, we need to solve the problem of multiple matches so that if one of the matches evaluates all the predicates to true, the HPDT keeps the element in the result. Example 6 depicts the scenario when the HPDT is processing the multiple matches.

EXAMPLE 6. *Consider the stream in Figure 2. We use the HPDT in Figure 11 to process it. When the HPDT encounters the `name` element on line 11, it is in state $14. However, there are three paths from $1 to $14: 1→2→7→11, 1→2→10→11, and 1→9→10→11. All the three paths lead into the same state since the predicates of*

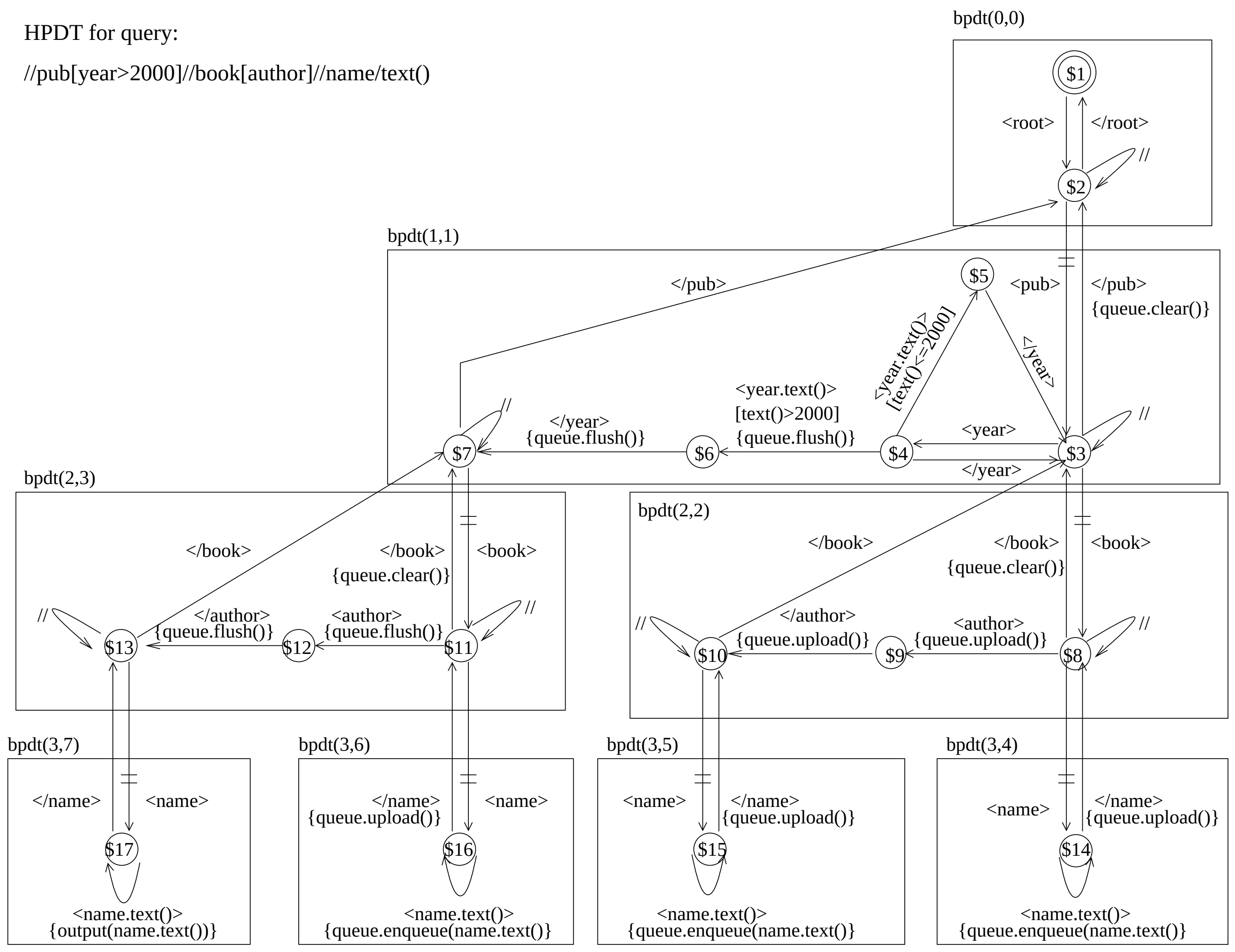


Figure 11: HPDT generated for query: `//pub[year>2000]//book[author]//name/text()`

all of them have not been evaluated. Since the current BPDT has the ID $(3,4)$, $4 = (100)_2$, we know that only the £rst predicate is true while the other two are unknown (the £rst predicate is in `/root` which is always true). However, we cannot simply enqueue the item Z at the text event of the current element. Otherwise, for the £rst path, the item will be cleared at the state transition from state \$8 to \$3 when the HPDT encounters the end of the `book` element on line 16 (which corresponds to the `book` on line 7). Since this `book` element does not have an `author` child, the predicate in the second location step evaluates to false. Similarly, for the third path, the HPDT will clear the item when it goes from state \$3 to state \$2, since the `year` child of the `pub` element on line 9 fails the predicate in the £rst location step. If HPDT follows the second path, it will output the item when it goes from state \$4 to state \$6 where it encounters the `year` element on line 17. Even if the elements are in different order, the HPDT in Figure 11 can always returns the correct result.

We use a **depth vector** to keep track of the path to each current state. At runtime, each current state s is associated with a depth vector dv. It records the depths of the events that trigger the state transitions that lead to the current state. The dv of every state is initialized as empty. Suppose e is the incoming event and s' is the new state. In the case that $e \in B$, if $s' \neq s$ (the state transition occurs), $s'.dv = s.dv + e.d$ ($e.d$ is **appended** to the end of dv), otherwise $s'.dv = s.dv$. In the case that $e \in E$, if $s' \neq s$, $s'.dv = s.dv - e.d$ ($e.d$ is **removed** from the end of dv), otherwise $s'.dv = s.dv$. Text events do not change the depth vectors. In addition to the append and remove operations, the operation **top** returns the last depth in the vector. The depth vector essentially simulates the stack operations for every possible path that the element matches the query.

When we enqueue an item, we associate the depth vector of the current state with the item as well. Thus, the same item may have more than one depth vector since it may be enqueued by different states. Accordingly, when we perform the other buffer operations, we also only operate the items with the depth vector that is equal to the depth vector of the current state. For example, in Example 6, when the HPDT goes from state \$3 to \$2 where it encounters the end of the `pub` element on line 15, it will clear the buffer. Note that the item of the correct match is also in the same buffer at the same time. However, since the the BPDT only operates the items in the buffer with the same depth vector as the current state, which is $(1,9)$, the item of the corrected match, whose depth vector is $(1,2)$, is not deleted.

Though the operations on the items in the buffer are not as a whole now, we organize the items so that items with the same depth vector are kept in a group, which will be operated together. Note that the operations on depth vector are implemented using bitmap vectors. All the operations and comparisons are done using integer and bit operations. It is quite ef£cient in the implementation.

It is possible that one item is enqueued or sent to output multiple times if more than one matches satisfy the query. The solution is as follows. Since we only operate on the reference of the item in the system, we mark the item as "output" as soon as one match satis£es the query. If the item marked as "output" is at the head of the queue, it is sent to the output immediately. Otherwise, it will remain unchanged no matter what the later operations are, until it becomes the £rst item in the queue. This operation is an important factor that affects the performance compared with the deterministic HPDT as we will show in the experimental results in Section 6. In a deterministic HPDT, the result items are always determined in document order. When we perform the $flush$ or $output$ function, we can directly write to output. Thus, we do not need to buffer these items and check the buffer later, which improves the throughput of the XSQ system.

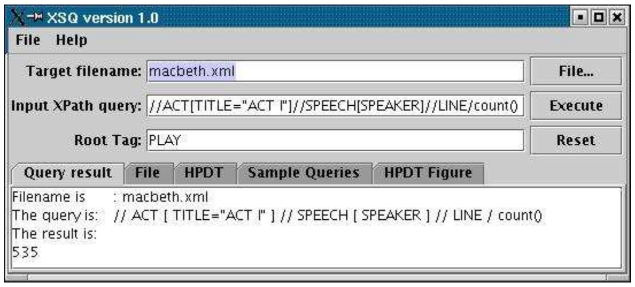
We also need additional rules for the state transitions due to the closures. Usually a transition arc that starts from state s accepts an event e if its depth $e.d$ satis£es the criterion: if $e \in B$, $e.d = s.dv.top() + 1$, otherwise $e.d = s.dv.top()$. However, for the special labels, we have the following different rules. Transition arcs labeled with $//$ accept any incoming *begin* event of *any* depth. The closure transitions that are marked with = signs, accept the speci£ed *begin* event at any depth. $\bar{*}$ matches *any* event e if $e.d > s.dv.top()$, which indicates that the element corresponding to the event is a descendant of the element that leads to the current state.

Now we can de£ne the $queue.upload()$ function as to move all items in the current BPDT to *the nearest ancestor that has the current BPDT in its right subtree*. We de£ne the upload function in this way such that it uploads the items in the buffer directly to the BPDT that is still in an NA state. For any ancestor of the current BPDT, if the current BPDT is in its left subtree, the predicate in this ancestor must have evaluated to $true$ (since it has reached the TRUE state). The de£nition ensures that the $flush$ function for the results are called before the $clear$ function.

EXAMPLE 7. *Consider the same example in Example 6. Suppose we put the $flush$ function on the transition arc from state \$7 to \$2 instead of the current one that is de£ned together with the end event of `year` element. What happens if a result `name` element comes after the text event of `year` element on line 17 but before its end event? If we do not have the $flush$ function on the arc from state \$6 to \$7. Then when the HPDT reaches state \$7, there will be two current states with the same depth vector $(1,2)$: \$3 (because of the self-transition on state \$3) and \$7. How can we guarantee that the result item will not be cleared by mistake? The de£nition of the upload function will ensure that the result item after the text event `year` element will be uploaded to $bpdt(1,1)$ before it performs the* ¤ush() *function from state \$6 to state \$7. For example, $bpdt(3,5)$ would upload its content to $bpdt(1,1)$ instead of $bpdt(2,2)$ because the predicate in $bpdt(2,2)$ has evaluated to true. Notice that this de£nition also prevents that the item will not be cleared by the $clear$ function from state \$8 to \$5. cleared by*

4.4 Aggregations

The XSQ system is augmented with a statistics buffer *stat* to handle aggregations. In the *stat* buffer, there is one item for each aggregation function with initial value to $null$. The operations for the *stat* buffer are: 1. `stat.update(aggr,value)`: update the item for aggregation function *aggr* in *stat* with the *value*. For example, `stat.update(COUNT,2)` will add 2 to the number in *stat*. 2. `stat.output(aggr)`: output the value in *stat*.



macbeth.xml

//ACT[TITLE="ACT I"]//SPEECH[SPEAKER]//LINE/count()

PLAY

```
Filename is      : macbeth.xml
The query is:    // ACT [ TITLE="ACT I" ] // SPEECH [ SPEAKER ] // LINE / count()
The result is:
535
```

Figure 13: Screenshot of the XSQ system

For example, consider the query:

`//pub[year>2000]//book[author]//name/count()`

The HPDT will keep the same except that we replace all `queue.flush()` with `stat.update(COUNT, number of items in the queue)`, `output(value)` with `stat.update(COUNT,1)`, and place `stat.out(COUNT)` on the transition arc from state \$2 to \$1 where the document ends. The resulting HPDT can answer this aggregation query. We also modify the semantics of `stat.update()` such that it emits a new value whenever the number in the buffer is updated. Thus we can always get the aggregation value for the data we have seen so far. This feature is useful when we process aggregation queries over unbounded streams.

5. RELATED WORK

Due to space constraints, we restrict our attention to the work that is most closely related to XSQ, and systems that are studied further in Section 6. For a more general discussion of stream processing, we refer the reader to a number of recent papers on the topic: For example, stream processing in the context of the *DSMS* system is discussed in [3]. Methods for dynamically grouping similar queries to increase system throughput in *NiagaraCQ* are discussed in [9]. Methods for validating streaming XML using pushdown automata are presented in [22]. Rewriting XPath queries with reverse axes into equivalent queries with only forward axes is studied in [21].

A k-pebble tree-walking tree-transducer model is de£ned for XML transformation in [20]. However, since streaming XML is traversed in depth-£rst order, some transition combinations, such as visiting previous siblings, are not always applicable. It is also not easy to apply techniques such as the alternating automaton [8] to process streaming XML. For example, for a *universal* state in an alternating automaton, we need to get the results for all its children to label it as acceptance or rejection, which is not always applicable in the streaming environment.

Systems for *£ltering* XML document focus on searching a collection of XML documents for those that match a query. The output is thus restricted to a set of document identi£ers. Further, such systems typically either do not handle predicates or handle only predicates restricted to structural matching. The **XFilter** system uses £nite-state automata to £lter XML streams; performance is improved by indexing and by combining similar FSAs [1]. The **YFilter** system uses one FSA to evaluate all concurrently submitted £lter expressions [12]. It supports only predicates that do not reference other elements. Such predicates can be evaluated immediately when the element to which the apply is encountered. Further, £ltering systems such as YFilter do not need to handle situations in which predicates must be evaluated in different sequences (as in Example 2). Methods for indexing common subexpressions of XPath queries using a data structure called **XTrie** are presented in [7]. Another related topic of *query labeling* is studied in [18]. The authors propose a notion of a *requirements index* as a dual to the traditional data index. A framework is provided to organize the index ef£ciently and to label the nodes in streaming XML documents with all the matched requirements in the index.

Recall, from Section 4, that supporting closure and other features of XPath requires nondeterminism in automaton-based approaches (or, equivalently, a combinatorial explosion in the number of states). The **XMLTK** system uses a lazy deterministic £nite state automaton to which new states are added as needed (at runtime) [2, 14]. The determinism results in higher system throughput. The trade-off is that the deterministic automaton requires more memory than its nondeterministic counterpart. (The authors provide a bound on its size.)

A transducer-based approach is presented in [19] to answer queries written in XQuery. Its main idea is to decompose the query into subexpressions, each of which is mapped to an *XML Stream Machine* (**XSM**). The XSMs are arranged in a network by chaining the output of one to the input of another, based on the query semantics. Techniques for transforming this XSM network into a single XSM, and for optimizing XSMs based on DTDs, are provided. The key differences between XSQ and XSM are the following: First, XSM does not handle queries with aggregations and closures (such as the queries in Example 2 and Figure 13). Second, the chaining method used by XSM is not always suitable for streaming queries. For example, the semantics of aggregation functions is not easy to express using the chaining method. Third, the XSMs after combination and optimization are very complicated. It is dif£cult to group similar queries. In contrast, the HPDT used by XSQ has a simple and regular structure, so that multiple HPDTs can be grouped using methods suggested by [12]. Currently the XSQ system is schema-unaware. It is an interesting topic to automatically incorporate schema information , if available, into the system for optimization. (Since a release version of XSM was unavailable at the time of writing, XSM does not appear in the empirical studies of Section 6.) This network-of-transducers approach is also used by **SPEX** [11], which evaluates regular path expressions with quali£ers against well formed XML streams.

We brie¤y mention some work on querying *non-streaming* XML documents that is discussed further in Section 6. **Galax** is a full-¤edged implementation of the *XQuery* language, with static typing guarantees [13]. The *OCaml* implementation is based on a DOM materialization of an XML document. **XQEngine** is a full-text search engine and uses an XQuery-based query language that supports boolean combinations of keywords in order to query *collections* of XML documents. XQEngine must preprocess a document collection to create a full-text index that is used in query processing. **Saxon** provides a system for transforming XML data [17]. Transformations are speci£ed using XSL, which uses XPath expressions to specify patterns. Saxon, like other XSLT processors, needs to build a DOM tree of the entire XML document in main memory before performing any operations, restricting its utility in streaming systems.

Simple Transformations for XML (STX), implemented by the **Joost** system, is a transformation language designed for streaming XML [6, 5]. STX is more procedural than XSLT, and uses boolean program variables to store the results of each predicate in a query. Predicate evaluation sets the appropriate variables, which must be cleared explicitly. At any time, these predicates may be examined to determine appropriate actions (such as output). For any element in an XML stream, only the data that precedes it can be used to determine the actions on the element. This restriction simpli£es the implementation, since many of the complexities illustrated by Examples 1 and 2 do not occur.

Name	Support	Streaming	Mutiple predicates	Closure	Aggregation	Buffered predicate evalaution
XSQ−F	XPath	X	X	X	X	X
XSQ−NC	XPath	X	X		X	X
XMLTK	XPath	X		X		
Saxon	XSLT		X	X	X	—
XQEngine	XQuery		X	X	X	—
Galax	XQuery		X	X	X	—
Joost	STX	X		X	X	

Figure 14: System features

Name	Size (MB)	Text size (MB)	Number of elements (K)	Avg/Max depth	Average tag length
SHAKE	7.89	4.94	180	5.77/7	5.03
NASA	25.0	15.1	477	5.58/8	6.31
DBLP	119	56.4	2,990	2.90/6	5.81
PSD	716	286	21,300	5.57/7	6.33

Figure 15: Dataset descriptions

6. IMPLEMENTATION AND EXPERIMENTS

We have implemented the XSQ system in Java using Sun Java SDK version 1.4. The XML parser used is Xerces 1.0 for Java. We have implemented two versions of the XSQ system: XSQ-NC supports multiple predicates and aggregations, but not closures; XSQ-F, supports multiple predicates, aggregations, and closures. Figure 13 shows a screenshot for the GUI of the XSQ-F system. In the screenshot, we query the *macbeth.xml* in the Shakespeare play collection with an XPath query that contains two closure axes, two predicates, and an aggregation function.

6.1 Experimental Setup

We conducted the experiments on a Pentium III 900MHZ machine with 1 GB memory running the Redhat 7.2 distribution of GNU/Linux (kernel 2.4.9-34). The maximum amount of memory the Java Virtual Machine could use was set to 512 MB.

We compare the XSQ system with the systems in Figure 14, which process XPath queries or XPath-like queries. We have described Galax [13] (version 0.1α), XQEngine [16] (version 0.56), XMLTK [2] (version 0.9), Saxon [17] (version 6.5.2), and Joost (version 20020828) [5]in Section 5. Figure 14 summarizes the query language and some basic features of these systems.

Not all the systems can handle all sizes of datasets and all XPath queries. However, our goal is not simply to compare their performance. Through our study of these XPath processors, we want to get more insights of the cost to support certain XPath features such as closures and to predict which system will perform better in what kind of environment. For example, if we only want to use simple XPath fragment without predicates, we do not need full-¤avored XQuery engine such as Galax. However, if we need to express complicated queries that involve constructing new elements, we have to resort to systems such as Galax.

Since some systems use query languages that are supersets of XPath, or variations of XPath, we modify the XPath queries as needed to ensure that queries convey the semantics remain unchanged. In most cases, the results are enclosed by different container elements but the contents are the same.

In our experiments, we use the above systems to evaluate queries over datasets that differ in size and characteristics, including real and synthetic datasets. We use four real datasets [2]: the Shakespeare play collection (SHAKE), NASA ADC XML repository (NASA),

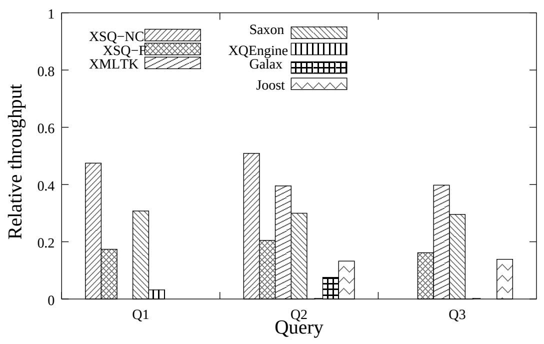


Figure 16: Relative throughputs of the systems of different queries on the SHAKE dataset

Q1:	`/PLAY/ACT/SCENE/SPEECH[LINE%love]/SPEAKER/text()`
Q2:	`/PLAY/ACT/SCENE/SPEECH/SPEAKER/text()`
Q3:	`//ACT//SPEAKER/text()`

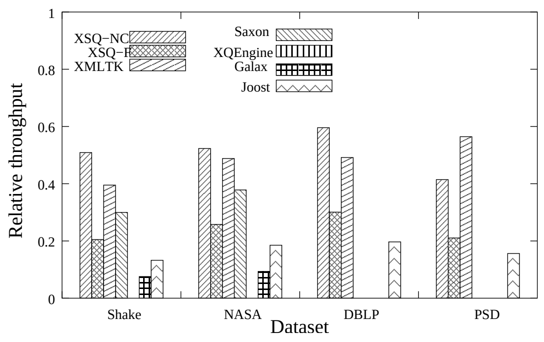


Figure 17: Relative throughputs of the systems when querying different datasets

Dataset	Query
SHAKE:	`/PLAY/ACT/SCENE/SPEECH/SPEAKER/text()`
NASA:	`/datasets/dataset/reference/source/other/name/text()`
DBLP:	`/dblp/article/title/text()`
PSD:	`/ProteinDatabase/ProteinEntry/reference /refinfo/authors/author/text()`

DBLP records (DBLP), and PIR-International Protein Sequence Database (PSD). Some characteristics of these datasets are listed in Figure 15. We also use synthetic datasets that are generated using IBM XML Generator [15] and Toxgene [4]. The characteristics of the synthetic datasets are described later with related experimental results.

6.2 Throughput

Throughput is an important metric for streaming systems since the data size varies and could be unbounded. All the systems in Figure 14 use the SAX API to parse the data. Therefore, the throughput of a SAX parser, which parses the XML data but does nothing else, gives an upper bound of the throughput for any XML query system. We wrote two parsing programs, named **PureParsers**, in C and Java. The *P*ureParser in C uses the Expat 1.2 parser used by XMLTK. The PureParser in Java uses Xerces 1.0 for Java, which we specify to use in XSQ-NC, XSQ-F, XQEngine, Saxon, and Joost in the experiments. C parsers are generally faster than Java parser since parsing involves a large number of string operation, which is more ef£cient in C parsers. For the 119MB DBLP dataset, the C PureParser £nises parsing in 10.6 seconds and the Java PureParser uses 28.2 seconds. Instead of raw throughput, we use the normalized throughput of the systems with respect to the throughput of the corresponding PureParser, called **relative throughput**, to measure the performance of the systems written in different programming languages and using different parsers. Galax implements its own parser in Ocaml. Here we use the Java PureParser instead since we do not have an Ocaml SAX parser, which we believe is faster than the Java PureParser.

Figure 16 shows the relative throughputs of the systems when they evaluate different queries on the SHAKE dataset. Figure 17 shows the relative throughput when they query different datasets. We can see that XMLTK and XSQ-NC are the fastest two systems when applying queries that they can handle. An important reason is the determinism in both systems. Although XSQ-NC has to buffer some of the data sometime, its underlying PDT is deterministic. Even when processing the same query without closure, XSQ-NC is faster than XSQ-F since XSQ-F uses a non-deterministic PDT. When searching for a matching transition arc in the automaton, XSQ-NC can stop searching after it £nds one match. In contrast, XSQ-F has to go through all the transition arcs of the current state to make sure every arc is handled. Moreover, as we have shown in Example 2, when an item is in the result, the XSQ-NC can output it immediately, while XSQ-F needs to do extra work to ensure that the item will not be outputted twice due to the non-determinism.

We can see from Figure 16 and Figure 17 that Saxon is faster than XSQ-F when they process XML data that can £t into main memory. Saxon uses the SAX parser to load all the data into the memory and build the DOM tree before it evaluates the query. After parsing the data, Saxon does all the process in main memory. In memory processing is ef£cient and can support more powerful queries. However, it is not suitable for streaming data in general. Moreover, as we will see next, the amount of memory it needs is usually 4 to 5 times of the £le size. Thus, it cannot scale up to process large XML £les.

We also study the time the systems spend on each phase of query evaluation. The dark bar in Figure 18 represents the query compilation time, which usually includes parsing the query and building the query engine. The gray bar represents the preprocessing time. For example, Saxon loads all the data into memory to build the DOM tree before it can evaluate the queries. XQEngine builds the full-text index before it can query the data.

From Figure 18, we see that an advantage of the streaming systems is that they can return the available results as soon as the data is available, which is crucial if the response time of the system is important. The non-streaming systems have to wait until all the preprocessing £nish to begin evaluating. However, as long as these systems remain in memory, the subsequent queries can be evaluated much faster since the results of the preprocessing can be reused then.

6.3 Memory Usage

Memory usage is critical for the scalability of the streaming system. Non-streaming systems need memory linear in the size of the input since they need to load the whole dataset into memory. In contrast, streaming systems need to store only a small fraction of the stream. Figure 19 shows the memory usage reported for the queries over the datasets size from 5MB to 50MB. All the datasets are excerpts of the DBLP dataset. For example, the 10MB dataset contains the £rst 10MB data in the dataset. (The size is an approximate since we have to include the closing tag of the elements at the 10MB offset.) From Figure 19, we see that Saxon and Galax use memory roughly linear in the size of the input data. Linear memory usage, with a constant factor of 4 to 5, makes the DOM-based system unsuitable for large XML £les.

We also use the XML Generator program to generate datasets of

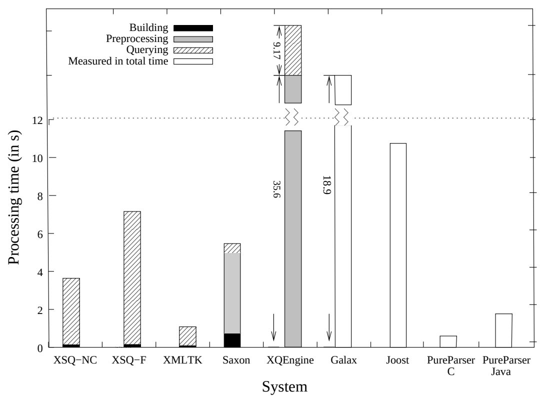


Figure 18: Preprocessing time, query processing time, and total querying time

Dataset: SHAKE Query: `/PLAY/ACT/SCENE/SPEECH/SPEAKER/text()`

Note: We were unable to determine the separate times for Joost and Galax.

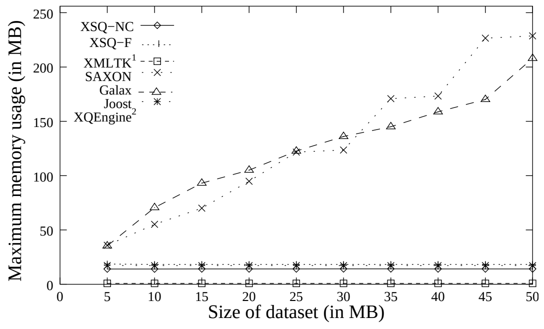


Figure 19: Memory usage of the systems when querying DBLP dataset of different sizes

Query: `/dblp/inproceedings[author]/title/text()`

1. The query for XMLTK : `/dblp/inproceedings/title/text()`

2. XQEngine is not tested since it currently supports only 32K elements per document.

varying size and recursiveness. For example, for the dataset of size 13MB, the nested level parameter of the XML Generator program is set to 15 and the maximum repeats parameter is set to 20. From Figure 20 we can see that even the highly recursive data and queries with closures, the amount of memory XSQ-F uses is still constant. Recall from Section 4 that XSQ needs to buffer more data if there are closures in the query. However, since all the items in the buffers can be determined when we encounter the end event of the element speci£ed in the £rst location step (the HPDT returns to the highest level BPDT), the maximum amount of memory the XSQ needs cannot exceed the maximum size of the elements in the stream.

6.4 Characterizing the XPath Processors

Some systems are sensitive to the order of the elements in the data if the elements are involved in the query. We generate a 10MB dataset using Toxgene, in which the following template is applied repeatedly to generate new elements `a` with increasing `id` attributes.

```
<a id="1"> <prior> 1 </prior>
     <foo> 1 </foo> (repeat 10,000 times)
     <posterior> 1 </posterior> </a>
```

Though all the queries `/a[prior=0]`, `/a[posterior=0]`, and `/a [@id=0]` return null result set, different systems behave differently in our experiments. In Figure 21, the throughputs of the Saxon system are almost the same, since it always loads all the data into the memory before it evaluates the queries. When it traverses the DOM tree in the main memory to evaluate the query, the document order is not important. However, the throughput of XSQ-NC is 30% larger in the last query than the other two queries. For the last query, XSQ-NC can decide at the beginning of the `a` element that all the contents in this element can be ignored. In the other two queries, all the data in the current element have to be buffered until the closing tag of the `a` element is met, which makes XSQ-NC much slower. XSQ-F is not as sensitive as XSQ-NC to the order. Recall from Section 4 that even if XSQ-F knows an item is in the result set, it marks the item as “output” £rst and output the item until it handles all the possible transitions due to the non-determinisms.

We also study the sensitivities to the result size of the systems. Most systems are sensitive to the result size, but in different degrees. For example, the XQEngine is slower than the other systems in Figure 18 since the query returns a large portion of the dataset. But if the query contains a tag that is not in the data, XQEngine returns the empty result set immediately. The other systems spend similar amount of time on the query no matter whether the tags in the query appear in the document or not.

We use Toxgene to generate a test dataset of 10MB consisting of three types of elements (besides a few top level elements): 10% of the elements have tag *red*, 30% *green*, and 60% *blue*. The content of each such element is a character. Figure 22 shows the relative throughputs of systems when the query returns the three different types of elements.

We can see that XSQ-NC is sensitive to the result size. The difference in the performance is due to the different handling of data items based on whether they are in the result. Items that are not in the result can be ignored and XSQ-NC stays in the same state. If there are more items in the result set, the XSQ-NC will make more state transitions and output operations, which consist a large portion of the running time of XSQ-NC. XSQ-F is not as sensitive as XSQ-NC. As described in Section 4, it always keeps the item £rst, no matter it is in the result or not, and checks the queue after all transition arcs are handled. The difference between the handlings is not as large as in XSQ-NC. Saxon is less sensitive to the result size since after it loads all data into main memory, the evaluation process is done in main memory except the output process, which constitutes only a small amount of the total execution time. The smaller difference in XMLTK might be caused only by the I/O cost as well. However, it is not clear why Joost is not sensitive to the

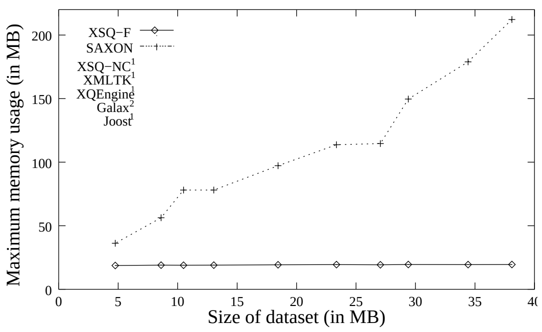


Figure 20: Memory usage of the systems when querying synthetic datasets of different sizes

Query: `//pub[year]//book[@id]/title/text()`

1. The system cannot handle the query in the dataset.

2. Galax reports “stack over¤ow” error when we try the query.

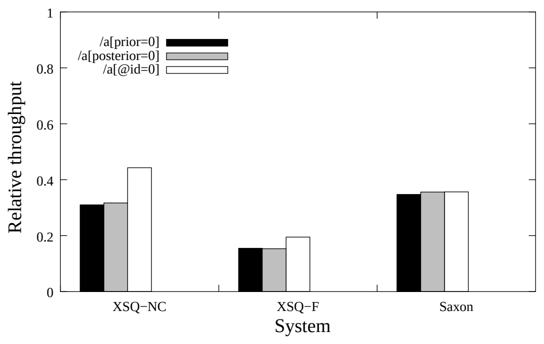


Figure 21: Effect of data ordering on throughput

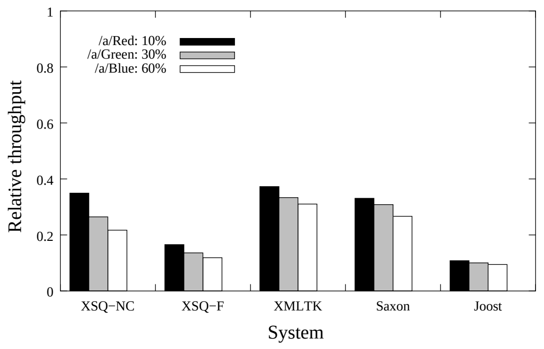


Figure 22: Effect of the result size on throughput

result size.

7. CONCLUSION

In this paper, we have described the design and implementation of the XSQ system for evaluating XPath 1.0 queries on streaming XML data. A distinguishing feature of XSQ is that it buffers only data that must be buffered by *any* streaming XPath query processor. Further, XSQ has a clean design based on a hierarchical network of pushdown transducers augmented with buffers. The XSQ system is fully implemented, and supports features such as multiple predicates, closures, and aggregation. We also presented an empirical study of XSQ and related systems in order to explore the costs and bene£ts of XPath features and implementation choices.

8. REFERENCES

[1] M. Altinel and M. J. Franklin. Ef£cient Filtering of XML Documents for Selective Dissemination of Information. In *The VLDB Journal*, pages 53–64, 2000.

[2] I. Avila-Campillo, D. Raven, T. Green, A. Gupta, Y. Kadiyska, M. Onizuka, and D. Suciu. An XML Toolkit for Light-weight XML Stream Processing, 2002. `http://www.cs.washington.edu/homes/suciu/XMLTK/`.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *The 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, Madison,Wisconsin, June 2002.

[4] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a template-based data generator for XML. In *Fifth International Workshop on the Web and Databases*, Madison, Wisconsin, June 2002.

[5] O. Becker. Joost is Ollie's Original Streaming Transformer, 2002. `http://joost.sourceforge.net/`.

[6] O. Becker, P. Cimprich, and C. Nentwich. Streaming Transformations for XML. `http://www.gingerall.cz/stx`.

[7] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Ef£cient Filtering of XML Documents with XPath Expressions. In *The 18th International Conference of Data Engineering*, 2002.

[8] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM (JACM)*, 28(1):114–133, 1981.

[9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *The ACM SIGMOD Conference*, pages 379–390, 2000.

[10] B. Choi. What are real DTDs like. In *Fifth International Workshop on the Web and Databases*, Madison,Wisconsin.

[11] F. Dan Olteanu, Tobias Kiesling. An Evaluation of Regular Path Expressions with Quali£ers against XML Streams. Technical Report PMS-FB-2002-12, Institute for Computer Science, Ludwig-Maximilians University, Munich, May 2002.

[12] Y. Diao, P. Fischer, and M. J. Franklin. YFilter: Ef£cient and Scalable Filtering of XML Documents. In *The 18th International Conference of Data Engineering*, San Jose, February 2002.

[13] M. Fernandez and J. Simeon. Galax, 2002. `http://db.bell-labs.com/galax/`.

[14] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with Deterministic Automata. In *The 9th International Conference on Database Theory*, Siena, Italy, January 2003.

[15] IBM. XML Generator, 2001. `http://www.alphaworks.ibm.com/tech/xmlgenerator`.

[16] H. Katz. XQEngine, 2002. `http://www.fatdog.com`.

[17] M. H. Kay. SAXON: an XSLT processor, 2002. `http://saxon.sourceforge.net/`.

[18] L. V. Lakshmanan and P. Sailaja. On Ef£cient Matching of Streaming XML Documents and Queries. In *International Conference on Extending Database Technology*, pages 142–160, Prague, Czech Republic, March 2002.

[19] B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *The 28th International Conference on Very Large Data Bases*, Hong Kong, August 2002.

[20] T. Milo, D. Suciu, and V. Vianu. Typechecking for xml transformers. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 11–22. ACM Press, 2000.

[21] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Workshop on XML-Based Data Management (XMLDM) at the 8th Conference on Extending Database Technology*, Prague, March 2002. Springer-Verlag.

[22] L. Segou£n and V. Vianu. Validating Streaming XML documents. In *The 21st ACM Symposium on Principles of Database Systems*, pages 53–64, Madison, Wisconsin, June 2002.

[23] XSL Working Group and the XML Linking Working Group. XPath XML Path Language (XPath) 1.0. W3C Recommendation, W3C, `http://www.w3.org/TR/xpath/`, November 1999.