XSQ: Streaming XPath Queries

Feng Peng   Sudarshan S. Chawathe

Department of Computer Science
University of Maryland, College Park, Maryland 20742, U.S.A.
{pengfeng,chaw}@cs.umd.edu

1 Introduction

XML is becoming the de facto standard for information exchange and the amount of XML data is growing rapidly. Some of this data is accessible only in *streaming* form. That is, data items are presented in a fixed 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), it is useful to process large XML datasets in streaming form because of the greater efficiency 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 on streaming XML. (XPath is an emerging standard query language that is useful by itself, and forms an important part of more expressive languages such as XQuery.) A streaming query engine or XPath cannot rely on any method that requires instantiation of a large subset of the data. For example, methods based on the DOM interface to XML do not satisfy this requirement. Some of the distinguishing features of our streaming XPath query processor, called XSQ, are as follows:

- To the best of our knowledge, XSQ is the first system for querying XML data streams that handles XPath features such as closures, aggregations, and multiple predicates. Recent work on querying XML streams has addressed some of these issues separately, but not in combination. For example, the XMLTK program [ACR+02] allows retrieval of a portion of an XML file specified using XPath. However, XMLTK does not support predicates in XPath expressions. The XSM system [LMP02] handles predicates in the query but it does not handle the closures and aggregations. As discussed later, the combination of the closures and multiple predicates introduces substantial difficulties.

- XSQ is very efficient. For example, we duplicate the DBLP dataset to generate a dataset of one gigabyte in our experiments. Parsing the dataset alone takes 297 seconds. Evaluating the query `//article[@key]//title/text()` over this dataset requires only 84 seconds of additional time to return all the results. Initial results are produced within one second of issuing the query. Another feature of the system is that it is very efficient in its use of main memory. Specifically, a data item is buffered by XSQ only if its membership in the query result cannot be decided based on the data seen so far. Such a data item must therefore be buffered by any streaming query processor that produces complete results.

- The XSQ system uses a clean design based on hierarchical pushdown transducers that consist of standard transducers augmented with queues. The system is easy to understand, implement, prove correct, and expand to more complex queries.

2 The XSQ System

The basic idea of the XSQ system is to use a pushdown transducer (PDT) to process the events that are generated by a SAX parser when it parses XML streams. A PDT is a pushdown automaton (PDA) with actions defined along with the transition arcs of the automaton. A PDT is initialized in the start state. At each step, based on the next input symbol and the symbols in the stack, it changes state and operates the stack according to the transition functions. The PDT also defines an output operation which could generate output during the transition. In the XSQ system, the PDT is augmented with a buffer so that the output operation could also be the buffer operations.

An XPath query in the XSQ system consists of a pattern expression, which is specified by a **location path**, and an output expression. The location path is a sequence of **location steps** that specify the path to a desired element. The output expression specifies what portions or functions of the element should be outputted. Each location step has an axis, a node test, and an optional predicate. For example, in the query `//pub[year>2000]/book/name/text()`, `//pub [year>2000]/book/name` is the pattern expression. The output expression, `text()`, specifies that only the text in the elements should be outputted. In the first location step `//pub[year>2000]`, `//` is the axis denoting closure, `pub` is the node test, and `year>2000` is the predicate.

Without predicates and the output expression, an XPath query can be deemed as a filter pattern that could be used to filter the XML documents in a collection, returning only the documents that match the pattern. Since these XPath queries are essentially regular expressions, they could be converted directly to finite state automata (FSAs) which accept the same set of XML documents. Systems such as [AF00, CFGR02] use FSAs to filter the XML streams and focus mainly on grouping and indexing similar XPath expressions. With predicates and output expressions in the XPath query, it is not so straightforward to convert the XPath query into an transducer that answers the query. When the query includes aggregations and closures, the problem becomes even more difficult. We briefly describe some of the difficulties below:

- The sequence of elements in the data stream makes it possible that some of the predicates cannot be evaluated when the we encounter an element that satisfies the pattern expression of the query. Thus we need to buffer these elements until the data required to evaluate the predicates is available. For example, consider the query `/book[year=2002]/name`. In the data stream, the year child of a book element may come after the name child of the book. Thus when we encounter the name child, we need to buffer it. Only after we encounter the year child of the book can we decide whether the name should be send to output.

- The predicates, which come in various forms, need to be associated with different SAX events. For example, for the query `/book[author@age<25]/name`, the predicate is evaluated upon the begin event of the author child of the book element. If one of the authors of the book is younger than 25, the name of the book should be outputted. Thus we need to see all the authors before we can decide that the name is not in the result. In contrast, for the query `/book[@id>25]/name`, we can decide at the begin event of the book element whether the name should be outputted since the attribute `id` always comes in the begin event of the book element in the stream.

- When a predicate is evaluated, we need to first decide what items in the buffer are affected by the result of this predicate. If the predicate evaluates to false, we need to remove all the items that are affected by it from the buffer right away. If the predicate evaluates to true, for each item in the buffer that is affected by the predicate, we need to decide whether there are other predicates unevaluated required by the item. If not, the item now satisfies all its predicates and should be send to the output immediately. Otherwise, we have to keep track that the item has satisfied the current predicate but is still waiting for the evaluation of other predicates.

- When there are closure axes "//" in XPath queries denoting "descendant-or-self", the above problem becomes more complicated. There may be several different ways that the path to an element matches the pattern expression of the query. Each match gives a different evaluation of the predicates. Some of these evaluations may be false. However, as long as there is one match for which all predicates evaluate to true, the element should be included in the result. At the same time, duplicates should be avoided if multiple matches have true evaluation for all predicates.

Our solution uses a hierarchically structured PDT, called HPDT, that consists of smaller PDTs that have their own buffers. Each small PDT, called a basic PDT (BPDT), is generated using templates based on the kinds of location steps in XPath queries. The templates are based on the following categorization of location steps:

1. Test whether the current element has a specified attribute, or whether the attribute satisfies some condition, (e.g., `/book[@id]`, `/book[@id ≤ 10]`).

2. Test whether the current element contains some text, or whether the text value satisfies some condition, (e.g., `/year[text() = 2000]`).

3. Test whether the current element has a specified type of child, (e.g., `/book[author]`).

4. Test whether the the current element's specified child contains an attribute, or whether the value of the attribute satisfies some condition, (e.g., `/pub[book@id ≤ 10]`) .

5. Test whether the specified child of the current element has a value that satisfies some condition, (e.g., `/book[year ≤ 2000]`).

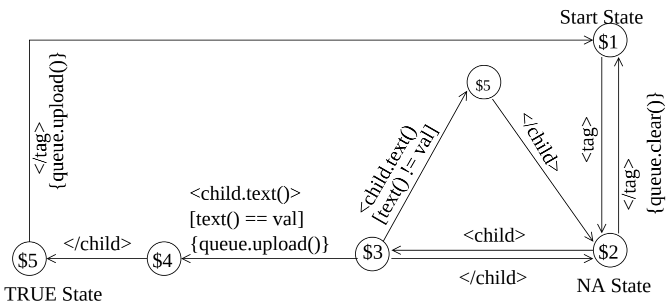

Figure 1: Template of BPDT for a location step of the form `/tag[child=val]`

Figure 1 depicts a BPDT that processes a single location step with a predicate testing the text of a child. In each BPDT, there is a *TRUE* state that indicates the predicate in this location step has been evaluated to be true and an *NA* state that indicates the predicate has not been evaluated yet. Each BPDT also has its own buffer which is organized as a queue. In Figure 1, notice that only when all children have failed the test and the current element reaches its end event will the automaton clear the content of its buffer, which exactly expresses the logic of the predicate. Due to the space limit, the graphs of other templates are omitted here. Details appears in our technical report.[1]

[1] Available at http://www.cs.umd.edu/~pengfeng/xsq/

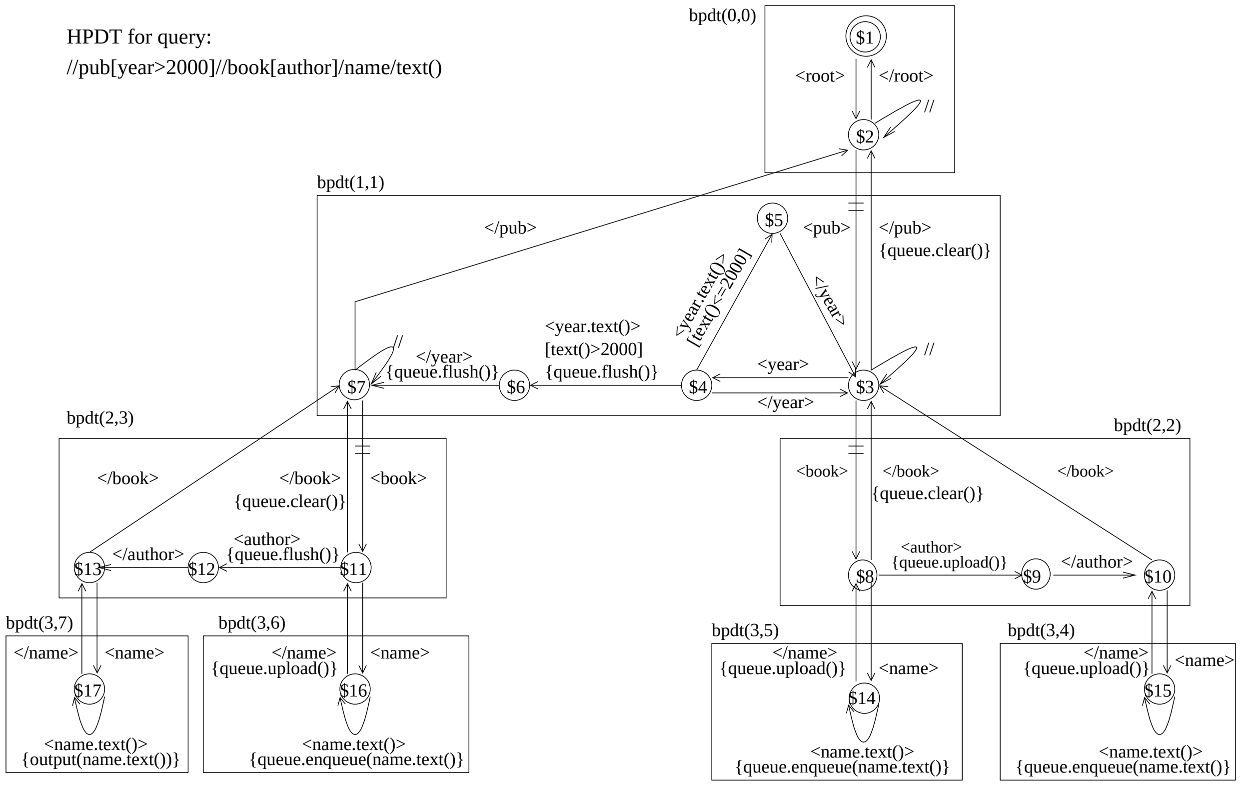

Figure 2: HPDT generated for query: `//pub[year>2000]//book[author]/name/text()`

Next we use an example HPDT shown in Figure 2 to illustrate how to build the HPDT. Each BPDT has an unique id consisting of the level and the id inside the level. We first generate the single bpdt(0,0) for the root axis in level 0, which exists in all XPath queries. Then the *TRUE* state of bpdt(0,0) is used as the start state of bpdt(1,1) in level 1, which is generated for the first location step. For both *TRUE* state \$7 and *NA* state \$3 of bpdt(1,1), we generate a BPDT in level 2 for the second location step. The difference between them is the operations on the transition arcs. In bpdt(2,3), which is connected to the *TRUE* state, we **flush** the content in the queue to the output if current predicate is satisfied. In the contrast, since bpdt(2,3) is connected to the *NA* state, which indicates that the predicate in the upper level has not been evaluated yet, we have to **upload** the content in the queue to the upper level BPDT after the current predicate evaluates to true. In both BPDTs, the queue will be **cleared** if the predicate evaluates to false at the end of the element. In the BPDT in the lowest layer, the content of the potential result is put into the queue unless we know that all the previous predicates are true, in which case we should output the data directly. Thus we can see that only in bpdt(3,7) do we output the data directly, and in all the other BPDTs in the lowest level, we **enqueue** the data into the queue.

There are 3 location steps in the query. Instead of generating $2^4 - 1 = 15$ BPDTs, which is the maximum number of BPDTs a 4-level HPDT could have, there are actually only 8 BPDTs in the system because only BPDTs with predicates have two children. Thus the number of the states in the HPDT is reduced significantly. Moreover, in the implementations, because of the similarities of the BPDTs in the same level, we actually do not need to instantiate all the BPDTs. It suffices to only create one BPDT for each layer and use bitmap flags to keep track of the operations. As to the memory usage, from Figure 2 we can see that the HPDT only puts the items into the buffer when they satisfy the pattern expression but have predicates that cannot be evaluated yet. As soon as one of the predicates evaluates to false, the content in the buffer of the current BPDT will be cleared right away. Meanwhile, if all the predicates have been evaluated true, the content will be flushed to the output immediately. Thus the system only buffers data that any streaming system must buffer.

In order to process closures and aggregations, we make some extensions to the basic ideas presented above. For aggregations, the HPDT is augmented with a statistics buffer. It uses templates for the possible aggregation functions in the lowest level BPDTs. For closures, the basic idea is to keep track of all the possible paths toward the current element that match the pattern expression. Unless all the paths have failed the predicates, the content will be kept in the queue. Due to space limitation, the details to handle the closures and aggregations are omitted here. Our experiments show that both features are handled without any significant degradation of performance.

3 Demonstration

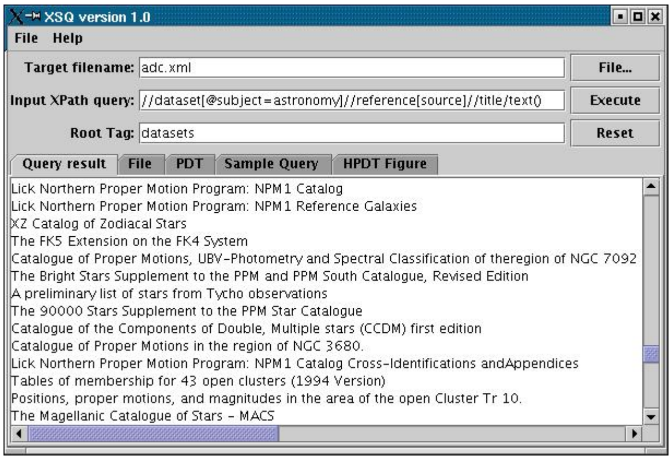

Figure 3: Screenshot of the XSQ interface

We have implemented the XSQ system using Sun Java SDK version 1.4 and Xerces 1.0 parser for Java. The experiments use the Redhat 7.2 distribution of GNU/Linux (kernel 2.4.9-34) on a Pentium III 900MHZ machine with 1GB of main memory.

For the demonstration, we use large XML files (20MB to 2GB) to simulate XML streams. One dataset is synthetic data generated using IBM XMLGen. Another dataset is generated by duplicating the DBLP dataset of size 119MB, which is commonly used as the test set for XML query systems. Other real life datasets, such as NASA ADC XML repository of size 24MB and Swiss-Port Protein knowledgebase of size 114MB , are also part of the demonstration. (Any well formed XML data stream can be used as input data.)

We use queries such as the one in Figure 2 to demonstrate various features of the system: closures, predicates on the text of a child, and multiple predicates in a query. Even if the data is heavily nested and there are multiple closures in the query, we demonstrate that the amount of memory used to process the query is still small. We also demonstrate that the XSQ system generates output while the SAX parser is still parsing the XML file. In the graphical interface shown in Figure 3, we also display the structure of the HPDT (similar to the graph in Figure 2). (Users can pose other XPath queries to test the performance of the system.)

In our experiments, the XSQ system is faster than other currently available systems that we tested, such as Galax [FS02], Joost [BCN02], and XALAN. For example, it took the Galax system, which is DOM-based, about 47 seconds to evaluate the query Q4 in Figure 4 while XSQ only uses 10.4 seconds. Further, in our experiments with these, and some other similar systems, the DBLP and Swiss-Port datasets exhausted available main memory. An exception is the XMLTK [ACR+02] program which is about 50% faster than XSQ. XMLTK is efficient since it uses a simpler automaton without buffers. Because XMLTK does not handle predicates and aggregations, it always outputs the result directly. Moreover, buffer operations, which are essential to process predicates and aggregations, introduce significantly larger amount of string operations. Since parsing accounts for a significant fraction of query processing time, another factor that affects the performance is the different SAX parsers used in the systems XSQ uses Xerces for Java that is slower than Expat parser written in C that is used by XMLTK. To demonstrate the efficiency of the XSQ system, we also compare the speed of the XSQ system with a pure parser which parses the XML file and does nothing else. The pure parser gives an lower bound on the time of a streaming processing system needs to process the streams. Some sample queries and experimental results are shown in the Figure 4. The query times refer to the times for computing the entire result sets. In all cases, initial (streaming) results are available within 1 second.

Dataset	Size	Query	Query Time	Parse Time
DBLP	119MB	Q1	20.9s	48.3s
DBLP-1GB	1024MB	Q2	84s	298s
SYN-2GB	2135MB	Q3	224s	774s
NASA	25MB	Q4	2.6s	7.88s
Swiss-Port	114MB	Q5	11.7s	49.6s

Q1: `//inproceedings[year>1995]/count()`
Q2: `//article[@key]//title/text()`
Q3: `/pub[year]/book[@id=a3]/author/name/first/text()`
Q4: `/dataset[@subject=astronomy]/reference/source/other[date]/name`
Q5: `/Entry/Ref/Cite`

Figure 4: Representative experimental results

References

[ACR+02] I. Avila-Campillo, D. Raven, et al. An XML toolkit for light-weight XML stream processing, 2002. http://www.cs.washington.edu/homes/suciu/XMLTK/ .

[AF00] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of 26th International Conference on Very Large Data Bases*, pages 53–64, Cairo, Egypt, 2000.

[BCN02] O. Becker, P. Cimprich, and C. Nentwich. Streaming transformations for XML, 2002. http://www.gingerall.cz/stx.

[CFGR02] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *18th Intl. Conf. on Data Engineering(ICDE)*, pages 235–244, San Jose, February 2002.

[FS02] M. Fernndez and J. Simon. Galax, 2002. http://db.bell-labs.com/galax/.

[LMP02] B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou. A transducer-based XML query processor. In *Proceedings of 28th International Conference on Very Large Data Bases*, Hongkong, August 2002.