XSQ: A Streaming XPath Engine

FENG PENG
SUDARSHAN S. CHAWATHE
Department of Computer Science
University of Maryland, College Park

We have implemented and released the XSQ system for evaluating XPath queries on streaming XML data. XSQ supports XPath features such as multiple predicates, closures, and aggregation, which pose interesting challenges for streaming evaluation. Our implementation is based on using a hierarchical arrangement of pushdown transducers augmented with buffers. A notable feature of XSQ is that it buffers data for only as long as it must be buffered by any streaming XPath query engine. We present a detailed experimental study that characterizes the performance of XSQ and related systems, and illustrates the performance implications of XPath features such as closures.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*query processing*

General Terms: Experimentation, Performance

Additional Key Words and Phrases: XPath, streaming processing

1. INTRODUCTION

The XSQ system is an XPath engine for streaming XML data. We begin this section by describing the characteristics and sources of streaming XML data. We introduce XPath using a simple example and briefly touch on some prior work in the area. We then outline the distinguishing features of XSQ and the contributions of this paper. Next, we present some examples that illustrate some of the challenges faced by an XPath query engine that operates in a streaming environment. We end the section with a map of the rest of the paper.

The Extensible Markup Language (XML) has become a well-established data format and an increasing amount of information is becoming available in XML form [Bray et al. 1998]. The term *streaming data* is used to describe data items that are available for reading only once and that are provided in a fixed order determined by the data source. Applications that use such data cannot seek forward or backward in the stream and cannot revisit a data item seen earlier unless they buffer it on their own. Examples of data that occur naturally in streaming form include real-time news feeds, stock market data, sensor data, surveillance feeds, and data from network monitoring equipment. One reason for some data being available in

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

only streaming form is that the data may have a limited lifetime of interest to most consumers. For example, articles on a topical news feed are not likely to retain their value for very long. Another reason for such data is that the source of data may lack the resources required for providing non-streaming access to data. For example, a network router that provides real-time packet counts, error reports, and security violations is typically unable to fulfill the processing or storage requirements of providing non-streaming (so-called *random*) access to such data. Similar concerns may lead servers hosting large files to offer only streaming network access to data even though the data is available internally in non-streaming form. Finally, since sequential access to data is typically orders of magnitude faster than random access, it is often beneficial to use methods for streaming data on non-streaming data as well. In what follows, we focus on streaming data that is in XML form and use the term **streaming XML** to refer to XML data in all of the above scenarios.

There have been a number of recent proposals on query languages for XML and XML-like data models [Abiteboul et al. 1996; Fernandez et al. 1997; Buneman et al. 1996; Deutsch et al. 1998; Clark and DeRose 1999; Boag et al. 2002]. Of these proposals, XPath and XQuery have emerged as the standards recommendations that are likely to receive broad support. In this paper, we focus on XPath. However, since XPath forms an important core of XQuery, the methods we describe are useful not only for XPath engines, but also for XQuery engines. An **XPath query** consists a **location path** and an **output expression**. We may think of the location path as a selection operator and the output expression as a projection operator. The former selects a set of XML elements and the latter determines the parts, or functions, of those elements that form the result. While the projection operator in XPath is quite simple, the selection operator is fairly complex because it permits predicates on all elements that lie on the path from the document root to the selected element. For example, the location path of the query `//book[year>2000]/review[@source="BN"]/text()` is `//book[year>2000]/review[@source="BN"]`. This location path matches the review elements that have a `source` attribute with value `BN` and that are children of book elements that have year subelements with values greater than 2000. Interpreting the location path as a path expression, the `/` connective denotes a child and the `//` connective denotes a descendant. The output expression, `text()`, indicates that the result consists of the text contents of reviews matching the location path. (Further details on XPath appear in Section 2.3.)

Automaton-based methods for processing streaming data are attractive due to their efficiency and clean design. An important task in building such systems for XPath queries is the generation of the automaton from the query. The difficulties (explained further by the examples below) are due to XPath features such as closures and predicates in conjunction with the read-once nature of streaming data. Briefly, 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 such as flagging buffered data based on subsequent satisfaction or falsification of predicates, and duplicate avoidance.

There has been a considerable amount of work on stream processing and XML

```
<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>
```

Fig. 1: Input data for 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>
```

Fig. 2: Input data for Example 2

query processing, and some recent work on query processing for streaming XML as well. Below, we touch on only on recent work that is most closely related to our work, deferring a longer discussion to Section 7. Much of the previous work on processing streaming XML data focuses on *filtering* a collection of XML documents using restricted XPath expressions [Altinel and Franklin 2000; Diao et al. 2002; Chan et al. 2002]. Since XPath expressions without predicates are essentially regular expressions, they can be transformed into finite state automata that accept exactly the documents that satisfy the expressions. If the automaton accepts the document, the filtering system returns the identifier of the current document to the user. 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 [Avila-Campillo et al. 2002] 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 directly to output and no buffering is needed. 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 [Ludascher et al. 2002] 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 significant challenges to query evaluation.

We note that XPath features such as (multiple) predicates, closures, and aggregations are important usability advantages, especially if the data is semistructured or has a structure unknown to the query formulator. Closures, in particular, are indispensable in queries on data whose structure is partly unknown. For example, the query `//book[author="Adams"]//price` returns the prices of books by Adams in a variety of likely structuring of bibliographic data, regardless of whether book occurs at the top level in the document or several levels deep and, similarly, regardless of whether the price element is a child of the book element or a descendant separated by intervening `bookstore` elements. 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 recent survey of 60 real datasets found 35 to be recursive [Choi 2002].)

The major **contributions** of this paper may be summarized as follows:[1]

—To the best of our knowledge, our method for evaluating XPath queries over streaming data is the first one that handles closures, aggregations, and multiple predicates (together). As the examples below illustrate, these features, especially in conjunction, pose significant 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. Further, this method provides a clean separation between high level design and lower-level implementation techniques. For example, it is easy to use our methods in a query engine that implements our automaton independently.

—We present a detailed experimental study of XSQ and several related systems in Section 9. In addition to providing a comprehensive evaluation of the methods we propose, our study also illustrates the costs and benefits of different XPath features and implementation trade-offs as embodied by different systems.

—All the methods described in this paper are fully implemented in the XSQ system, which has been publicly released under the GNU GPL license [Peng and Chawathe 2002; GNU 1991]. The Java-based implementation should work on any platform for which a Java virtual machine is available. 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).

EXAMPLE 1. Consider the following query for the XML stream depicted in Figure 1: `/pub[year>2000]/book[price<11]/author`. When we encounter the first `author` element in the stream, we know that it satisfies the path `/pub/book/author`. However, the predicate in the first location step, `[year > 2000]`, cannot be evaluated yet because we have not encountered any `year` subelements and qualifying elements may occur later. We have encountered the first `price` subelement of the `book` element. This element does not satisfy the predicate `[price < 11]`. However, we cannot conclude that the `book` element of line 3 fails to satisfy the predicate on price because there may be additional `price` subelements later in the stream. Therefore we must to buffer the `book` element. Indeed, when we encounter the second `price` element (line 7) of this `book` element, we determine that the book element satisfies the predicate on price. At this point, we still do not know whether the `pub` element of line 2 satisfies the predicate on year. Consequently we do not know whether the `author` element of line 6 belongs to the result. Therefore, we

[1] A brief outline of our methods and the results of a preliminary experimental study of XSQ appear in [?].

must continue to buffer the pub and author elements. When we encounter the two author subelements of the second book (lines 12 and 13), we need to buffer them as well. Now there are three author elements in the buffer: two with value A and one with value B. Next, we encounter the second price element (line 14) of the second book and find that it does not satisfy the predicate. However, only when we encounter the end tag of the second book element, can we conclude that this book element fails to satisfy the predicate [price < 11]. Consequently, the two author elements of the second book should be removed from the buffer. Note that one author with value A should remain in the buffer because it belongs to the first book. When we encounter the year element on line 16, we determine that the pub element satisfies the first predicate. Recalling that the author element remaining in the buffer has already satisfied the other predicate, we determine that this author element should be sent to the output.

The above example, although quite simple, illustrates some of the intricacies that we must handle: 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 in the result. We need to buffer such potential result items. In Example 1, we buffered three author elements as well as the pub and book elements for this reason. 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. In Example 1, for instance, we needed to delete the author elements belonging to the second book while retaining the author element for the first book in the buffer. Third, in order to buffer items for the least amount of time possible, we need to encode the implicit existential quantification within predicates: When a single item satisfying a predicate is found, we must check whether the elements within the scope of the newly satisfied predicate can be sent to the output. On the other hand, we cannot delete items from the buffer until we encounter the end tag of the appropriate element. In the above example, for instance, only we reach the end of the second book element may we conclude that it fails to satisfy the predicate on price. 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, are discussed in detail later.

Let us now consider a slightly more complex example, featuring closures in the query and recursive structure in the input stream. We say an XML stream has **recursive structure** if it contains ancestor-descendant pairs of elements that have the same tag. Figure 2 depicts an example of such data: the pub element in line 2 has a grandchild named pub in line 9. As the following example illustrates, such recursive structure in the input poses additional challenges to XPath query processing.

Example 2. Consider the following query for the XML data of Figure 2: //pub[year>2000]//book[author]//name. (The predicate [author] checks for the existence of an author child.) 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. For instance, the pub elements in both line 1 and line 9 match the node test in the first location step. Consider the `name` element in line 11. There are three ways in which it matches the main trunk of the query (ignoring predicates) and each matching yields a different result for the predicates in the query. The situation is summarized by the following table, which indicates the truth value of the two predicates for each matching (combination of `pub` and `book` elements) that leads to the `name` element of line 11:

pub	book	`[year > 2000]`	`[author]`	`name`
line 2	line 7	true	false	line 11
line 2	line 10	true	true	line 11
line9	line 10	false	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 of line 15, we know that the `pub` element of line 9 fails the predicate `[year > 2000]`. However, we cannot remove the `name Z` from the buffer because it is still possible that this item satisfies the query due to a subsequent `year` element. A similar situation occurs 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 need to be careful with the other cases where multiple matches evaluate all predicates to true. For example, if there were an additional `author` element between lines 8 and 9, the match indicated by the first row of the above table would also result in both predicates being satisfied. In such cases, we must avoid duplicates (outputting the same element twice).

These examples illustrate the difficulties encountered in designing an automaton for evaluating XPath queries systematically. Briefly, difficulties arise due to the fact that elements in an XML stream may arrive in an order that does not match the order of the predicates that use them in the query, and due to recursive structure in the data, which leads to multiple matchings for an input item. When the query contains the closure axis and multiple predicates, it is even more difficult to keep track of all the information needed for proper buffer management.

The rest of this **paper is organized as follows**. Some preliminaries, including the DOM and SAX models for XML, and the XPath language, are covered in Section 2. Section 3 describes the use of pushdown automata for document filtering and contrasts the task of filtering with the task of querying streams. Section 4 introduces an extended pushdown transducer that provides a convenient method for keeping track of multiple matching paths. A buffered version of these automata is described in Section 5, and a hierarchical arrangement of such automata is described in Section 6. Related work is discussed in Section 7. We describe the architecture and implementation of the XSQ system in Section 8. Section 9 presents our experimental study of XSQ and related systems. We conclude in Section 10.

2. PRELIMINARIES

In this Section, we provide brief descriptions of the DOM and SAX models for parsed XML, and of XPath. We focus on the features that are essential for understanding our methods presented in subsequent sections and do not provide comprehensive descriptions, which may be found elsewhere [Bray et al. 1998; XSL Working

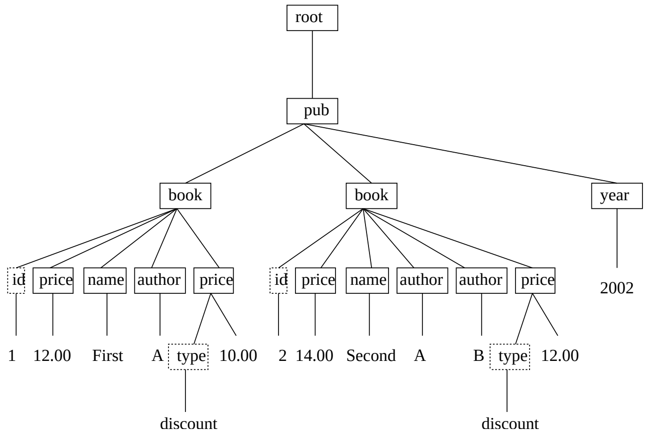


Fig. 3: The DOM tree for the data in Figure 1

Group and the XML Linking Working Group 2000; Megginson et al. 2002; Clark and DeRose 1999].

2.1 Data Model for XML

XML data is usually modeled as an edge-labeled or node-labeled tree [Abiteboul et al. 2000]. In the commonly used *Document Object Model (DOM)* [XSL Working Group and the XML Linking Working Group 2000], an XML document is modeled as a node-labeled tree. Each element in the document is mapped to a subtree in the tree, whose root node is labeled with the tag of the element. Although element E is mapped to a subtree of the DOM tree, it is convenient to refer to the root of this subtree as the *node* E. The subelements of an element E are mapped to subelements of the node E that have *node type* of *element*. The attributes and text contents of element E are also mapped to subelements of node E, but with *node types Attr* and *Text*, respectively. Figure 3 depicts the DOM tree of the XML document in Figure 1. In the figure, the nodes with dotted boxes are Attr nodes and the nodes without boxes are Text nodes.

2.2 Data Model for XML Streams

For streaming data, building a DOM tree in memory is not usually desirable because the data may be unbounded. Further, we may not need all of the DOM tree to process the given query. Therefore, streaming data is better modeled using the SAX (Simple API of XML) model [Megginson et al. 2002]. Parsers based on the SAX Application Programming Interface process an XML document and generate a sequence of SAX events. For each opening and closing tag of an element, the SAX parser generates, respectively, a *begin* and *end* event. The begin event of an element comes with an *attribute list* that encodes the names and values of attributes associated with the element. (Since the XML standard does not allow an element to be associated with multiple attributes with the same name, this list is composed of pairs that are uniquely identified by their first element.) The text contents enclosed by the opening and closing tag result in the SAX parser generating a *text* event. Essentially, the sequence of the SAX events corresponds to a pre-order traversal of the DOM tree of the data in which the attribute nodes are combined with their parents. The SAX events generated by a SAX parser given the data of Figure 1 as input are discussed in Example 3 below.

The SAX API does not explicitly associate events with the depth of the corresponding elements in the document tree. However, this information is easily added to SAX events by maintaining a counter that is incremented and decremented by the handlers for the begin and end events, respectively. Our system makes use of such a counter. For modularity of the code, this counter is stored separately, outside the SAX parser. However, in the descriptions that follow, it is convenient to regard the depth information as part of the SAX event. In more detail, we model the input as a sequence of SAX events, where each event is a quadruple $(\mathbf{tag}, \mathbf{attrs}, \mathbf{type}, \mathbf{depth})$ where:

—**tag** is a string that corresponds the name of the element that generates the SAX event.

—**attrs** is the attribute list of this element. That is, it is a list of elements of the form (a, v) indicating that the element has attribute a with value v. Recall that since elements do not have multiple attributes with the same name, there is at most one pair of the form (a, v) in the attribute list of any element, for all a. We use the notation $e.a$ to refer to the *value* of the a attribute of element e; if e does not have an attribute a, $e.a$ is null.

—**type** is B for a begin event, E for an end event, and T for a text event. Events of type E have an empty attribute list, while events of type T have an attribute list containing the single pair $(text, t)$, indicating that t is the text content of the element.

—**depth** is the depth of the element in the document tree. The attr and text nodes have the same depth as their parent node.

EXAMPLE 3. *Using the notation described above, we list below the first ten events generated by a SAX parser given the input of Figure 1.*

(1) $(root, \phi, B, 0)$*: the begin event of root element.*

(2) $(pub, \phi, B, 1)$*: the begin event of pub element.*

(3) $(name, \{(is, "1")\}, B, 2)$*: the begin event of book element. The name-value list* $\{(id, "1")\}$ *is associated with the event.*

(4) $(price, \phi, B, 3)$*: the begin event of price element.*

(5) $(price, \{(text, "12.00")\}, T, 3)$*: text event of price element. The text "12.00" is associated with the event.*

(6) $(price, \phi, E, 3)$*: the end event of price element.*

(7) $(name, \phi, B, 3)$*: the begin event of name element.*

(8) $(name, \{(text, "First")\}, T, 3)$*: the text event of name element. The text "First" is associated with the event.*

(9) $(name, \phi, E, 3)$*: the end event of name element.*

(10) $(author, \phi, B, 3)$*: the begin event of author element.*

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

Fig. 4: EBNF for core XPath

2.3 XPath

A simplified grammar for XPath is depicted in Figure 4. An XPath query is an expression of 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. Each location step N_i in the location path is in the form `/a::n[p]` where `a` is an **axis**, `n` is a **node test**, and `p` is an optional **predicate** that is specified syntactically using square brackets. A location step matches a node in the document tree. The axis specifies the relation between the previous node and the current node. In the simplified grammar, `/` is shorthand for the `/child::` axis, which selects the children of the current node. Similarly, `//` is shorthand for the `/descendant-or-self::node()/` axis, which selects the current node and its descendants. We use the simplified grammar in our descriptions in this paper. If no axis is specified, the default axis is the child axis. However, if the axis before the first location step is omitted, the default axis is `//`, not the child axis. For example, expression `title/text()` returns the text content of all `title` descendants of the document root.

An element matches a location path if the path from the document root to that element matches the sequence of labels in the location path, and satisfies all predicates. For each matching element, the result of evaluating the output expression on the element is added to the query result. The output expression may specify an attribute of the element, or its text value. It may also use an aggregation function such as sum() and count(). If no output expression is specified in the query, the query returns all the elements in the result set.

The following queries, evaluated on the data of Figure 2 or Figure 1, illustrate some of the key features of XPath.

—`//author/count()`: This query returns the number of `author` elements in the document. The first location step is `//author`, which consists of the closure axis `//`, and the node-test `author`; it does not include a predicate. This location step matches all descendants of the document root that have tag `author`. The output expression, `count()` is applied to all qualifying elements to produce the query result. The result is 2 for the data of Figure 2. We note that this query may also be expressed as `author/count()` because a missing axis in the first location step defaults to closure.

—`//pub[book]//year`: This query returns the `year` elements that have `pub` ancestors that have at least one `book` subelement each. Here, the predicate of the first location step requires the existence of a `book` subelement. We note that both location steps in this query use the closure axis. Further, there is no explicit output function, implying that the elements that match the location path constitute the query result. The result for the data of Figure 2 is `<year>1999</year><year>2002</year>`. Although the `<year>1999</year>` element in line 14 has two `pub` ancestors, both of which satisfy the predicate `[book]`, the `year` element appears in the result only once.

—`/pub[book]/year/text()`: This query returns the text contents of the `year` elements that have a `pub` parent that occurs at the top level. We note that this query does not use the closure axis; thus the depth of elements matching the location path is fixed at 2. Further, the use of the `text()` output function indicates that only the text contents of matching elements are included in the result, without the enclosing tags. The result for the data of Figure 2 is 2002.

—`//pub/book[@id>1]/price[@type="discount"]/text()`: This query returns the text contents of the price elements that have a type attribute with value `discount`. The price element must have a book parent, which in turn has a pub parent. The `id` attribute of the book element must be greater than 1. The result for the data of Figure 1 is `12.00`. Though the `id` attribute and the `discount` attribute are displayed both as strings in the document, the `id` attribute is compared using its numerical value since it is compared to a numerical value. If the value of an attribute cannot be converted to a number successfully, the operation returns false. (Such implicit type coercion semantics provide intuitive results on semistructured data and have been used in other languages, such as Lorel [Abiteboul et al. 1996].)

We note that all the above queries are supported by XSQ, as are other, more complex queries, involving several closures and predicates. The effect of such features on the running time and memory usage of XSQ is discussed in Section 9.5.

3. PUSHDOWN AUTOMATON FOR FILTERING XML STREAMS

In this section, we will first briefly describe pushdown automata and pushdown transducers. Next, we describe the simple relationship between these automata and the XML streams they accept. Systems for *filtering* XML documents make use of this relationship [Diao et al. 2002; Altinel and Franklin 2000]. We discuss why this simple relationship cannot be directly used for the purpose of *querying* XML data. We use the term filtering to refer to the task of finding the documents (from a given set) that satisfy a given predicate and the term querying to mean the task of extracting relevant portions of data from one or more documents, or from streaming XML.

A **pushdown automaton (PDA)** [Hopcraft and Ullman 1979] is a finite-state automaton that operates on both an input tape and a stack. It has a finite set of states, including one start state and one or more accepting states, a set of input symbols, and a set of stack symbols. At each step, a PDA consumes a symbol from the input. A PDA's transition function determines, as a function of the input symbol, the current state, and the stack, the next state and the operation performed on the stack. A PDA is said to accept an input string if, when the input is consumed, it is in one of its accepting states.

A **pushdown transducer (PDT)** is a PDA with actions defined along the transition arcs on the automaton. The transition function of a PDT specifies an optional output operation as a function of the current state, input symbol, and stack (in addition to specifying the next state and the stack operation). When a

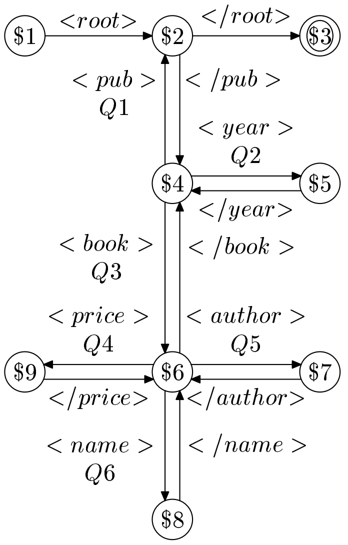


Fig. 5: A simple PDA for the XML stream in Figure 1

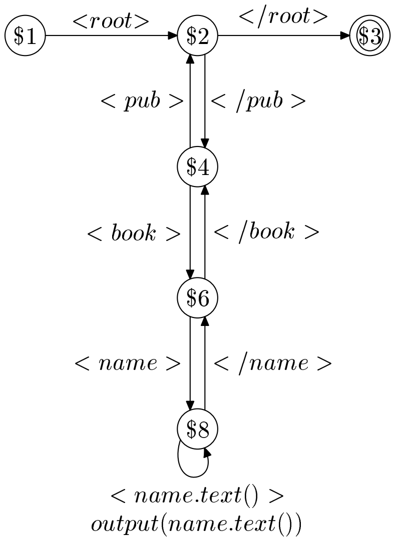


Fig. 6: A simple PDT

state transition occurs, such an operation results in some items being appended to the output of the PDT. A PDT is defined to accept an input string in a manner analogous to a PDA. However, a more common use of a PDT is to transform data by using a state transition function with output operations.

It is easy to devise a PDA that accepts XML documents that have a specified structure. One may begin with an automaton that intuitively traces the desired document structure. For example, Figure 5 depicts an automaton that outlines documents with structure similar to that of the document of Figure 1. The start state is $1 and the only accepting state is $3. This automaton accepts a document that has, at the top level, a pub element that has a year element as child and that also has a book subelement that, in turn, has subelements with labels price, name, and author. It also accepts a much simpler document that only has a pub element without any subelements. However, it will not accept a document with a book element at the top level. In more detail, for each of the SAX events generated for the XML stream in Figure 1, the PDA in Figure 5 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 compares the tag of the current element and the tag on the top of the stack. If these two tags match, it pops the tag off the stack. Otherwise the XML stream is not well-formed and an error is flagged. For acceptance, when the PDA has processed all the events generated from the stream, it should be in the final state *$3*. It implies that the stack should be empty since the stack operations have a bijective mapping to the state transitions (e.g., pushing `pub` onto the stack corresponds to the transition on the `<pub>` event). In the following discussion, we assume the XML stream is always well-formed.

The skeleton PDA described above can be adapted to filtering XML documents as follows: Suppose we wish to find all the documents that contain at least one element matching the pattern `//pub//book//name`. We may use the simple automaton suggested by Figure 5. Here and in what follows we shall assume that if there is no arc (transition) matching an input symbol then the automaton remains in the current state. Whenever this simple automaton transitions from state *\$6* to state *\$8*, we know that the current XML document contains an element that satisfies the filter expression `//pub//book//name` and we can return the document to the user. The PDA in Figure 5 can be further used for simultaneously filtering an XML stream using multiple queries. When a transition marked with Q_i is triggered by an event in the incoming XML file, the PDA reports to the user that the document satisfies query Q_i. For example, Q_2 is `//pub//year`, Q_6 is `//pub//book//name`, and the other queries can be inferred similarly. Of course, in many real applications, the queries do not have such convenient similarities. Combining such queries then requires more sophisticated techniques [Altinel and Franklin 2000; Diao et al. 2002; Chan et al. 2002].

The PDA can also be modified into a PDT that answers simple queries. For example, if we remove the branches of the PDA in Figure 5 and put an output operation on a self-transition from state *\$8*, we get the PDT depicted in Figure 6. This PDT evaluates the XPath query `//pub//book//name/text()`.

However, it is not straightforward to extend this simple idea for building PDAs and PDTs to more general XPath queries. The main reason is that the PDA cannot buffer previously processed data. (The stack of the PDA is used exclusively to ensure proper nesting of begin and end tags.) Such buffering is required for answering XPath queries that have predicates because the data required for evaluating these predicates for a given XML element (that satisfies the rest of the XPath query) may appear at various points in a stream. In particular, the data required to evaluate a predicate for an element may appear long after (much farther downstream from) the element itself. A naive solution is to record the current results for every predicate, and mark every item in the buffer with flags that indicate which predicates have been satisfied and which have not yet been satisfied. Every time we evaluate a predicate, such a method would need to check if some items are affected by its result, resulting in poor performance. Further, as queries get more complex, such a method would soon become too unwieldy as it uses ad-hoc methods to keep track of all the necessary information. In Example 1, if the first `year` element has satisfied the predicate `[year > 2000]`, the other `year` element of the same `pub` element should not be tested. If there are closures in the query and the data is recursive, such flags need to be set on a per-matching basis, not just for each item. These and other difficulties are explored in more detail when we describe our methods in the following sections.

4. EXTENDED PUSHDOWN TRANSDUCERS

The traditional PDT and PDA are not suitable for streaming XML processing since the states in the PDT and PDA do not encode enough information about the patterns they match. For example, when the PDT of Figure 6 is in state *\$6*, we know only that the current element satisfies the pattern `//pub//book`. We do not know the depths of the `pub` and `book` elements that match this pattern. Further, when there are multiple matches between the pattern and the data, these matchings cannot be distinguished. However, recall from Example 2 that if the path from the

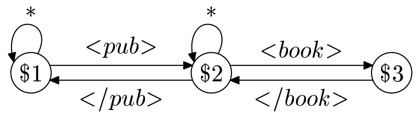


Fig. 7: A PDT with Kleene star

document root to an item matches a path in a query in multiple ways, we must consider each matching; the item belongs in the query results if any one of these matchings satisfies the predicates in the query.

Another problem with PDTs is that the semantics of closures (Kleene closure, $\star$) in these automata do not map easily to the semantics of closures in XPath (descendant axis, `//`). For example, the query `//pub//book` may suggest the PDT depicted in 7, with the transition from \$2 to \$3 generating output for the result. However, in addition to matching book elements that are descendants of pub elements, such a PDT also produces output for book elements that occur anywhere after a pub element in document order. For example, it erroneously produces output for the book element in the XML input `<pub>p1</pub><book>b1</book>`.

To address the above problems, we define an **extended PDT (XPDT)** by augmenting a PDT with a stack (separate from the main stack) called the **depth stack** and modifying the transition function to take this stack into account. We also permit transitions to be conditional on the evaluation of a predicate adorning the transition. We show that the XPDT is a useful extension of the PDT that permits convenient processing of depth information in XML streams.

An XPDT is specified by means of a 7-tuple $(\Sigma, \Gamma, Q, P, \delta, F, s_0)$, indicating the input alphabet Σ, the stack alphabet Γ, the set of states Q, the set of predicates P, the transition function δ, the set of operations F, and the start state $s_0 \in Q$. We describe these components in more detail below.

—The input alphabet Σ may be infinite and is composed of **input symbols**, which are SAX events of the form (*tag*, *attrs*, *type*, *depth*). (Recall the SAX model from Section 2.2.)

—The stack alphabet Γ consists of **stack symbols** of the form of (*tag*, *depth*). On encountering an input symbol of type B (begin element; see Section 2.2), the tag and depth of the element is pushed on to the stack. This stack item is removed from stack when the corresponding end element event (type E) is encountered. The stack is subject to the standard operations: *push*(x), which pushes x onto the stack; *pop*(), which removes and returns the element at the top of the stack; and *peek*(), which returns the top element without displacing it.

—**States** in Q are of the form (i, d), where i, called the **base ID**, is a unique identifier and d, called the **depth stack**, is a stack of integers. We may think of i as the traditional state ID for an automaton. However, it forms only one dimension of the two dimensional state identifiers in XPDTs. The second dimension, the depth stack, is used to distinguish between different paths that lead to states with the same base ID, corresponding to different matchings between the query and an input item. This two-dimensional naming scheme for states is convenient for describing the operation of the XPDT. (See Example 4 below.)

—Each **predicate** in the set P is of the form of *attr op lit* and compares the value of the named attribute with the provided literal using the operator *op* (chosen from the list in Figure 4, with the usual semantics). The predicate associated with a transition is evaluated on input symbols that trigger that transition; the transition is taken if and only if the predicate evaluates to true.

—The set F contains the **operations** that are associated with the transitions. If an operation f is associated with a transition, the operation f will be executed when the transition happens. Choices for the operation include the `null` operation that does nothing, printing the current input symbol, and displaying a predefined message based on the content in the stack. Operations form the interface for more complex transducers to extend the function of an XPDT. For example, the BPDT described in Section 5.1, defines a set of buffer operations that operate on the augmented buffer.

—The **transition function** δ is a mapping from $Q \times \Sigma \times \Gamma^*$ to $\mathcal{P}(P \times Q \times \Gamma^* \times F)$, where $\mathcal{P}(X)$ denotes the power set of X and Γ^* denotes the stack as a string over the stack alphabet. We may think of δ as defining, for each state in Q, a set of outgoing transitions based on the state of the stack and the input symbol. Each transition is described by a predicate, a destination state, a new stack state, and an operation from F. The semantics of the transition function are explained further below.

—The start state q_0 is simply the initial state of the automaton. The automaton commences execution by evaluating the transition function for this state, with an empty stack, and with the first symbol in the input.

In the following discussion, we use the term **current element** to refer to the input element that generated the SAX event currently being processed by the XPDT. The depth stack is used to record the run-time information of which elements in the input lead to the current state. The *begin* events of all ancestors of the current element are processed before the current element; however, not all of them result in a state change during this process. The XPDT only needs to record the ancestors that lead to state changes. In the application of evaluating XPath queries, only these ancestors take part in the matchings between the XPath query and the result. For example, suppose we wish to evaluate the query `//book//pub//name` on the input listed in Figure 2. Although the `name` element in line 11 has five ancestors, in lines 1, 2, 7, 9, and 10, we need to record only the two ancestors in lines 7 and line 9. Although the other three elements may match a single step in the query (and may be involved in the matching for other `name` elements), we do not need the results of predicate evaluations at these elements when we process the `name` element in line 11.

The depth stack contains the integer i if only if the current element's ancestor at depth i produced a state change in the sequence of transitions leading from the start state to the current state. The depth stack essentially records the states of the main stack in the states leading to the current state. We define the **depth of a state** (i, d) to be the integer at the top of the stack d. That is, the depth of (i, d) is $d.peek()$. In addition to the standard stack operations (push, pop, and peek), we define the operation $remove(k)$ on depth stacks to result in the removal of the top k elements of the stack. We say two depth stacks are equal if they have the same number of elements and the corresponding elements are equal.

We now describe the semantics of the transition function $\delta : Q \times \Sigma \times \Gamma^* \rightarrow \mathcal{P}(P \times Q \times \Gamma^* \times F)$ in more detail. The XPDT maintains a set of currently active states Ψ, which initially is $\{(q_0, ())\}$, where we use $()$ to denote an empty depth stack. When the XPDT reads the input symbol e, it computes, for every state $(n, d) \in \Psi$, the set of transitions $\delta((n, d), e, K)$, where K denotes the stack. For each transition $(p, (n', d'), K', f)$ in this set, the XPDT evaluates the predicate p. If p evaluates to true, the XPDT replaces (n, d) with (n', d') in Ψ and updates the stack from K to K'. Further, the operation f associated with the transition is performed. If p evaluates to false for all transitions, (n, d) remains in Ψ and no further action is taken. A special case is when $\delta((n, d), e, K)$ is empty, i.e., when there are no transitions from (n, d) on e. We contrast this convention for undefined transitions with that used in many traditional automata, which report an error if such a situation occurs. This convention allows us to simplify the definition of automata for XPath queries. For example, given a query `/A/B`, the automaton need only consider the `B` subelements of the `A` elements. All the other subelements of `A` can be ignored. In the state corresponding to `A` in the automaton, we may achieve this behavior by defining only one transition, on `B`.

The above definition of an XPDT permits arbitrary transitions and arbitrary modifications to the depth stack at each transition. However, we focus our attention on the XPDTs used by XSQ to process XPath queries. In such XPDTs, the transitions may be classified as described below and the depth stack is modified in only a few different ways. In the following description of transitions, we consider an input symbol e, source state $q = (n, d)$ and target state $q' = (n', d')$.

Self-closure transition:. Such a transition is taken for an input symbol e of type B (begin element) that has depth greater than the depth of the current state. The source state of the transition remains in the set of current states, and no new state is added. That is, for a self-closure transition, if $e.type = B$ and $e.depth > d.peek()$ then $q' = q$. In state transition diagrams, self-closure transitions are identified using the symbol `//` next to the arrows denoting the transitions.

Closure transition:. Such a transition is also taken for an input symbol e of type B (begin element) with depth greater than the depth of the current state. The source state of the transition remains in the set of current states. However, unlike the case of the self-closure transition, new states are added to the set of current states. The depth stacks of the new states are obtained by pushing the depth of the event onto a copy of the depth stack of the current state. That is, for a closure transition, if $e.type = B$ and $e.depth > d.peek()$ then $d' = d.push(e.depth)$. In state transition diagrams, closure transitions are identified using the symbol `=` on the arrows denoting the transitions.

Regular transition:. Such a transition is taken for an input symbol e if

$$e.depth = \begin{cases} d.peek() + 1, & \text{when } e.type = B \\ d.peek(), & \text{when } e.type = T \text{ or } e.type = E \end{cases}$$

The current state q is removed from the set of active states. The depth stack of

#	event	active states	
		$\{(\$1,\ \phi)\}$	
1	$(pub, \phi, B, 1)$	$\{(\$1,\ \phi),\ (\$2,(1))\}$	
2	$(pub, \phi, B, 2)$	$\{(\$1,\ \phi),\ (\$2,(1)),\ (\$2,(2))\}$	
3	$(book, \phi, B, 3)$	$\{(\$1,\ \phi),\ (\$2,(1)),\ (\$2,(2)),\ (\$3,(1,3)),\ (\$3,(2,3))\}$	
4	$(name, \phi, B, 4)$	$\{(\$1,\ \phi),\ (\$2,(1)),\ (\$2,(2)),\ (\$4,(1,3,4)),\ (\$4,(2,3,4))\}$	f_0 executed
5	$(name, \{(text(), \mathtt{A})\}, T, 4)$,	$\{(\$1,\ \phi),\ (\$2,(1)),\ (\$2,(2)),\ (\$4,(1,3,4)),\ (\$4,(2,3,4))\}$	no transition
6	$(name, \phi, E, 4)$	$\{(\$1,\ \phi),\ (\$2,(1)),\ (\$2,(2)),\ (\$3,(1,3)),\ (\$3,(2,3))\}$	
7	$(book, \phi, E, 3)$	$\{(\$1,\ \phi),\ (\$2,(1)),\ (\$2,(2))\}$	
8	$(pub, \phi, B, 2)$	$\{(\$1,\ \phi),\ (\$2,(1))\}$	
9	$(pub, \phi, B, 1)$	$\{(\$1,\ \phi)\}$	

Fig. 8: The XPDT of Example 4 in action

each state q' added to the set of active states is obtained as follows:

$$d' = \begin{cases} d.push(e.depth) & \text{when } e.type = B \\ d & \text{when } e.type = T \\ d.pop(), & \text{when } e.type = E \end{cases}$$

In state transition diagrams, regular transitions are represented by arrows with no special markings.

Catch-all transition:. Such a transition is taken for an input symbol e of any type if the depth of e is greater than the depth of the current state q or if e is of type T and has depth equal to the depth of q. The state q remains in the set of current states, i.e., $q' = q$. In state transition diagrams, catch-all transitions are identified using the symbol $\bar{*}$ next to the arrows representing the transitions.

Given the above rules relating the depth stacks of the source and destination states of a transition in an XPDT, we do not need to specify the depth stacks explicitly in the transition function. In particular, we can determine the operations on the depth stacks by noting the symbols adorning the arrows (//, =, $\bar{*}$, or none) in a state transition diagram.

EXAMPLE 4. *Consider an XPDT* $(\Sigma, \Gamma, Q, P, \delta, F, s_0)$ *where* $Q = \{\$1, \$2, \$3, \$4\}$, $P = \phi$, *and* $F = \{f_0\}$, *where* f_0 *is an operation that writes the string* `matched` *to the output. The start state* s_0 *is $1, and the transition function is summarized by the state transition diagram depicted in Figure 9. The diagram uses* `<pub>` *and* `</pub>` *to denote, respectively, the begin and end events of* `pub` *elements. The XPDT is designed to produce one* `matched` *string in the output for each element matching the query* `//pub//book/name`.

Figure 8 summarizes the actions of the XPDT on the following input:
`<pub><pub><book><name>A</name></book></pub></pub>`
In step 3, a transition on `book` *is taken from both ($2,(1)) and ($2,(2)) because the transition from $2 to $3 is a closure transition, as indicated by the = on the arrow. These two states also remain active because of the self-closure transition in state $2, as indicated by the // on the arrow from $2 to itself. The transition taken in step 4,* $\$3 \to \$4$, *results in the execution of the print operation. We note that we may also put this operation on the transition* $\$4 \to \$3$ *instead of on* $\$3 \to \$4$ *because we assume that the input is well-formed. In step 5 there is no transition defined on the input event and the set of active states is unchanged. More precisely,*

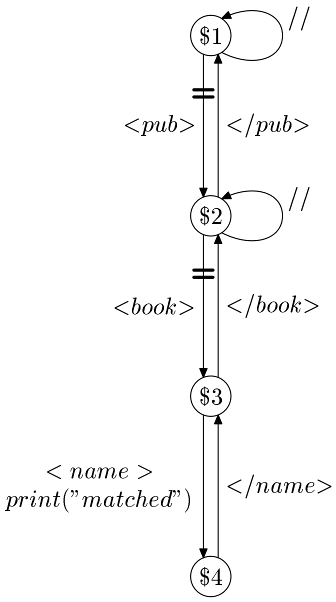


Fig. 9: A simple XPDT

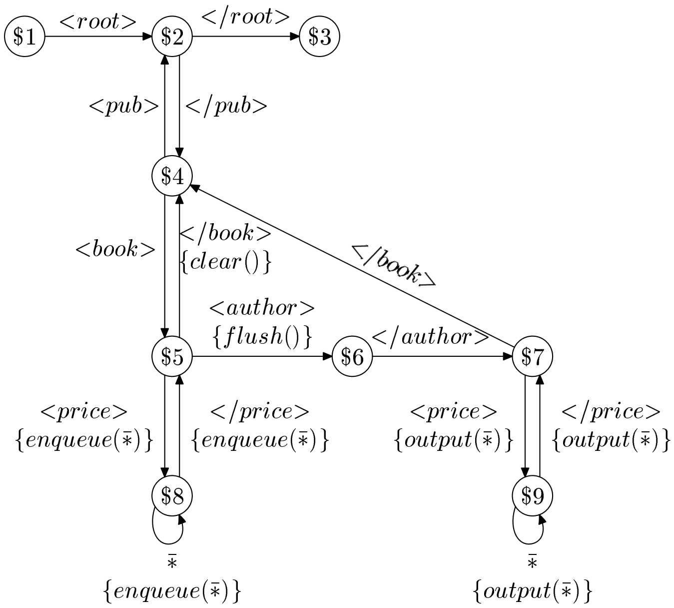


Fig. 10: A simple BPDT for query /pub/book[author]/price

$\delta(q, (name, \{(text(), \mathtt{A})\}, \Gamma^*) = \emptyset$ for all states q active in step 5. Finally, a simple change to this XPDT yields an XPDT that produces the text contents of matching name elements as output (instead of the string `matched`). We remove the operation from the transition $\$3 \rightarrow \$4$ and add a transition $\$4 \rightarrow \$4$ on the text event, with an operation that outputs the value of the `text()` attribute of the event.

The above example suggests how an XPDT is used to answer simple XPath queries. It also illustrates that we do not need to specify the depth stacks explicitly in the transition function. They are determined at runtime based on the type of the transition taken, using the rules described earlier. However, the reader may notice that there is a problem with the XPDT used in this example. In step 4, the operation f_0 is executed twice: once for of the transition $\$3 \rightarrow \$4$ out of $(\$3, (1, 3))$ and once for the same transition out $(\$3, (2, 3))$. Thus, the string `matched` is printed twice although there is only one matching name element in the data. This problem is caused by the two ways in which the name element matches the query. The first matching uses the outermost pub element of the input while the second uses the inner pub element. In order to fix this problem, as well as to enable evaluation of predicates that require buffering, the next section introduces a buffered version of this automaton.

5. BUFFERED PUSHDOWN TRANSDUCERS

Recall our discussion in Example 1, which indicated that a streaming XPath processor must buffer data items whose result membership cannot be decided until additional data arrives in the stream. Since the XPDTs introduced in the previous section do not have a buffer, they cannot answer XPath queries with predicates, which require buffering. In this section, we augment the XPDT with a buffer and a set of buffer operations. The resulting automaton, which we call a buffered pushdown transducer (BPDT), is used to encode a single location step of an XPath query. A collection of such automata is then used to encode the entire query. We introduce the buffer and the operations used to manipulate it in Section 5.1. In Section 5.2, we describe our method for mapping XPath location steps to BPDTs. We discuss the combining of BPDTs in Section 5.3.

5.1 BPDTs and Buffer Operations

A **Buffered Pushdown Transducer BPDT** is an 8-tuple $(\Sigma, \Gamma, Q, P, \delta, F_B, \Omega, s_0)$, where $\Sigma, \Gamma, Q, P, \delta$, and s_0 are defined as in the definition of the XPDT (Section 4) . The buffer alphabet Ω specifies the items in the buffer, which is organized as a queue. The set F_B is composed of the buffer operations described below.

The buffer operation **enqueue(a)** puts the value of *feature* a of the current input event at the end of the queue. There are three kinds of features that may be enqueued using this operation. First, a may be the name of an XML attribute, in which case the value of the named attribute of the current event is enqueued. Second, a may be the literal `text()`, in which case the text content of the current event is enqueued. Finally, a may be the *catch-all symbol* $\bar{*}$, in which case the serialized (string) representation of the input event is enqueued, including all its attributes. For example, for the begin event $(book, \{(id, "1")\}, B, 1)$, the operation $enqueue(\bar{*})$ enqueues the string `<book id="1">`. Other operations on the buffer include **clear()**, which clears the contents in the queue, and **flush()**, which flushes the contents of the queue to the output. The operation **output(a)** emits the value of attribute a directly as the output. Although it does not operate on the buffer, we include it in F_B for ease of presentation. In state transition diagrams, we indicate the buffer operation associated with a transition by using the operation as a label on the arrow representing the transition. The following example illustrates how a BPDT may use the buffer to answer an XPath query that requires buffering.

EXAMPLE 5. *The BPDT depicted in Figure 11 performs a streaming evaluation of the query:* `/pub/book[author]/price`*. It uses the catch-all symbol* $\bar{*}$ *to indicate that all subelements of the price element should be in the result. Let us consider the first few actions of this BPDT on the XML stream of Figure 1. After processing the elements in lines 1 through 3, the BPDT is in state \$5. It then enqueues the item in line 4 into the buffer. We note that it will return from state \$8 to state \$5 when it encounters the end event of the price element since the catch-all transition accepts only events with depth larger than the depth of the current state (Section 4) while the end event of the price element has the same depth as the current state. When the BPDT encounters the begin event of the author element in line 6, it flushes the items to the output and goes to state \$6 (and state \$7 at the end event of the author). The BPDT encounters the next price element in line 7 and this time it emits the element directly to output.*

5.2 Templates for Single Location Steps

The following example describes the intuition behind our mapping from XPath location steps to BPDTs.

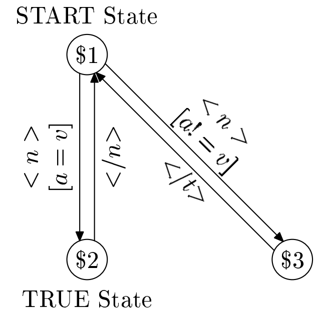


Fig. 11: Template BPDT for: /n[@a = v]

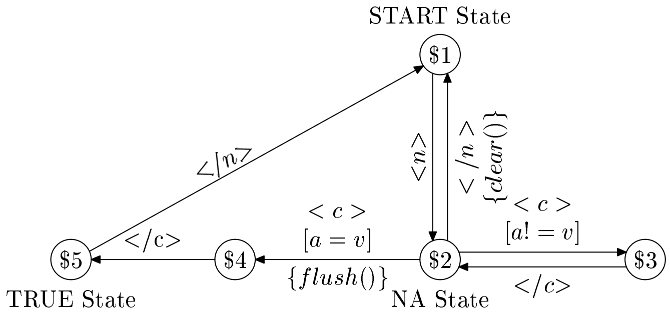


Fig. 12: Template BPDT for: /n[c@a = v]

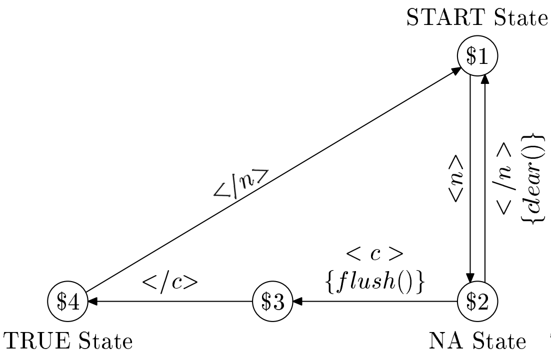


Fig. 13: Template BPDT for: /n[c=v]

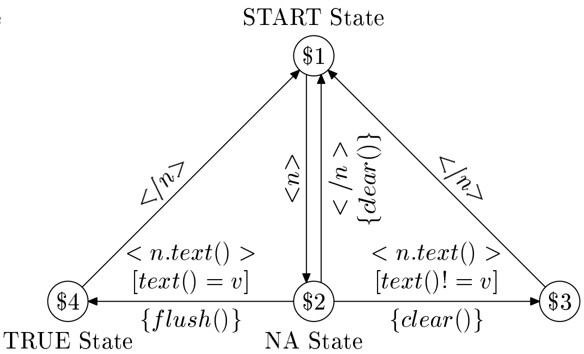


Fig. 14: Template BPDT for: /n[text() = v]

EXAMPLE 6. *Consider the XPath query `/book[author]/text()`, which consists of a single location step `/book/[author]`. Given the semantics of this query, a BPDT for this query must operate as follows on streaming data: If it encounters a `<book>` event followed by an `<author>` event, it must record the fact that this `book` element satisfies the `[author]` predicate, so that it can output the text contents of the element immediately when they are encountered later. On the other hand, if the text contents of a book element are encountered before a `<author>` event, then the contents must be buffered until either a `<author>` event is encountered, in which case the buffer is flushed to the output, or a `</book>` event is encountered, indicating that the book element has no author subelement, in which case the buffer is cleared. These observations suggest mapping this location step to a BPDT similar to the one depicted in Figure 14, substituting `<book>` for `<n>` and `<author>` for `<c>`. In order to extend this BPDT for the location step `/book/[author]` to one that answers the query `/book[author]/text()`, we add transitions for the `text()` event of book elements to states $2 and $4. The buffer operation on the transition out of $2 enqueues the text content while the buffer operation on the transition out of $4 sends the content directly to the output.*

As suggested by Example 6, there are three special states in the BPDT corresponding to a location step: The START state is the entry point to the BPDT. The TRUE state indicates the predicate of this location step has evaluated to true, while the NA state indicates that the predicate has not yet been satisfied. As discussed later (Section 6), these states are used to connect the BPDTs for individual lo-

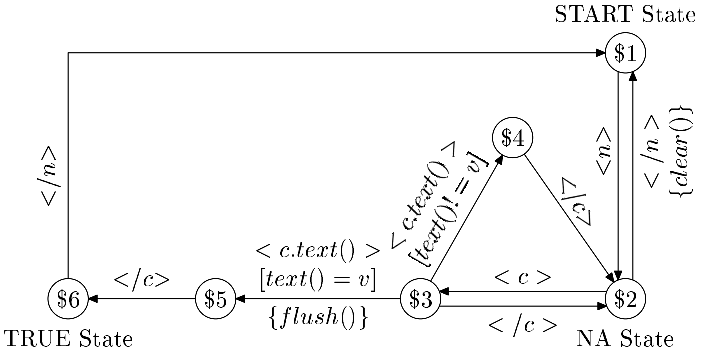


Fig. 15: Template BPDT for: /n[c=v]

cation steps to form an automaton that answers a multi-step XPath query. The reader may note that the satisfaction of a predicate has limited scope; for instance, in Example 6, the presence of an author element satisfies the `[author]` predicate only for the book element that is its parent. The predicate must be evaluated separately for other book elements. We describe our methods for such bookkeeping in Sections 5.3 and 6.

For the purpose of mapping XPath location steps that contain predicates to BPDTs, we classify the location steps into the five categories described below. Recall, from Section 4, that our transition diagrams do not explicitly mention the manipulations of the depth stacks, which are governed by the rules described earlier.

—Location steps that test whether the current element has a specified attribute, or whether the attribute satisfies some condition. These steps have the general form `/n[@a]` and `/n[@a op v]`, where `n` is an element name, `a` is an attribute name, `op` is one of the comparison operators (Figure 4), and `v` is a literal. For example, `/book[@id]` denotes a book element that has an id attribute, while `/book[@id <= 10]` requires further that the id attribute have a value no greater than 10. Such steps are mapped to a BPDT using the template suggested by Figure 12. The figure illustrates the case for `/n[@a = v]`; for the `/n[@a]` case, the test of the attribute value is replaced by a test for the existence of the attribute.
When evaluating the comparison for the attribute value with a literal, the semantics of XPath require that if the literal `v` is a number, the value of the contents (here is the attribute) should be converted to numerical value and the numerical values are compared. If the comparison is failed, the predicate returns false.

—Location steps that test whether the current element contains a specified string, or whether the value of the text satisfies some condition. These steps take the general form of `/n[text() op v]`. For example, `/year[text() = 2000]` denotes a year element whose text content is equal to 2000. Figure 15 depicts the template for the BPDTs that process location step of the form `/n[text()=v]`.
A subtle point here is that we have implicitly assumed that an element may generate at most one text event. However, when the text content of an element is interspersed with subelements, a SAX parser generates multiple text events, one for each contiguous text segment. For example, the XML fragment <review> The<color>yellow</color>road...</review> generates two text events, one

for “`The`” and the other for ”`road...`” XPath semantics require that this fragment match the location step `/review[text() contains "The road"]`. However, if the two text events are treated separately, such a match will be missed. XSQ therefore aggregates multiple text events of this kind into a single event that is issued just before the end tag of the element to which the text belongs (just before `</review>` in our example). For ease of exposition, we will henceforth assume that text events are in such an aggregated form.

—Location steps that test whether the current element has a specified type of subelement. These steps are of the form `/n[c]`. For example, `/book[author]` matches a book element that has at least one author subelement. Figure 14 depicts the template for the BPDTs that process location step `/n[c]`.

One may note that there is only one transition out of state \$3 and consider the possibilty of merging \$4 with \$3, with a transition on `</c>` from \$3 to itself and a transition on `</n>` from \$3 to \$1. However, BPDT generated using this template cannot be combined with other BPDTs to answer XPath queries that have several location steps. Consider the BPDT in Figure 11. If we merge state \$6 and \$7 with a transition on `</author>` from \$6 to itself, the state \$6 will accept not only the `price` subelements of the `book` element, but also the the `price` subelements of the `pub` element (while the query asks for only the former).

—Location steps that test whether a specified subelement of the current element contains an attribute, or whether the value of such an attribute satisfies a predicate. These steps are of the form `/n[c@a]` and `/n[c@a op v]`, respectively. For example, `/pub[book@id <= 10]`) denotes a pub element that has a book subelement whose id attribute is less than or equal to 10. The BPDT template of the location step `/n[c@a op v]` is depicted in Figure 13.

—Location steps that test whether the value of a specified subelement of the current element satisfies a given predicate. These steps are of the form `/n[c op v]`. For example, `/book[year <= 2000]` matches a book element that has a year subelement whose value is less than or equal to 2000. Figure 16 depicts the template for the BPDTs that process location step `/n[c op v]`. This template is similar to the template depicted in Figure 14, but includes transitions to process the text event.

We note that the above templates encode the existential semantics of XPath predicates: An element matching the name in a location step qualifies for matching the location step if there is at least one subelement data that satisfies the predicate. The element fails to qualify only if all its subelements fail to satisfy the predicate.

Although the above templates provide a simple method for mapping location steps to automata, using them to answer a given query requires some manipulation of the buffer operations. For example, Figure 17 depicts a BPDT generated for a single step XPath query that returns an entire element: `/pub/[year > 2000]`. The need to return an entire element (not just its text content) requires the use of the *catch-all transition* in the BPDT. Such modifications for generating BPDTs for answering single-step XPath queries from the templates (the template in Figure 16 in our example) are straightforward.

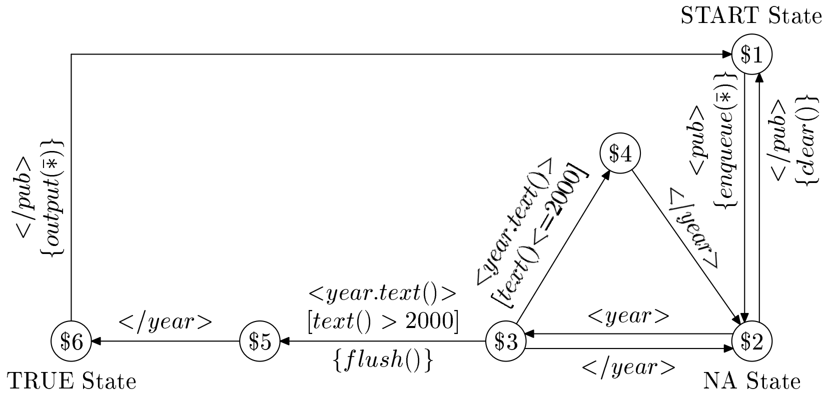


Fig. 16: BPDT for query `/pub[year>2000]`

5.3 Connecting the BPDTs

We now discuss methods to connect BPDTs for the location steps of an XPath query into a larger BPDT that answers the complete query. When we connect BPDTs for individual location steps, we must maintain the structural relations among the location steps. For example, for query `/book[author]/price/text()`, we must ensure that the BPDT generated for the second location step outputs the text content of only those `price` elements that have a `book` element satisfying the `[author]` predicate as parent. This requirement is easily satisfied following the scheme discussed in Example 5: We merge the START state of the second BPDT with states that are right after the begin event of the `book` element or right before the end event of the `book` element, ensuring that any `price` element considered by the second BPDT is a child of a `book` element. Figure 18 illustrates this idea. We note that we need multiple copies of the second BPDT, which is a BPDT of the simplest kind, having no predicate. These copies differ in their buffer operations. We defer to Section 6 the description of our method for modifying buffer operations in the BPDT templates to ensure proper operation of the automaton composed of multiple BPDTs.

Recall that XSQ is designed to buffer only those items whose result membership cannot be immediately determined (i.e., those that any streaming XPath processor must buffer). For example, for the query `/book[author]/price[@type="discount"]/text()`, the operations in the BPDT generated for the second location step should output all the text contents directly if the predicate in this BPDT has been satisfied and the predicate in the first location step is known to be true. If the result of the first predicate is currently unknown, text contents should be enqueued if the type attribute is named `discount`. By applying the idea of generating multiple copies of BPDTs for the second location step and merging the START states of these copies with the appropriate states in the BPDT of the first location step, we arrive at the BPDT depicted in Figure 19. We note that although we generate the BPDTs for the second location step by instantiating the same template from Section 5.2, the operations on the transitions differ between the instances, depending on state in the BPDT for the first location step to which they connect. For queries with three or more location steps, determining the appropriate buffer

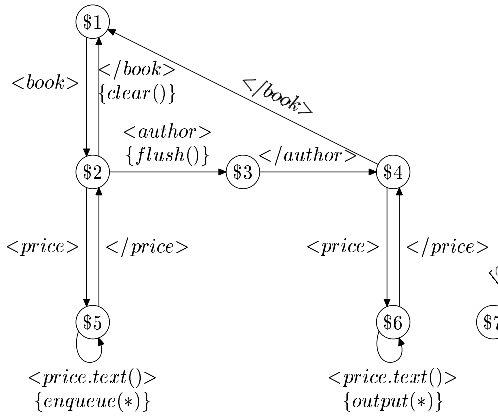


Fig. 17: BPDT for query `/book[author]/price/text()`

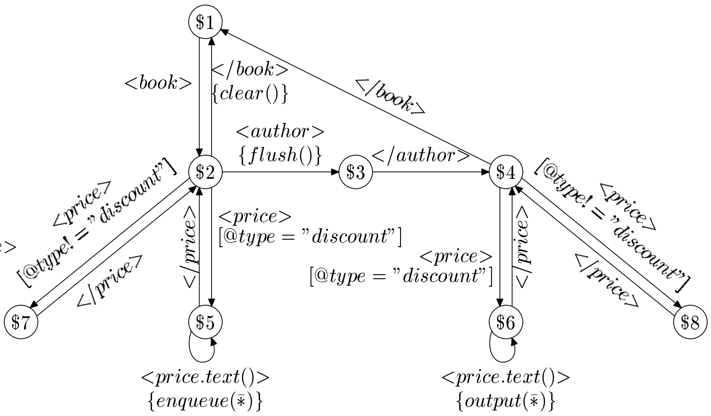


Fig. 18: BPDT for query `/book[author]/price[@type="discount"]/text()`

operations for each instantiation of a template is likely to be complicated because the operations may be affected by results of predicates both before and after the current location step. Although such a scheme can be worked out, we prefer to use the simpler scheme described in Section 6 because that scheme is needed to address the problem we describe next.

When BPDTs are interconnected, we need to ensure that when a predicate is evaluated, all the contents in the buffers that are affected by the result of this predicate are processed right away. If the result is true, items in the buffer that are waiting only for this result (and not the result of some other predicate as well) should be sent to output. If the result is false, all items in the buffer that are waiting for this result (and perhaps other results) should be removed. An additional complication occurs when there there are multiple matchings between the data and the query, such as those described in Example 2. In this case, we must ensure that we remove from the buffer only those items for which there is no matching that satisfies all the predicates. The simple buffer organization used by BPDTs makes it impossible to differentiate between buffer items in this manner. For example, as we described in Example 1, when we evaluate the query `/pub[year > 2000]/book[price < 11]/author` over the stream in Figure 1, there are three `author` items in the buffer when we encounter the end event of the second `book` element. At this time, the predicate of the second location step, `[price < 11]`, of the second `book` element evaluates to false. Therefore, the two authors of the second book should be removed from the buffer. However, the BPDT cannot distinguish between the author of the first book and the authors of the second book. We may address this problem by extending the buffer alphabet to include flags that allow us to distinguish between different groups of items. Another alternative, and one used by XSQ and described next, is to organize buffers hierarchically and define buffer operations that transfer items from a buffer to its parent.

6. BUILDING A HIERARCHICAL PUSHDOWN TRANSDUCER

In this section, we put together the ideas from earlier sections to describe the complete method used by XSQ to build an automaton to answer an XPath query. This automaton is obtained by aggregating the BPDTs generated for each location step in the query. As discussed in Section 5.3, we may need to use multiple instances of the BPDT corresponding to a location step, with each instance using a different set of buffer operations. We first describe how BPDTs are connected in a hierarchical manner so that the buffer operations in each BPDT are determined solely by the location step from which the BPDT is generated and the position of the BPDT in the structure, but not by the runtime information of the results of the predicates. We then extend the set of BPDT buffer operations to support communication between the BPDTs. We refer to the resulting network of BPDTs as a **Hierarchical Pushdown Transducer (HPDT)**.

6.1 A hierarchical structure

As we described in Section 5.3, a single buffer does not enable us to properly process buffer items that differ in the sets of predicates they must satisfy in order to qualify as query results. To address this problem, we introduce a separate buffer for each BPDT in an HPDT. We also introduce an **upload(bpdt)** function that transfers all the items from the buffer of the calling BPDT to the buffer of the BPDT specified as the argument. (The details are described below.)

Recall, from Section 4, that although states are identified using a two-dimensional identifier (i, d) where i is a base identifier and d is a stack of integers (the depth stack), the rules governing the depth stacks during transitions permit us to specify a transition function using only the base identifiers of states. The depth stacks are manipulated at run-time based on the rules in Section 4. In this section, our focus is on the compile-time construction of an HPDT. Therefore, we will identify states using only their base identifiers.

Recall, from Section 5.2, that each BPDT template has a single START state, a single TRUE state, and an optional NA state. Given an XPath query with n location steps, we generate instances of BPDTs using these templates, and connect the instances as follows: For the l'th location step, we generate 2^l BPDTs from the templates described in Section 5.2. The buffer operations are initially set to those in the templates. The BPDTs generated for the l'th location step are assigned identifiers of the form (l, k), where $k \in [0, 2^l)$. We use $bpdt(l, k)$ to denote the BPDT with identifier (l, k). After we generate BPDTs for all the location steps, we connect the BPDTs in a layered fashion. Each $bpdt(l, k)$ $(l < n)$ has two children: a right child $bpdt(l + 1, 2k)$ whose START state is the NA state of $bpdt(l, k)$ and a left child $bpdt(l + 1, 2k + 1)$ whose START state is the TRUE state of $bpdt(l, k)$. It is possible that the $bpdt(l, k)$ does not have an NA state; in this case, $bpdt(l + 1, 2k)$ is set to *null*. A null BPDT does not exist in the structure, but it is counted when we compute the sequence numbers of BPDTs. In this layered structure, we refer to the BPDTs generated for the l'th location step as **the l'th layer**. We maintain a separate buffer for each BPDT and use $B(l, k)$ to denote the buffer of $bpdt(l, k)$. The zeroth location step refers to the leftmost / in an XPath query and it matches the document root. The BPDT generated for the zeroth location step is depicted in Figure 20. Null BPDTs resulting from missing NA states result in pruning in the HPDT. For example, since the BPDT for the zeroth location step does not have an NA state, there is no need to generate $bpdt(1, 0)$ for the first location step; similarly, its descendants, $bpdt(l, k)$ for $l \in [2, n]$ and $k \in [0, 2^{l-1}]$ are not generated. An

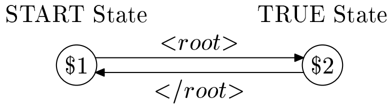


Fig. 19: Template for the root BPDT

example of an HPDT is depicted in Figure 21. Each box in the figure encloses the states of a BPDT (with the exception of BPDT start states that have been merged with states in BPDTs at a higher layer). The identifiers of the BPDTs are shown on the shoulders of the boxes enclosing them.

Figure 21 suggests why this method of connecting BPDTs ensures that the structural relations between the location steps are satisfied. For example, consider states \$14, \$15, \$16, and \$17, belonging to BPDTs generated for the location step `//name`. The start states for these BPDTs, \$8, \$10, \$11, and \$13, respectively, coincide with the TRUE and NA states of BPDTs for the location step `//book`. Therefore, only name elements that occur within a book element result in states \$14, \$15, \$16, and \$17 being active. Using a similar argument, we infer that the only book elements that result in states \$8, \$10, \$11, and \$13 being active are those that occur within a pub element.

Another property of our method of connecting BPDTs is that at states of the HPDT that lie in BPDTs in the right subtree of $bpdt(l, k)$, the predicate in l'th location step has not yet been satisfied because these states can be reached only via a path of state transitions that contains the NA state in $bpdt(l, k)$. On the other hand, at states that lie in BPDTs in the left subtree of $bpdt(l, k)$, the predicate in l'th location step has already been satisfied because such these states can be reached only via the TRUE state in $bpdt(l, k)$. Therefore, within each BPDT, the status (satisfied or pending) of predicates in all higher layer (lower numbered) BPDTs are known. The $bpdt(n, 2^{n-1} - 1)$ is in the left subtree of all its ancestor BPDTs. Therefore, at states in this BPDT, all predicates have been satisfied. Consequently, when data that matches the trunk of the query (i.e., the query excluding predicates) is found, it is sent directly to the output using the *output* buffer operation. This situation is exemplified by $bpdt(3, 7)$ in Figure 21; the self-transition emerging from \$17 sends the text contents of the name element to the output. At states within all other BPDTs in layer n, there is at least one predicate that has not yet been satisfied. Therefore, when matching data is found, it is buffered using the *enqueue* operation. This situation is exemplified by $bpdt(3, 5)$ in Figure 21. In state \$15, the predicate `[author]` has been satisfied but the predicate `[year > 2000]` has not been satisfied. Therefore, the self-transition emerging from \$15 buffers the text contents of the name element to the output.

When input events result in a transition out of a BPDT, the truth value of the BPDT's predicate is known. If the predicate evaluates to false, the items are discarded using the *clear* operation. Otherwise, the the *upload* operation is used to transfer the items to the buffer of one of its ancestor BPDTs. More precisely, a BPDT b uploads its buffer items to the buffer of the nearest ancestor b' such that b is in the right subtree of b'. We say that b' is the **D-ancestor** of b. Recalling the method of connecting BPDTs, we note that the predicates of all the ancestors of b that lie below the D-ancestor b' are known to be satisfied since b lies in their left subtrees. Therefore, the contents in b's buffer are waiting next for the result of the predicate that is evaluated by b', and uploading them to b''s buffer is the correct action. We do not need any runtime information to compute the D-ancestor of a BPDT. We note that the parent of $b = bpdt(l, k)$ is $b' = bpdt(l-1, \lfloor k/2 \rfloor)$. Further, b is the left child of b' iff k is even. Thus, the D-ancestor of $bpdt(l, k)$ is $bpdt(l-s, \lfloor k/2^s \rfloor)$ where s is the smallest positive integer such that $2\lfloor k/2^s \rfloor = \lfloor k/2^{s-1} \rfloor$. Equivalently, we may compute the D-ancestor by scanning the binary representation of k right-to-left, looking for the first 0 bit after the least-significant bit. Let k' be the result of truncating k by deleting the suffix that begins at this bit, and s is the length of the truncated suffix. Then, the D-ancestor of $bpdt(l, k)$ is $bpdt(l-s, k')$.

Thus, a BPDT b accepts from its child BPDTs buffer items that are known to satisfy the lower layer predicates (those to the right of its location step in the XPath query) and that must satisfy b's predicate in order to qualify for the result. If b's predicate evaluates to true, these items are sent to the output if no higher level predicates are pending. Otherwise, the items are uploaded to the buffer of the BPDT with the closest pending predicate (b's D-ancestor).

EXAMPLE 7. *This example outlines the basic features of an HPDT, illustrating how it is used to answer XPath queries with multiple predicates. Figure 21 depicts the state transition diagram for the query* `//pub[year>2000]//book[author]//name/text()`. *However, if we ignore the closure and self-closure transitions (arcs marked with* = *and* //, *respectively), we are left with the state transition diagram for the following query without closures:* `/pub[year>2000]/book[author]/name/text()`. *(The original query is discussed is Section 6.2 below.)*

Let us trace the actions of this HPDT given the input stream of Figure 1. Recall that depth stacks are used to distinguish between multiple query matchings for a single element in the input. For a query that does not use the closure axis, there is at most one matching for each element. Therefore, we do not need to consider the depth stacks in this example. (Example 9 shows how the depth stack is used to evaluate the original query, which uses the closure axis.) The HPDT starts in state \$1. When it encounters the `name` *"first," it is in state \$14; thus it enqueues the text content "first" into* $B(3, 4)$. *At the end event of the* `name` *element, the item is uploaded to* $B(2, 2)$. *The next event is the begin event of the* `author` *element. The HPDT goes from state \$8 to state \$9 and uploads the item to the buffer* $B(1, 1)$. *A similar 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* $B(1, 1)$ *contains two items: "first" and "second." When the HPDT encounters the text event of the* `year` *element, it evaluates the predicate* `[text() > 2000]` *to yield true. Therefore the HPDT goes from state \$4 to \$6 and flushes the content of its buffer to the output.*

The above example illustrates how the buffer operations in each BPDT can be determined based on the BPDT's position within the HPDT. For example, since $bpdt(3, 4)$ is the right child of $bpdt(2, 2)$, it is connected to the NA state of $bpdt(2, 2)$. Therefore, at states within $bpdt(3, 4)$, the predicate in $bpdt(2, 2)$ (`[author]`) has not yet been satisfied. Similarly, since $bpdt(2, 2)$ is the right child of $bpdt(1, 1)$, at states within $bpdt(2, 2)$, the predicate in $bpdt(1, 1)$ (`[year > 2000]`) has not yet been satisfied. Combining these facts, at states within $bpdt(3,4)$ we know that neither of the two predicates in the query are satisfied. We note that this information is obtained solely from the positions of the BPDTs, so that the buffer operations in the BPDTs are easily determined.

6.2 Extended buffer operations

Although the *upload* operation and the related data flow described above support multiple predicates in the queries, they cannot correctly handle the case of multiple matchings between the data and the query (described in Example 2). The reason is that items corresponding to different matchings may be stored in the same buffer, rendering them indistinguishable to the subsequent buffer operations. For example, if a BPDT's predicate evaluates to false based on one of the matchings, the entire buffer is cleared. The items that have other matchings that result in the predicate being satisfied cannot be recovered. Since we cannot guarantee the sequence of the evaluation for different matchings, we need to ensure that if the predicate in this BPDT for one of the matchings is not evaluated, the items corresponding to that matching remain in the buffer. (If one of the matchings results in the predicate being satisfied, we can output or upload the items because we only need one correct matching to determine the destination of the buffer items). Example 8 illustrates some of these ideas.

EXAMPLE 8. *Consider the HPDT in Figure 21, for the query `//pub[year>2000]//book[author]//name/text()`, operating on the stream of Figure 2. When the HPDT encounters the `name` element on line 11, it is in state \$14. However, there are three matchings between this element and the query:*
pub in line 2→ book in line 7→ name in line 11
pub in line 2→ book in line 10→ name in line 11
pub in line 9→ book in line 10→ name in line 11.
These different matchings lead to the same sequence of state transitions:
\$1→\$2→\$3→\$8→\$14.
(However, the depth stacks of these states in different matchings are different; this fact is used for distinguishing the buffer items as described later.)

All three matchings lead to the same BPDT because they agree on the predicates that have been satisfied. Since the current BPDT, $bpdt(3,4)$, is in the left subtree of the $bpdt(0,0)$, but in the right subtrees of $bpdt(1,0)$ and $bpdt(2,0)$, we know that only the first predicate is true while the other two are unknown. (The first predicate is the trivially true predicate for the implicit `/root` at the beginning of every XPath query.) However, we cannot simply enqueue the item Z from the text event of the current element. If we do so, then following the first matching, the item will be cleared at the transition from \$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, using the third matching, the HPDT will clear the item when it goes from state \$3 to state \$2, since the `year` subelement of the `pub` element on line 9 fails the predicate in the first location step. However, following the second path, the HPDT should output the item on the transition from \$4 to \$6 when it encounters the `year` element on line 17 since the `book` element in line 10 *has an* `author` *subelement in line 12 (which comes before this* `year` *element and satisfies the second predicate in the query). This need for different behavior for different matchings suggests the need for additional bookkeeping in the buffer.*

To support multiple matchings of the kind discussed above, we store a depth stack with each buffer item, and extend the buffer operations accordingly. Recall, from Example 4, that the depth stack distinguishes between different matchings between a query and a single element in the input. Essentially, we create a copy of the item for each matching, keyed by the corresponding depth stack. The buffer operations of the HPDT in any state operate only on the items in the buffer whose depth stack agrees with that of the state, according to the rules described below.

The depth stack of a buffer item is set by the *enqueue* operation in such a way that it records the depths of the elements that take part in the matching that resulted in enqueuing the item. In Example 8, we listed three matchings for the name element in line 11 of Figure 2 and noted that they correspond to the same sequence of state transitions when we identify states using only their base identifiers. However, if we include the depth stack of each state in addition to its base identifier, we have the following three paths for the three matchings in that example:

($1,$\phi$) $\rightarrow$ ($2,(0)) $\rightarrow$ ($3,(0,1) $\rightarrow$ ($8,(0,1,2))) $\rightarrow$ ($14,(0,1,2,5))
($1,$\phi$) $\rightarrow$ ($2,(0)) $\rightarrow$ ($3,(0,1) $\rightarrow$ ($8,(0,1,4))) $\rightarrow$ ($14,(0,1,4,5))
($1,$\phi$) $\rightarrow$ ($2,(0)) $\rightarrow$ ($3,(0,3) $\rightarrow$ ($8,(0,3,4))) $\rightarrow$ ($14,(0,3,4,5))

The three states with base identifier $14 but different depth stacks represent different matchings between the element and the query. For example, the depth stack $(0, 1, 4, 5)$ associated with an item indicates that the ancestors at depths 0, 1, and 4 are matched with the first, second, and third location step, respectively. Therefore, when the three different states enqueue the text content of a `name` element, they associate different depth stacks with the copies of the items the enqueue. Copies of the same item are later distinguished by their associated depth stacks.

As described in Section 6.1, the items enqueued in the buffer may be uploaded to the upper layer BPDTs and be operated on by operations defined in them. Since states in upper layer BPDTs always have different depth stacks than the states in the lowest layer BPDTs (which have the initial depth stacks associated with the buffer items), we need to ensure that the transitions in upper layer BPDTs operate on the correct buffer items based on the depth stacks. For example, as shown in Example 8, one of the matchings leads the item being cleared in the transition from $8 to $3 in $bpdt(2, 2)$. The other matching leads the item be cleared in the transition from $3 to $2 in $bpdt(1, 1)$. The third matching leads to the item being sent to output from $4 to $6 in $bpdt(1, 1)$. All the states involved in these transitions have depth stacks that are different from the depth stack of the enqueued item. Therefore, we need to devise rules that match operations on transitions to the appropriate buffer items.

The matching of the depth stacks of buffer items with and the depth stacks of HPDT states is achieved by making the following two modifications to the buffer operations: First, the upload operation truncates the depth stack of the uploaded buffer items so that the new depth stack is the same as the depth stack of the NA state of the target BPDT for the same matching. (Recall that there are, in general, multiple active states that have the base identifier of the NA state but different depth stacks, corresponding to different matchings.) Second, a buffer operation on a transition out of a state q in BPDT b operates only on those buffer items whose depth stack is equal to the depth stack of b's NA state for the matching corresponding to q.

When an upload operation moves the buffer items from BPDT b to its D-ancestor b', according to the definition of D-ancestor, the NA state of b' must be in the path of state transitions from the start state of the HPDT to the state that enqueues those buffer items. This NA state will be active when the HPDT returns to b' to process the pending predicate in b'. (It may be active when the items are enqueued if there is a self-closusre axis on it.) Therefore, the depth stack of the NA state must be the first portion of the initial depth stack of the buffer items. It is also easy to conclude that the NA state in a BPDT in layer m must have a depth stack of length $m+1$ where the first element of the stack is always the depth of the document root (which is 0). Therefore, if b' is in layer l', the depth stack of the NA state in b' is the the first $l'+1$ integers of the depth stack that are initially associated by the enqueue operation. We then define the upload operation from b to b' to remove the top $l-l'$ integers in the stack, where l is the layer of b. This process repeats itself for each upload operation that acts on a buffer item. The result is that the depth stacks of an item in a buffer is equal to the depth stack of the NA state of that buffer's BPDT for the matching that originally enqueued the item.

Not all the transitions associated with buffer operations are directly related to the NA state. Therefore, we need to connect the depth stacks of states in those transitions to the depth stack of the NA state. We note that, in any BPDT, the depth stack of an NA state for a given matching is always equal to the depth stack of the TRUE state for that matching. (This observation follows from an examination of the templates in Figures 13, 14, and 16, which include paired begin and end events between their NA and TRUE states; the template in Figure 12 does not have an NA state, while the template depicted in Figure 15 has no begin and end events between the NA state and TRUE state.) Therefore, for a transition from state (s_1, d_1) to state (s_2, d_2), if $s_1(s_2)$ is the TRUE (or NA) state, we set the operation associated with this transition to operate on the buffer items with depth stack d_1 (respectively, d_2). According to the templates in Figure 12 through Figure 16, all the buffer operations are related to either the TRUE state or NA state (or both) except the transition from state \$3 to state \$4 in the template depicted in Figure 16. However, it is also easy to determine the depth stack of the NA state because the depth stack of state \$3 is created by pushing one element onto the depth stack of the NA state. We can obtain the depth stack of the NA state by removing the topmost element from the depth stack of state \$3.

Based on above analysis, we now describe the modified buffer operations used in an HPDT. The buffer alphabet Ω is extended to Ω_H and consists of buffer symbols of the form (v, d), where v is a data item as described in Section 5 and d is the depth stack of that item. The depth stack is associated with the data item by the modified *enqueue* operation described below. The set of buffer operations, F_H, consists of the following, for a transition from state $q_1 = (s_1, d_1)$ to state $q_2 = (s_2, d_2)$ on event

e:

—*enqueue*(v): If $e.type$ is B, add (v, d_2) to the end of the buffer (which is organized as a queue). If $e.type$ is T or E, add (v, d_1) to the end of the buffer. This definition ensures that the depth stack of a buffer item contains exactly the depths of the elements that participate in the matching that justifies enqueuing this item.

—*clear*(): Remove from the buffer all items with depth stack d_1 (since the clear operation is always executed in the transition from the NA state to other states).

—*flush*(): If $s_1(s_2)$ is TRUE or NA state, send to the output the values of all buffer items that have depth stack $d_1(d_2$, respectively). Otherwise, send to the output the values of all buffer items that have depth stack $d_1.remove(1)$. (Recall, from Section 4, that the operation $remove(k)$ removes the top k items from a depth stack.)

—*upload*(): The implicit argument of the upload operation is the target BPDT, which is always the *D-ancestor* of the current BPDT. The rules to determine the depth stack of the buffer items on which the *upload* operation acts are the same as those described for the *flush* operation (since we replace the flush operation with upload operation for BPDTs in which some predicate is still unknown). The upload operation moves all buffer items that have the determined depth stack d to the D-ancestor of the current BPDT. Let l and l' denote the layers of the current BPDT and its D-ancestor, respectively. The depth stack of all the items moved by the upload operation is set to $d.remove(l - l')$.

As before, the operation $output(v)$ outputs the value v directly without buffering.

EXAMPLE 9. *Let us revisit Example 8 using the buffer structure and operations described above. In Figure I, we summarize the actions of the HPDT of Figure 21 given the input data of Figure 2. The first, second, and third columns of the table list the sequence number, summary, and depth of each event as it arrives in the stream. The fourth column, labeled* `Current State Set`*, lists the active states in each step* before *the event is processed by the HPDT. The state transitions that the incoming event triggers are also listed in this column. The* `Buffer` *column lists contents of buffers* after *the event has been processed. Buffers that are not listed are empty.*

Each state is represented as a pair (s, d), where s is the base identifier of the state and d is the depth stack. The base identifier s is used to label the state in the state transition diagram depicted in Figure 21. The depth stack d is determined at runtime based on the rules described in Section 4.

In the `Current State Set` *column, we also list the transitions the current active state takes for the incoming event in the* `Event` *column. Closure transitions (labeled with a = on the arrow in the state transition diagram) are represented as $[(s_1, d_1) \overset{x}{\Rightarrow} (s_2, d_2)]$. Regular transitions are represented as $[(s_1, d_1) \xrightarrow{x} (s_2, d_2)]$ Self-closure transitions are implied if the state stays in the current state set without any explicit transitions. Recall that a transition is taken only if the depth of the incoming event and the depths of the source and target states satisfy the conditions described earlier. The labels on top of the transition arrows are the operations that are executed when the transition occurs. The label e stands for* enqueue*, c for* clear*,*

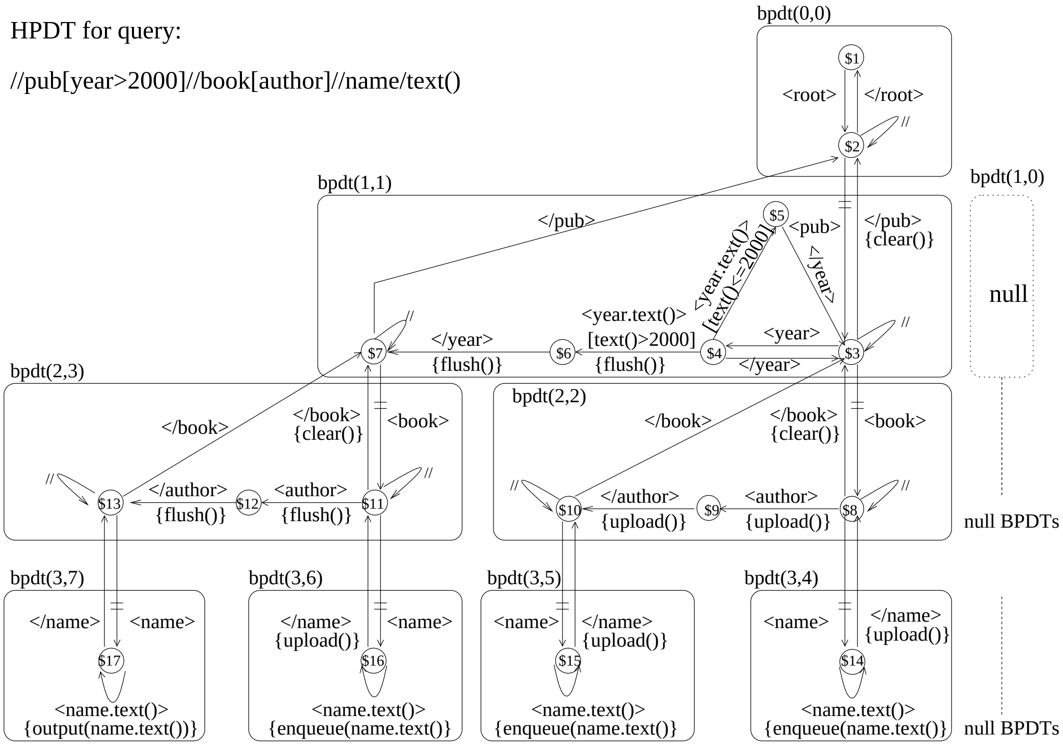


Fig. 20: HPDT for Example 9

u for upload*, f for* flush*, and o for* output*. The states of the buffers after these operations are performed are summarized in the* `Buffer` *column.*

We use $B(i,j)$ to denote the buffer of bpdt(i,j). The last column lists the items in each nonempty buffer $B(i,j)$ using the syntax $B(i,j)$: $e_1, e_2, \ldots$. Each buffer item e_i is of the form $[v,d]$, where v is the value and d is the depth stack associated with the value. Items listed in bold font are those that are enqueued or uploaded by the operation denoted in the labeled state transitions in the previous column. Items with strike-through line are items cleared by an operation in the previous column. Items displayed in a box are items that are flushed or sent to output.

We now highlight some features of this example. First, we note that the HPDT correctly handles multiple matchings. In line 17, for the `z` *element, there are three current states that will respond to the input event. These states have the same base identifier 14, but different depth stacks: $(0,1,2,5)$, $(0,1,4,5)$, and $(0,3,4,5)$. The HPDT puts three copies of the element's content into the buffer $B(3,4)$; the depth stacks for the three copies are the depth stacks of the three source states: $(0,1,2,5)$, $(0,1,4,5)$, and $(0,3,4,5)$. We note that these depth stacks record exactly the depths of the elements that that match the location steps of the query leading to the current state. For example, consider the depth stack $(0,3,4,5)$. It indicates a matching consisting of the* `root` *element at depth 0, a* `pub` *element at depth 3, a* `book` *element at depth 4, and a* `name` *element at depth 5, leading the HPDT to the state $(\$14,(0,3,4,5))$. These copies of the text content* `z` *are processed differently*

	Event	d	Current State Set (before the event)	Buffer (after the event)
1	<root>	0	[($1,($\epsilon$))→($2,(0))]	
2	<pub>	1	[($2,(0)⇒($3,(0,1))]	
3	<book>	2	($2,(0)) [($3,(0,1))⇒($8,(0,1,2))]	
4	<name>	3	($2,(0)) ($3,(0,1)) [($8,(0,1,2))⇒($3,(0,1))]	
5	text()=x	3	($2,(0)) ($3,(0,1)) ($8,(0,1,2)) [($14,(0,1,2,3))$\xrightarrow{e}$($14,(0,1,2,3))]	$B(3,4)$:**[x,(0,1,2,3)]**
6	</name>	3	($2,(0)) ($3,(0,1)) ($8,(0,1,2)) [($14,(0,1,2,3))$\xrightarrow{u}$($8,(0,1,2))]	$B(2,2)$:**[x,(0,1,2)]**
7	<author>	3	($2,(0)) ($3,(0,1)) [($8,(0,1,2))$\xrightarrow{u}$($9,(0,1,2,3))]	$B(1,1)$:[x,(0,1)]
8	</author>	3	($2,(0)) ($3,(0,1)) ($8,(0,1,2)) [($9,(0,1,2,3))→($10,(0,1,2))]	$B(1,1)$:[x,(0,1)]
9	</book>	2	($2,(0)) ($3,(0,1)) [($8,(0,1,2))→($3,(0,1))] [($10,(0,1,2))→($3,(0,1))]	$B(1,1)$:[x,(0,1)]
10	<book>	2	($2,(0)) [($3,(0,1))⇒($8,(0,1,2))]	$B(1,1)$:[x,(0,1)]
11	<name>	3	($2,(0)) ($3,(0,1)) [($8,(0,1,2))⇒($14,(0,1,2,3))]	$B(1,1)$:[x,(0,1)]
12	text()=y	3	($2,(0)) ($3,(0,1)) ($8,(0,1,2)) [($14,(0,1,2,3))$\xrightarrow{e}$($14,(0,1,2,3))]	$B(1,1)$:[x,(0,1)] $B(3,4)$:**[y,(0,1,2,3)]**
13	</name>	3	($2,(0)) ($3,(0,1)) ($8,(0,1,2)) [($14,(0,1,2,3))$\xrightarrow{u}$($8,(0,1,2))]	$B(1,1)$:[x,(0,1)] $B(2,2)$:[y,(0,1,2)]
14	<pub>	3	[($2,(0))⇒($3,(0,3))] ($3,(0,1)) ($8,(0,1,2))	$B(1,1)$:[x,(0,1)] $B(2,2)$:[y,(0,1,2)]
15	<book>	4	($2,(0)) (($3,(0,1))⇒($8,(0,1,4))] [($3,(0,3))⇒($8,(0,3,4))] ($8,(0,1,2))	$B(1,1)$:[x,(0,1)] $B(2,2)$:[y,(0,1,2)]
16	<name>	5	($2,(0)) ($3,(0,1)) [($8,(0,1,2))⇒($14,(0,1,2,5))] ($3,(0,3)) [($8,(0,1,4))⇒($14,(0,1,4,5))] [($8,(0,3,4))⇒($14,(0,3,4,5))]	$B(1,1)$:[x,(0,1)] $B(2,2)$:[y,(0,1,2)]
17	text()=z	5	($2,(0)) ($3,(0,1)) ($3,(0,3)) ($8,(0,1,2)) [($14,(0,1,2,5))$\xrightarrow{e}$($14,(0,1,2,5))] ($8,(0,1,4)) [($14,(0,1,4,5))$\xrightarrow{e}$($14,(0,1,4,5))] ($8,(0,3,4)) [($14,(0,3,4,5))$\xrightarrow{e}$($14,(0,3,4,5))]	$B(1,1)$:[x,(0,1)] $B(2,2)$:[y,(0,1,2)] $B(3,4)$:**[z,(0,1,2,5)]** **[z,(0,1,4,5)] [z,(0,3,4,5)]**
18	</name>	5	($2,(0)) ($3,(0,1)) ($3,(0,3)) ($8,(0,1,2)) [($14,(0,1,2,5))$\xrightarrow{u}$($8,(0,1,2))] ($8,(0,1,4)) [($14,(0,1,4,5))$\xrightarrow{u}$($8,(0,1,4))] ($8,(0,3,4)) [($14,(0,3,4,5))$\xrightarrow{u}$($8,(0,3,4))]	$B(1,1)$:[x,(0,1)] $B(2,2)$:[{y,**z**},(0,1,2)] **[z,(0,1,4)] [z,(0,3,4)]**
19	<author>	5	($2,(0)) ($3,(0,1)) [($8,(0,1,4))$\overset{u}{\Rightarrow}$($9,(0,1,4,5))] ($3,(0,3)) ($8,(0,1,2)) [($8,(0,3,4))$\overset{u}{\Rightarrow}$($9,(0,3,4,5))]	$B(1,1)$:[{x,**z**},(0,1)] **[z,(0,3)]** $B(2,2)$:[{y.z},(0,1,2)]
20	</author>	5	($2,(0)) ($3,(0,1)) ($3,(0,3)) ($8,(0,1,2)) ($8,(0,1,4)) [($9,(0,1,4,5))→($10,(0,1,4))] ($8,(0,3,4)) [($9,(0,3,4,5))]→($10,(0,3,4))]	$B(1,1)$:[{x,z},(0,1)] [z,(0,3)] $B(2,2)$:[{y,z},(0,1,2)]
21	</book>	4	($2,(0)) ($3,(0,1)) ($3,(0,3)) ($8,(0,1,2)) [($8,(0,1,4))$\xrightarrow{c}$($3,(0,1))] [($8,(0,3,4))$\xrightarrow{c}$($3,(0,3))] [($10,(0,1,4))→($3,(0,1))] [($10,(0,3,4))→($3,(0,3))]	$B(1,1)$:[{x,z},(0,1)] [z,(0,3)] $B(2,2)$:[{y,z},(0,1,2)]
22	<year>	4	($2,(0)) ($3,(0,1)) [($3,(0,3))⇒($4,(0,3,4))] ($8,(0,1,2))	$B(1,1)$:[{x,z},(0,1)] [z,(0,3)] $B(2,2)$:[{y,z},(0,1,2)]
23	text()=1999	4	($2,(0)) ($3,(0,1)) ($3,(0,3)) ($8,(0,1,2)) [($4,(0,3,4))→($5,(0,3,4))]	$B(1,1)$:[{x,z},(0,1)] [z,(0,3)] $B(2,2)$:[{y,z},(0,1,2)]
24	</year>	4	($2,(0)) ($3,(0,1)) ($3,(0,3)) ($8,(0,1,2)) [($5,(0,3,4))→($3,(0,3))]	$B(1,1)$:[{x,z},(0,1)] [z,(0,3)] $B(2,2)$:[{y,z},(0,1,2)]
25	</pub>	3	($2,(0)) ($3,(0,1)) [($3,(0,3))$\xrightarrow{c}$($2,(0))] ($8,(0,1,2))	$B(1,1)$:[{x,z},(0,1)] ~~[z,(0,3)]~~ $B(2,2)$: [{y,z},(0,1,2)]
26	</book>	2	($2,(0)) ($3,(0,1)) [($8,(0,1,2))$\xrightarrow{c}$($3,(0,1))]	$B(1,1)$:[{x,z},(0,1)] $B(2,2)$:~~[{y,z},(0,1,2)]~~
27	<year>	2	($2,(0)) [($3,(0,1))⇒($4,(0,1,2))]	$B(1,1)$:[{x,z},(0,1)]
28	text()=2002	2	($2,(0)) ($3,(0,1)) [($4,(0,1,2))$\xrightarrow{f}$($6,(0,1,2))]	$B(1,1)$: **[{x,z},(0,1)]**
29	</year>	2	($2,(0)) ($3,(0,1)) [($6,(0,1,2))→($7,(0,1))]	
30	</pub>	1	($2,(0)) [($3,(0,1))→($2,(0))] [($7,(0,1))→($2,(0))]	
31	</root>	0	($2,(0))	
32			($1,($\epsilon$))	

Fig. 21: HPDT actions for Example 9

based on their depth stacks. The `z` *item with depth stack* $(0, 3, 4, 5)$ *is first uploaded to BPDT* $bpdt(2, 2)$ *and the depth stack is modified to* $(0, 3, 4)$ *in line 18, indicating that the predicate for the* `name` *element at depth 5 has been satisfied. It is then uploaded to* $bpdt(1, 1)$ *and the depth stack is modified to* $(0, 3)$ *since the predicate for the* `book` *ancestor at depth 4 has also been satisfied. When the predicate for the* `pub` *ancestor at depth 3 (*`[year > 2000]`*) evaluates to false at the end of the* `pub` *ancestor in line 25, the item is cleared from the buffer. However, since different copies of the items follow different flows among the buffers, the other copies will not affected by this operation and be processed correctly based on their matchings.*

The process outlined in Figure I also demonstrates that an item is always removed or flushed from a buffer as soon as its membership in the result set can possibly be determined. For example, the `z` *element with depth stack* $(0, 3, 4, 5)$ *is removed from the buffer when the* `pub` *element in the matching fails the predicate* `[year > 2000]` *at the end of the* `pub` *element. (Before this point in the stream, it is impossible to determine that the predicate fails because an additional* `year` *element satisfying the predicate may appear at any point before the* `</pub>` *event.) The items in the result (*`x` *and* `z`*) are sent to the output as soon as the last pending predicate* `[year > 2000]` *is satisfied.*

This example also illustrates how buffer items with the same depth stack are processed together. In line 19, the entry $[\{x, z\}, (0, 1)]$ *indicates that the two items have the same depth stack, and thus should be processed as a group. Although these two items are at different depths, they are in the same group because they both have satisfied all the predicates in lower layer BPDTs. The item* $[x, (0, 1)]$*, which comes from* $[x, (0, 1, 2, 3)]$*, has satisfied the predicate in the third location step* `/name` *(which has the null predicate that is always true) and the second location step* `book[author]` *(with the* `book` *element at depth 2). The item* $[z, (0, 1)]$*, which comes from* $[z, (0, 1, 4, 5)]$ *has satisfied the predicate in the third location step and the predicate in the second location step (with the* `book` *element at depth 4). Although they match different elements that satisfy the predicates in lower layer BPDTs, they are both waiting for the result of the predicate of the same* `pub` *element at depth 1 (and predicates of the same ancestors in the upper layer BPDTs, if there are any), which is determined by their depth stacks.*

We note that the matching rules between the depth stacks of buffer items and the states in the transitions ensure that operations act only on the buffer items for the matching relevant to the transition. For example, in line 26, the state $(\$3, (0, 3))$ *transitions to state* $(\$2, (0))$ *and the* clear() *operation is executed. At the time, there are three items in the buffer* $B(1, 1)$*:* $[\{x, z\}, (0, 1)]$ *and* $[z, (0, 3)]$*. The* clear *operation removes only the item* $[z, (0, 3)]$ *from the buffer since the depth stacks match. The other copy of z (*$[z, (0, 1)]$*) remains in the buffer since it is waiting for the end of the other* `pub` *element (which will later result in the predicate evaluating to true). In line 28, at the text event, the state* $(\$4, (0, 1, 2))$ *transitions to state* $(\$6, (0, 1, 2))$ *and the* flush *operation is executed. Although the depth stack is* $(0, 1, 2)$ *for the source state, according to the rules defined earlier in this section, for this text event of the subelement* `year`*, the matched depth stack for the flush operation should be the depth stack without the last integer, which is* $(0, 1)$*. Therefore, the items in the buffer with depth stack* $(0, 1)$ *are flushed to output.*

Input: XPath Query query = $N_1N_2...N_n/O$
Output: an HPDT in the form of an array of BPDTs

```
GenerateHPDT( ){
   /* Generate bpdt(0,0) based on Figure 20. */
   bpdt(0,0) = generateRootBPDT( );
   for ( l = 1 to n ) {
      for ( k = 0 to 2^(l-1) -1 ) {
         if ( bpdt(l-1, k) != null ){
            bpdt(l,2k+1)=addBPDT(bpdt(l-1,k),N_l,TRUE);
            if ( bpdt(l-1, k).na != null )
               bpdt(l,2k)=addBPDT(bpdt(l-1,k),N_l,NA);
         }
      }
   }
   /* Add output to the lowest layer BPDTs. */
   for ( k = 0 to 2^(n-1) -1 )
      addOutput(O, bpdt(n,k)) ;
}
```

Fig. 22: Algorithm GenerateHPDT

We also note that the state $(\$2,(0))$ *remains current for almost the whole process. The reason is that, due to the closure axis in the first location step, a* **pub** *element at any depth matches the first location step. Since this state is used to match the* **pub** *elements, not until we encounter the end of the stream can we remove this state from the current state set. However, if we know beforehand that the data is not recursive, i.e., no node has an ancestor with the same name, then we do not need to keep the state as active after it is matched with an element even when the query has closure axes. The reason is that once we match a* **pub** *element with this state, we know there will not be any more* **pub** *elements inside this element, and this state will not match any other elements until the end of this current matched* **pub** *element.*

The above example illustrates some of the complexities resulting from closure axes in the query and recursive structure in the input data. Due to the possibly multiple matchings between the query and the data, we have to check all the possibilities and record extra information. We note that all streaming XPath processors that use minimal buffering (i.e., any data they buffer must also be buffered by any other streaming XPath processor) need to perform such bookkeeping. As demonstrated in Section 9, XSQ is able to handle these difficult cases without compromising the efficiency in the simpler cases.

6.3 Building HPDTs from XPath Queries

We now complete our description of the method used by XSQ to map an XPath query to an HPDT that evaluates the query over streaming data. We first describe the high level method that builds the HPDT structure. Two important subroutines of the process, called *addBPDT* and *addOutput* are explained in detail later. Consider an XPath query $N_1N_2...N_n/O$, where N_i denotes the ith location step and

```
addBPDT(BPDT p, LocationStep N, State s){

  /* instantiate template matching N */
  n = createBPDT(N);

  /* connect to parent BPDT */
  mergeStates(n.start , s);

  /* set BPDT id = (layer, seqnum) */
  n.layer = p.layer + 1;
  if(s.type == TRUE)
     n.seqnum = 2 * p.seqnum + 1;
  else /* s == NA */
     n.seqnum = 2 * p.seqnum;

  if (n.seqnum != 2^{n.layer}-1)
     n.bufOp = UPLOAD;
     /* set flush ops to upload */
     setFlushToUpload(n);
  }
  else n.bufOp = FLUSH;

  /* For closure axis, add a self-closure transition to the START state.*/
  if (N.axis == closure)
    newTrans(n, START, START, BEGIN, "//");

  /* Make all transitions on the BEGIN event of n.tag out of the START
     state closure transitions. */
    transitions = getTrans(n, START, BEGIN, n.tag);
    for(t in transitions) t.type = CLOSURE;

    /* Put an extra flush/upload operation on the transition in the parent
       that processes the end event of the predicate's subelement. */
    t2 = getEndOfChildTran(p);
    addOp(t2 , p.bufOp, null)
  }
}
```

Fig. 23: Subroutine addBPDT

O denotes the output expression. For ease of exposition, we will use N_0 to denote an implicit `/root` prefix for all XPath queries. Figure 22 presents the pseudocode that summarizes the top-down creation of an HPDT as described in Section 6.1.

As indicated by the pseudocode of Figure 22, the bulk of the BPDT-generation work is done within the *addBPDT* subroutine. The pseudocode for *addBPDT* is listed in Figure 23. This subroutine is responsible for creating a new BPDT based on a location step and setting the buffer operations in the new BPDT. Further, it is responsible for connecting the new BPDT to the appropriate higher-level BPDT, which may also need some modifications. The subroutine takes three parameters: the parent BPDT p, the location step N, and the state s in BPDT p to which the

```
addOutput(BPDT b, OutputFunction O){
  if (b.seqnum != 2^{b.layer} -1) op = ENQUEUE;
  else op = OUTPUT;
  switch (O.type){

    case ATTRIBUTE:
      t = getTrans(b, START, BEGIN, b. tag);
      addOp(t, op, "@"+O.attrname);
      break;

    case TEXT:
      /* Add a new transition from the NA state to the NA state that
         processes the TEXT event of b.tag.*/
      t = newTran(b, NA, NA, TEXT, b.tag);

      /* Add the operation op with the parameter b.tag+".text()" to the transition.*/
      addOp(t, op, b.tag+".text()");
      t = newTran(b,TRUE,TRUE,TEXT,b.tag);
      addOp(t, op, b.tag+".text()");
      break;

    case CATCHALL:
      t = getTrans(b, START, BEGIN, b.tag);
      addOp(t, op, "*");
      t = newTrans(b, NA, NA, CATCHALL);
      addOp(t, op, "*");
      t = newTrans(b, TRUE, TRUE, CATCHALL);
      addOp(t, op, "*");
      /* Get the transition that going from the TRUE state to the START state
         processing the END event of b.tag. */
      t = getTrans(b, TRUE, START, END, b.tag);
      addOp(t, op, "*");
      /* add extra upload/flush operation */
      t = getEndOfChildTran(b);
      addOperation(t, b.bufOp, null);
      break;
  }
}
```

Fig. 24: Subroutine addOutput

new BPDT is connected. It uses the *createBPDT* function to generate a BPDT by matching the location step N with the templates (depicted in Figures 12 through 16 in Section 5.2) and binding the symbols in the templates to the actual values in the location step. The START state of this new BPDT is merged with s (the TRUE or NA of p). That is, the two states are assigned the same ID and the transitions associated with them are combined (function *mergeStates*). Other states in the new BPDT are assigned a unique (arbitrary) state identifier. The newly created BPDT is then assigned an identifier of the form (l, k) based on the state s to conform to the scheme described earlier.

The *addBPDT* subroutine also sets the buffer operations in the new BPDT based on its identifier. The *clear* operations remain identical to those in the templates. For $bpdt(l, k)$ with $k \neq 2^l - 1$ (i.e., all except the leftmost BPDT), the *flush* operation in this BPDT is replaced by an *upload* operation. Since this BPDT is in the right subtree of at least one ancestor (otherwise $k = 2^l - 1$), we know that at least one predicate for the items in the buffer is still not satisfied. For $bpdt(l, 2^l - 1)$, the *flush* operation is left unchanged.

If the location step N uses the closure axis, the *addBPDT* subroutine modifies the transitions in the new BPDT. It first adds a *self-closure transition* from the START state to itself, labeled with `//`. This transition permits the HPDT to stay in the START state for any begin event that comes from the subelements for the current element. It then sets as *closure transitions* the transitions that emerge from the START state and process the begin event of the node test in the location step. (There is only one such transition in all BPDTs except those generated using the template in Figure 12.) These transitions permit the HPDT to accept the subelement of any depth that matches the node test of the current location step.

In addition to the modifications made to the transitions, an extra buffer operation is needed for the p BPDT when the location step N has a closure axis. Let us consider an example to illustrate the necessity of the extra buffer operation. Recall the BPDT depicted in Figure 18, which evaluates the query `/book[author]/price/text()`. Now consider the following query, which differs from the earlier one only in the axis of the second location step being descendant-or-self instead of child: `/book[author]//price/text()`. The corresponding changes to the BPDT of Figure 18 involve adding two self-closure transitions to states \$2 and \$4 and marking the transitions \$2 $\rightarrow$ \$5 and \$4 $\rightarrow$ \$6 as closure axes (marking the arcs with `=`). At first glance, these changes may seem sufficient and the resulting automaton may seem to accurately process the new query. However, a closer examination reveals a problem in the case of `price` elements that have both `book` and `author` elements as ancestors. Correctly processing such elements requires a *flush* operation on transition \$3 $\rightarrow$ \$4. The `price` elements that are descendants of both `book` and `author` elements always occur between the begin and end events of the `author` element; they will be enqueued by the self-transition on \$5, and flushed to output by the operation on the transition from \$3 to state \$4. We note that this modification is needed only for parent BPDTs generated using the templates in the following Figures (with the affected transitions in parentheses): Figure 13 (\$4 $\rightarrow$ \$5), Figure 14 (\$3 $\rightarrow$ \$4), and Figure 16 (\$4 $\rightarrow$ \$5). (The affected transitions are returned by the `getEndOfChildTran` function in Figure 23.) The added extra operation could be flush or upload, based on the buffer operation used in the parent BPDT.

Returning to Figure 22, we note that after all the BPDTs have been generated and connected, the *addOutput* subroutine is used to add output operations to the BPDTs in the lowest layer. The pseudocode for this subroutine is presented in Figure 24. First, as described in Section 6.1, only $bpdt(n, 2^n - 1)$ uses the direct *output* operation because it is the only BPDT in which any data matching the trunk of the query has already satisfied all the predicates in the query. The other BPDTs in the lowest layer use *enqueue* operations in place of the *output* operation. Next, the BPDT is modified by adding further operations and transitions, determined by the type of the output function O. If O specifies outputting an attribute of the element specified by the last location step, an output or enqueue operation will be added to every transition emerging from the START state that processes the begin event of that element. If O specifies outputting the text content of the element, a self-transition is added to every NA and TRUE state in this BPDT. A *output(text())* or *enqueue(text())* operation is added to the new transition. If O specifies outputting the whole element specified by the last location step, we add a self-transition labeled with $\bar{*}$ (catch-all) to every NA and TRUE state associated with the *output*($\bar{*}$) operation. These two transitions will match all the subelements and text contents of the current element. The operation *output*($\bar{*}$) is also added to the transition that emerges from the START state that processes the begin event of the current element and to the transition from the TRUE state to the START state that processes the end event of the current element. All these newly added operations will match every event within the current element. We also note that since the catch-all transitions essentially function as closure transitions (accepting incoming events at any larger depth), we have to add an extra buffer operation (flush or upload) to the current BPDT as described above.

6.4 Aggregations

Given the above machinery, very little extra work is required for supporting aggregates. For this purpose, XSQ uses a statistics buffer *stat*. In the stat buffer, there is one item for each aggregation function, with initial value *null*. There are two operations on this buffer:

—`update(aggr,value)`: Update the item for aggregation function *aggr* in *stat* with the *value*. For example, `update(COUNT,2)` will add 2 to the number in *stat*.

—`output(aggr)`: Output the value of the function *aggr* in *stat*.

For example, consider the following query, which differs from the query of Example 9 only in using an output function `count()` instead of `text()`:
`//pub[year>2000]//book[author]//name/count()`
To evaluate this query, we use an HPDT that is almost identical to the one depicted in Figure 21. We replace all occurrences of `flush()` with `update(COUNT, v)`, where v is the number of items in the BPDT's queue. We also replace all instances of `output(value)` with `update(COUNT,1)`. Finally, we place `output(COUNT)` on the transition from \$2 to \$1. We may also modify the semantics of `update()` so that it emits a new value whenever the number in the buffer is updated. This change makes the result of the aggregation query available in an online manner. This feature is especially useful when we process aggregation queries over unbounded streams.

6.5 Analysis

We provide a detailed experimental analysis of XSQ in Section 9. Here, we present a simple worst-case analysis of the space and time costs of the method described above. Consider an XPath query that has q location steps. In worst case, when each location step involves a predicate, our method results in an HPDT built from 2^q BPDTs. (Note that none of the BPDTs in the subtree rooted at $bpdt(1,0)$ are generated.) Recall, from Section 5, that our method generates BPDTs for each step based on the templates of Figures 12, 15, 13, 14, and 16. The largest of these (the template for a step of the form `/tag[child=val)` has six nodes. However, the start state of each BPDT other than the root BPDT is identical to one of the states in its parent BPDT. Therefore, the number of states in the HPDT is at most $5 \cdot (2^q - 1) + 2 = 5 \cdot 2^q - 3$ states (since the root BPDT has two nodes). Although the exponential dependence on query length may seem problematic at first, we note that the HPDT has a very regular structure that lends itself well to optimizations in the implementation. In particular, level k of the HPDT consists of 2^k BPDTs that are very similar to each other. An implementation may choose a compressed representation of the state space they encode, by using a bit vector to indicate which subset of the BPDTs at a given level are active. The current version of XSQ does not perform such optimizations. However, as we indicate in Section 9, the memory used for the HPDT is still modest. In fact, the dominant space cost for most query-data combinations is not the HPDT but the buffers used to hold potential query results. By examining all the cases for buffer operations in the HPDT, we observe that an item is in the buffer exactly when its membership in the query result cannot be decided based on the portion of the stream that has already been seen. It follows that every streaming processor must buffer such an item. Therefore, the buffering mechanism in XSQ is optimal in the sense that at any point in time, the buffer of any streaming XPath processor must include at least the buffer items in XSQ's buffer.

The appropriate measure of time complexity for a streaming query processor is the amount of work it must perform for each unit of input. In the case of XSQ, the critical factor determining the amount of such work is the number of currently active states in the HPDT. If the query does not use the closure axes (`//`, denoting descendant-or-self, and its variants), then there is only one active state at any time. Thus, for each input symbol, we need to check transitions from only one state. By hashing on tag names, matching transitions can be selected in constant time by using perfect hashing (ignoring the typically modest hash function evaluation time). Thus, the case of no closure axes leads to a constant amount of work per input byte.

The worst case is when all q location steps of the query use closure axes and have predicates associated with them. The amount of work performed by XSQ in this case depends on the structure of the input stream. If the stream does not contain recursive structure then each closure state generates only one state in the runtime set of current states (and the *depth stack* is not needed). The size of the current state set is at most $O(2^q)$, and for each SAX event, we have to check for possible transitions $O(2^q)$ times. The maximum number of transitions on the incoming event for each of these states is two (one self-transition and one transition to another state). The amount of work XSQ must perform for each input item (SAX event) is $O(2^q)$, in worst case. We note that this result is only a rough estimate. Since elements that match the first location step have at most one state to check for possible transitions, elements that match i'th location step have at most 2^i states to check, and only the elements that matche the pattern will be checked (the others will not lead to any state transitions at all). The actual number of operations depends on the degree of similarity between the data and the query and the structure of the data.

If the stream does contain recursive structure, the number of current states depends on the number of ways each element can match the corresponding location step. For example, if an element in the final result has k ways to match the query, the HPDT may create $O(2^q)$ current states. Then, for each state in layer q (the lowest layer), k copies of the state are generated at runtime, each with a different depth stack. For the states in a higher layers, no more than k copies are generated. Therefore, the amount of work per input item is is $O(2^q k)$ in worst case. As noted earlier, this worst case result is only a rough estimate and real queries and streams are unlikely to incur the worst case costs. We explore these and other issues in detail in Section 9.

7. RELATED WORK

Several papers have addressed the problem of *filtering* a stream of XML documents [Altinel and Franklin 2000; Green et al. 2003; Diao et al. 2002; Lakshmanan and Sailaja 2002; Chan et al. 2002]. This problem has been referred to variously as *selective dissemination of information (SDI)*, *publish-subscribe (pub-sub)*, and *query labeling*. Briefly, filtering assumes that the input is a stream of documents that are to be matched with a given set of queries. A query is said to match a document if the result of evaluating the query on the document is non-empty. Since there is no output other than the identifiers of the documents matching each query, methods for filtering are simpler than those needed for querying. As described in Section 3, we may think of methods for filtering as starting points for the exploration of more general methods for querying. Filtering systems typically focus on supporting high throughput for a large number of queries using only a moderate amount of main memory.

The *XFilter* system [Altinel and Franklin 2000] focuses on the problem of evaluating a large number of XPath filter expressions over every document in a stream of documents. It uses finite-state automata similar to those described in Section 3. Since the filter expressions are likely to have many common segments, the automata are combined and indexed to yield an efficient filtering method. The *YFilter* system [Diao et al. 2002] addresses a similar problem and uses one automaton to evaluate all submitted filter expressions. It combines all the automata into one big automaton that uses a run time stack to track all the possible states for all the queries. Instead of the index used by XFilter, YFilter uses query identifiers in the states to denote the queries corresponding to the results. The method described in [Chan et al. 2002] uses a data structure called *XTrie* instead of a flat table to index XPath queries based on common substrings among them. Automaton-based methods spend a significant amount of time matching transitions to incoming events; as a result, deterministic automata typically yield higher throughput than their nondeterministic counterparts. However, as usual, the deterministic version of an automaton may require a large amount of memory. This problem is addressed in [Green et al. 2003] by using a lazy deterministic finite state automaton. The main idea is to first build a naive finite-state automaton directly from the XPath expression. At run time, the system adds new states as needed on the fly. Since it does not need to use a stack to keep track of all possible states, its throughput is improved. Although the deterministic automaton requires more memory than its nondeterministic counterparts, an upper bound on the size of DFA is provided in [Green et al. 2003].

The problem of *query labeling* is studied in [Lakshmanan and Sailaja 2002]. The authors propose a *requirements index* as a dual to the traditional data index. A framework is provided to organize the index efficiently and to label the nodes in streaming XML documents with all the matched requirements in the index. The problem of validating XML streams using pushdown automata has been studied in [Segoufin and Vianu 2002]. (Briefly, an XML document is said to be *valid* with respect to a given *Document Type Definition (DTD)* if the document structure obeys the grammar specified in the DTD [Bray et al. 1998].) This problem can also be considered as a filtering problem because the pushdown automaton can filter the documents that satisfy the DTD.

As noted earlier, the above systems support filtering, not querying, of XML streams. Further, they either do not support predicates, or support only simple predicates that test structural information (whether an element has specified descendant). The *YFilter* system [Diao et al. 2002] supports predicates that do not reference other elements so that the predicate can be evaluated immediately when the related input element is encountered. Since the YFilter system only filters the XML stream, it need not handle the case where the predicates are evaluated in different sequences.

A transducer-based approach to evaluating *XQuery* queries on streaming data is presented in [Ludascher et al. 2002]. An XQuery is decomposed into subexpressions and each subexpression is mapped to an *XML Stream Machine (XSM)*. Each XSM consumes the content of its input buffer and writes output to its output buffers. The output buffer of one XSM may be the input buffer of another. This producer-consumer relationship of XSMs through their buffers results in a network of XSMs. This network is merged into a single XSM that can be optimized if the DTD for the input data is available. (In [Olteanu et al. 2002], a similar approach is used to evaluate regular path expressions with qualifiers over well-formed XML streams. That paper proposes a transducer network model called *SPEX*, in which each transducer is generated from a regular path expression construct. The output tape of one transducer forms the input tape of another.) The key differences between XSQ and XSM are as follows: First, XSQ supports XPath features such as aggregations, closures, and multiple predicates that are not supported by XSM. As described in earlier sections, these features, especially in combination, complicate query processing. Second, XSM supports constructors in XQuery expressions while XSQ supports only XPath (no constructors). XSQ uses this simplification to work with a simpler automaton and a simpler model of buffer interactions. Third, the combined, optimized XSM is quite complicated, making it difficult to group similar queries. In contrast, the HPDT has simple structure, and methods such as those in [Diao et al. 2002] can be easily applied to it. At the time of writing, the XSM system was not available for testing and it is therefore omitted from our study in Section 9. However, we believe that XSQ and XSM are practical demonstrations of the trade-offs between query language expressiveness and system simplicity and efficiency (XPath vs. full XQuery).

An interesting feature of the *XAOS* system [Barton et al. 2003] for streaming

XML is that it supports XPath's reverse axes, such as `parent` and `ancestor`. It uses two data structures called *X-tree* and *X-dag* to reduce the amount of streaming data buffered in a *matching structure*. Essentially, the X-tree is the parse tree of the XPath expression, with reverse axes permitted. The X-dag is the equivalent XPath representation with reverse axes removed. The X-dag is used as a pattern to filter the incoming stream to remove the irrelevant nodes. The relevant nodes are stored in the matching structure based on their relations in the X-tree. When the end of the stream is encountered, results are produced by traversing the matching structure. A drawback of this approach is that it does not output any results until the end of the stream is encountered. (For unbounded streams, a periodic evaluation of the matching structure could be used.) Unlike XSQ, XAOS supports reverse axes; however, unlike XAOS, XSQ produces incremental results and buffers data in an optimal manner (least amount of data for the least amount of time possible). Rewriting XPath queries with reverse axes into equivalent queries with only forward axes is studied in [Olteanu et al. 2002]. However, since the rewriting algorithm introduces node set comparison operations in the new expression, the approach is difficult to apply in a streaming environment. For example, for an expression `X[ancestor::Y/Z]`, the rewriting algorithm produces `X[/descendant::Y[Z]/descendant::node()=self::node()]`. We believe it should be possible to combine some of the ideas used in XSQ, XAOS, and the method of [Olteanu et al. 2002] to yield a system that supports reverse axes without sacrificing buffer space.

Several systems provide methods for querying non-streaming XML data. *Galax* [Fernandez and Simeon 2002] is a full-fledged XQuery query engine. It implements almost all of the XML Query Data Model along with the type system and dynamic semantics of the XML Query Algebra. *XQEngine* [Katz 2002] is a full-text search engine for XML documents that uses XQuery and XPath as its query language. XPath expressions and boolean combinations of keywords are used to query collections of XML documents. The engine creates a full-text index for every document before the document can be queried. It is difficult to adapt these systems for streaming data. Nevertheless, we use them in our experimental study in Section 9 for comparison purposes.

A topic closely related to XPath query processing is *XML transformation*. *XSLT* is a standard template-based language for transforming XML [W3C XSL Working Group 2002]. Since XSLT uses XPath to specify patterns in its rules, XSQ and other methods for XPath processing have applications in XSLT processors. As studied in Section 9, the popular implementation of XSLT in Saxon [Kay 2002] is based on an in-memory materialization of the entire XML document and is therefore limited in the size of documents it can efficiently transform. By using a streaming XPath processor such as XSQ, we can design an XML transformation system that buffers only limited amount amounts of data.

The *STX* system takes a different, more procedural, approach to transforming streaming XML [Becker et al. 2002]. It uses templates to specify the operations that should be performed when data matching the template pattern is encountered. We may think of STX as a general-purpose event-driven programming environment that is not tailored to a specific query language. However, it may be used for

XPath processing if we design a method for generating efficient STX templates from XPath queries. For example, if there are two predicates in an XPath query, we may create two variables in the program to store the current results of the predicates. When a predicate is evaluated, the corresponding variable is set to the result of the evaluation. We also need to specify explicitly when to reset the variables. We may then choose the right operation based on the current values of the variables. However, in this scheme, the positions of the elements have to satisfy the requirement that the predicate is evaluated before the target items. In general, it is not obvious how to generate STX templates equivalent to an XPath query in a systematic manner. However, this approach is an alternative to our automaton-based approach and would benefit from further attention.

The *query complexity* of XPath is addressed by [Gottlob et al. 2002], which provides a main-memory algorithm for evaluating XPath on non-streaming data that is polynomial in the size of the query (and data). The method is based on reducing every axis to two primitive axes: *first-child* and *next-sibling*. The algorithm traverses the XPath parse tree in a bottom-up manner. The subexpressions in the lowest level are evaluated by scanning the data. The results of these subexpressions are then used in the evaluation of their parent subexpressions, recursively. The paper also provides a refined top-down algorithm and suggest a core subset of XPath that can be evaluated in linear time. Since these methods require multiple passes of the data, it is not easy to adapt them methods for a streaming environment. However, it should be interesting to investigate the issues raised by this paper in a streaming environment.

The evaluation of XPath queries over XML data is closely related to the problem of *tree pattern matching* [Miklau and Suciu 2002; Chen et al. 2001]. As described in [Miklau and Suciu 2002], despite the resemblance, there are important differences between XPath evaluation and the classical problems of *tree pattern matching* [Hoffmann and O'Donnell 1982] and *unordered tree inclusion* [Kilpel 1992]. In particular, the problem of unordered tree inclusion is NP-hard (by direct reduction from SAT) [Kilpel 1992], while XPath queries can be answered in polynomial time [Gottlob et al. 2002]. Intuitively, the reason the inclusion problem is harder than the XPath problem is that the former does not permit multiple nodes in the pattern tree to be mapped to the same node in the data tree. Most of the algorithms for these problems require a postorder (bottom-up) traversal of the data trees and are thus unsuitable for streaming data that is provided in preorder. As an exception, the algorithm described in [Hoffmann and O'Donnell 1982] for the classical tree pattern matching problem needs only a preorder traversal of the data tree. However, it allows only parent-child (not descendant) edges in patterns and finds only matches for which siblings occur in the same order in the data and as in the pattern. On the other hand, tree patterns corresponding to XPath queries include ancestor-descendant edges (for the closure axis) and XPath semantics require that the sibling order in the pattern (order of nodes mentioned in predicates) be ignored. Therefore, this algorithm cannot be easily applied to XPath.

An *alternating automaton* is an automaton in which each state has a flag indicating the acceptance or rejection [Chandra et al. 1981]. There are three types of states: *universal*, *existential*, and *negating*. A universal (existential) state becomes an accepting state if all (respectively, at least one) of its offspring states reach accepting states. A negating state has a unique offspring and becomes an accepting state only when the offspring state is a rejecting state. There are two difficulties in applying alternating automata for streaming XPath evaluation: First, alternating automata naturally express the semantics of filtering expressions, but not querying expressions. In particular, they do not provide a mechanism to solve the address the buffering problems discussed in Section 5. Second, they use a bottom-up model of computation that does not fit well with the preorder arrival of streaming XML input. However, it may be possible to adapt some of the ideas used by alternating automata for XPath.

The *Aurora* system [Carney et al. 2002; Cherniack et al. 2003; Zdonik et al. 2003] is a data stream management system for monitoring applications, in which typical tasks include tracking the abnormalities among multiple streams, filtering specific target data for the user, and executing queries involving aggregations and joins. The Aurora system processes data streams using a large trigger network. The trigger, which is essentially a data-flow graph, is generated from the persistent queries provided by applications. The tuples in the results of these queries are created from the incoming streams and fed into the original application also in streaming form. The Aurora system provides a set of operators for an application to specify the persistent query and quality of service (QoS) requirements. At runtime, the Aurora system is optimized by using techniques such as load shedding (discarding data that requires a long time to process) and real-time scheduling.

The *Fjords* architecture [Madden and Franklin 2002] has been developed for managing multiple queries over the numerous data streams generated from sensors. Sensor data is generated in streaming form and the data rate is typically high and variable. The Fjords architecture is designed to maintain a high throughput for queries even when the data rate is unpredictable. It provides an efficient and adaptive infrastructure for more sophisticated query applications. The main components of the architecture are the queuing system and the sensor proxies. The queues can function in either pull or push mode. They are the basic functional structures to route data between the operators in a query plan. Query operators may be adaptive, such as Eddies [Avnur and Hellerstein 2000]. Each sensor has a sensor proxy that accepts queries and tries to simplify the queries for the sensor's processor. The proxy adjusts the sample rate of the sensor based on the queries and permits different users share data from the sensor. Such optimizations result in higher throughput and longer sensor battery life, since energy is conserved by avoiding unnecessary sampling.

The *NiagaraCQ* system is designed to efficiently support a large number of subscription queries expressed in *XML-QL* over distributed XML datasets [Chen et al. 2000]. It groups queries based on their signatures. Essentially, queries that have similar query structure by different constants are grouped and share the results of the subqueries representing the overlap among the queries. NiagaraCQ and XSQ work at different granularities of data. Although NiagaraCQ handles both change-based and timer-based continuous queries, the events it handles (such as changed remote XML file and activated timer) are at a high level. Therefore, it can use materialized data that is managed by a cache manager. In contrast, systems such as XSQ and XFilter respond to every event generated by a SAX-like parser. XSQ evaluates queries on streaming data, and the result is also in streaming form. These two granularities are complementary: One can combine the methods of NiagaraCQ for the larger granularity with the methods of XSQ for the finer granularity.

A related system, *WebCQ*, implements server-based Web page monitoring [Liu et al. 2000; 1999]. Users use WebCQ's own query language to specify a sentinel, which is essentially a request for monitoring the specified Web objects. The sentinel supports different kinds of objects, such as images and links in Web pages, different time intervals for change detection, and different kinds of notification mechanisms. Although both WebCQ and XSQ are event-driven systems, the events in WebCQ systems are specified by the user and are mostly timer-based. When a timer is activated, WebCQ visits the specified Web resource and pulls the content that will be compared with its stored version in the cache. XSQ, in contrast, is more like a push-based system that receives the data passively and returns the results continuously. Further, like NiagaraCQ, WebCQ also operates at a larger granularity than does XSQ.

Another system for processing data streams is *dQUOB* [Plale and Schwan 2003; 2000]. It views the data streams as a relational database. Each event in the stream maps to a tuple in a relation that characterizes the stream. It uses SQL extended with *create-if-then* rules from *Starburst*'s active database query language [Widom 1996]. The *create* clause specifies the name of the rule and the data source, the *if* clause contains a SQL query, and the *then* clause specifies an optional function that accepts the result of the SQL query for further processing (including serving as the input of another query). The dQUOB system can generate optimized query plans for the continuous queries presented in the system based on the relational model and allows user-specified adaptation for changes in data streams.

Most work on streaming data, including XSQ, assumes that the input consists of only the raw data. In this environment, certain limitations are unavoidable. For example, it is easy to devise XPath queries and sample inputs for which an unbounded amount of buffering is required for any XPath processor that produces exact results. An interesting alternative to this environment is one in which the input provides some assistance to the query processor by specifying constraints on forthcoming data or some other similar hints. For example, [Tucker et al. 2003] describes a method for embedding *punctuations* in streaming data, facilitating the streaming evaluation of queries that include blocking operators such as *group by*. It should be interesting to use similar ideas for streaming XML to support XPath queries that include traversal axes such as *following*.

8. SYSTEM ARCHITECTURE AND IMPLEMENTATION

We have implemented the XSQ system in Java using Sun Java SDK version 1.4. The code is publicly available (GNU GPL terms) at `http://www.cs.umd.edu/projects/xsq/`. The architecture of the XSQ system is depicted in Figure 25. The single arrows denote streaming data transfer between components at runtime (query execution time). The double arrows denote the flow of information during compile time. The XSQ system generates the HPDT corresponding to a given XPath as follows. The XPath query is parsed by the **XPath parser** into a sequence of

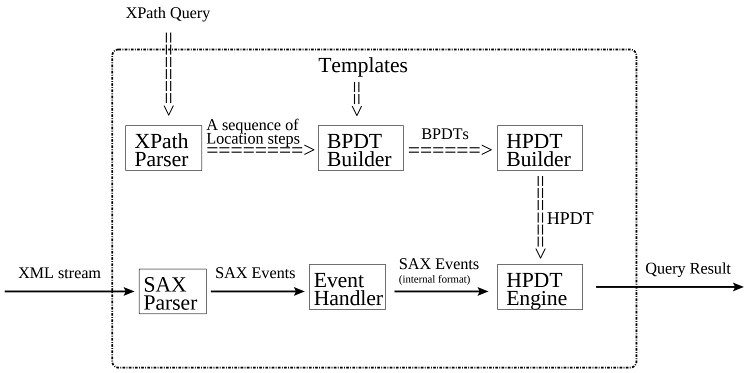


Fig. 25: XSQ system architecture

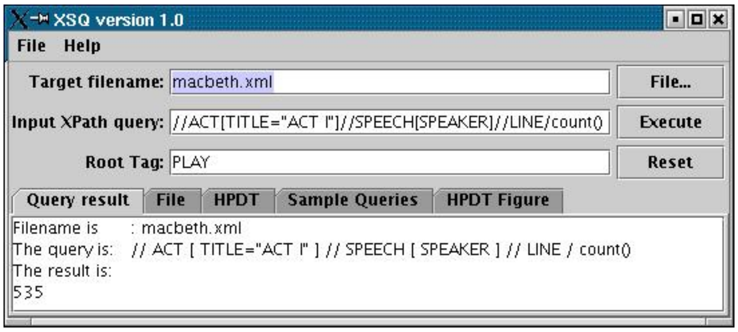


Fig. 26: Screenshot of the XSQ system

location steps. Each step consists of an axis, a node-test, and a predicate in the form of *object op const*. The predicate may be null. The object of a non-null predicate falls in one of the five categories we summarized in Section 5.2. Based on the category of the object, the **BPDT builder** builds a BPDT for each location step by instantiating the template for its category. The BPDT builder first creates the root BPDT which is denoted by $bpdt(0,0)$. For the ith location step, it starts with 2^i identical BPDTs and assigns each copy a unique ID (i,k). If $k \neq 2^i - 1$, i.e., the $bpdt(i,k)$ is not the left most BPDT in the layer, the *flush* operation in the templates should be modified to *upload* operation. It also adds a self-closure transition to the START state of the current BPDT and modifies the existing transitions if the axis in the location step is a closure axis. (The details of these modifications are described in Section 6.3.) The set of BPDTs is stored indexed by their source states. Each state stores the set of transitions emerging from it as a set indexed by the targets of the transitions. For each transition arc, we store the target state, the predicate, the buffer operations, and the type of the transition (self-closure, closure, regular, or catch-all). We thus obtain an array of BPDTs in which the $bpdt(l,k)$ is stored at the offset $2^l + (k-1)$.

The **HPDT builder** connects all the BPDTs into one HPDT by assigning a unique state ID for each state in all the BPDTs. For the TRUE state of the $bpdt(l,k)$, it will be assigned the same state ID as the START state of $bpdt(l+1, 2k+1)$. For the NA state of $bpdt(l,k)$, it will be assigned the same state ID as the START state

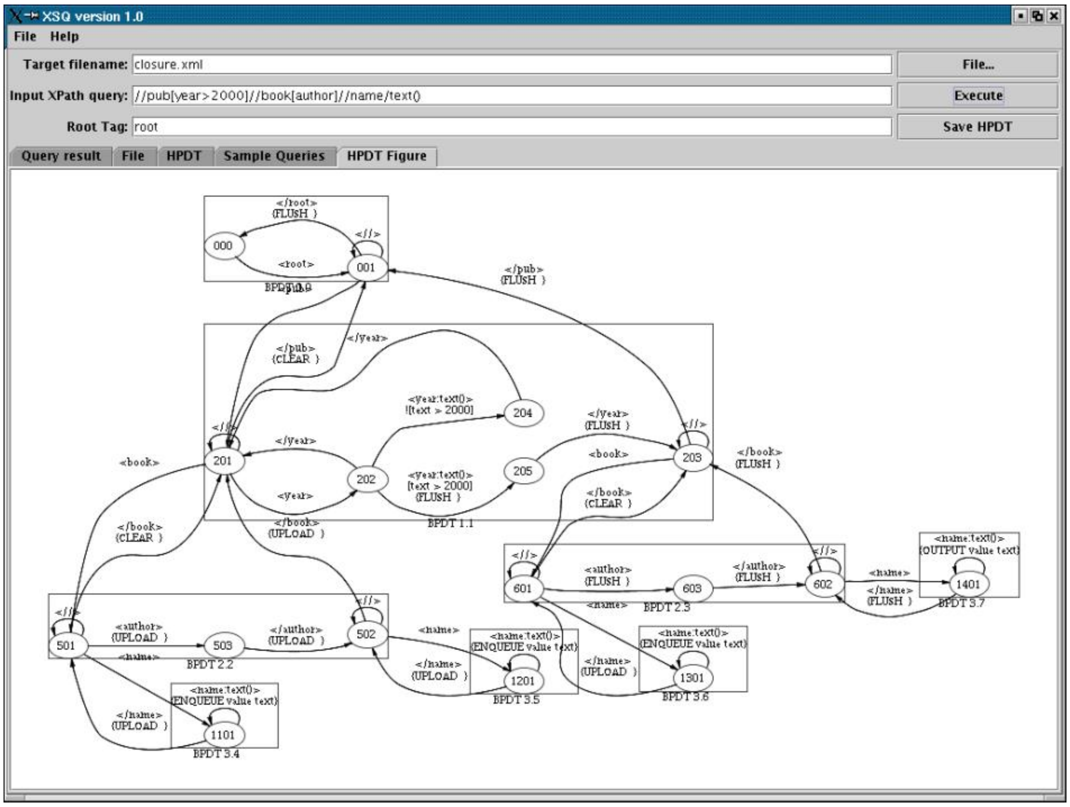


Fig. 27: Screenshot of XSQ displaying a HPDT

of $bpdt(l+1, 2k)$. The transition arcs from the two states are combined. However, each transition arc stores the ID of the original BPDT to which it belongs. After such assignment, all the states in the HPDT are stored in a single set.

All the tasks described above are performed offline when the query is issued (query compilation time). At runtime, the **HPDT engine** is responsible for executing the HPDT specification produced by the HPDT builder. It maintains the active states, stack, buffers, and other runtime objects associated with the HPDT. When the HPDT engine reads an HPDT specification, it first creates a global queue that is used for storing all items (raw content, without depth stacks) that need to be buffered in FIFO order. It also creates an array of buffers whose items are references to the items in the global queue (with depth stacks). The buffer of the $bpdt(l, k)$ is stored at the offset $2^l + (k-1)$ in the array. When the processing of streaming data begins, the HPDT has the start state with the depth stack (0) as the active state. Whenever a transition arc in $bpdt(l, k)$ is executed, the buffer operations defined for this arc, if there are any, operate on the buffer at the offset $2^l + (k-1)$ in the array of the buffers.

The streaming input to the HPDT engine comes from the **SAX parser**, which generates a sequence of SAX events in response to the incoming XML data. We use the SAX interface of the *Xerces* parser [XER 2000]. For each event, it calls a user-defined **event handler** that processes the event. Our event handlers first record the depth of the current event in order to support the event structure described in Section 2.2 (since the standard SAX API does not provide the information). It also performs the stack operations and validations since these operations are the same for all XML data. If the XML data is well-formed, it forwards the event to the HPDT engine in the XSQ format, which is a quadruple of *(tag, attrs, type, depth)*.

In order to facilitate our experimental evaluation of the effects of different XPath features, we have implemented two versions of XSQ: **XSQ-NC** supports multiple predicates and aggregations, but not closures. **XSQ-F** supports closures in addition to multiple predicates and aggregations. Figure 26 depicts a screenshot of the graphical interface of the XSQ-F system. The screenshot displays the result of the query `//ACT[TITLE="ACT I"]//SPEECH[SPEAKER]//LINE/count()` on the *macbeth.xml* file from the SHAKE dataset, which contains XML versions of some of Shakespeare's work [Bosak 2002]. The query illustrates the use of aggregation functions; it returns the number of lines that occur as descendants of a SPEECH element with a SPEAKER child in Act I of the play. Figure 27 depicts another screenshot of the interface. The query is one used in Example 9: `pub[year>2000]//book[author]//name/text()` The dataset has structure similar to that depicted in Figure 2. As indicated by the figure, in addition to query results, XSQ produces a graphical representation of the HPDT it uses for query processing. (We use the Graphviz package [Gansner and North 2000] for rendering the HPDT.)

The conceptual data structures introduced in earlier sections are implemented using more efficient low-level mechanisms in several instances. For example, depth stacks are stored as integers and operations on the depth stacks are implemented as fast bitwise operations on the integer representations. For example, if the depth stack is $(1, 2, 5)$, the integer representation is 11001. That is, the i'th bit is set if and only if the depth stack contains i. This representation is unambiguous because the depth stack consists of monotonically strictly increasing numbers (reading the stack bottom to top). Thus, the depth stacks use very little memory and operations on them incur very little overhead. We use long integers (64 bits) for this purpose. In order to support data with depth greater than 64, we can switch to using a pair of long integers. (Currently, this switch requires a recompilation of the HPDT engine module.)

Another implementation optimization is that used for buffers. There is only one copy of any data item in a global queue. The separate buffers of each BPDT only store references to this copy. Since we are using the references, we can mark the item in the global queue with an output flag when one BPDT determines that the item should be output. If there are several transitions processing the item, the other operations can be ignored. (Some of these may call for dequeuing the item; however, from the existential semantics of predicates in XPath it follows that the item belongs in the result.) Moreover, the document order of items is maintained automatically since we always output from the head of the global queue; that is, even if an item is flagged for output, it is not sent to output until it becomes the head of the global queue. Given this guarantee, the references of the items in one buffer can be grouped based on their depth stacks regardless of their document order, which will be maintained in the final result. Therefore, when a buffer operation operates on the current buffer, it compares the desired depth stack (according to the states involved in the transition) with the depth stacks of the items group by group instead of going through the items one by one.

9. EXPERIMENTAL EVALUATION

In this section, we summarize the results of our experimental evaluation of XSQ. We begin by describing our experimental setup in Section 9.1. Next, we study the two main performance metrics: throughput in Section 9.2 and memory usage in Section 9.3. Section 9.4 presents a broader study of a set of query engines aimed at characterizing their features and performance. In Section 9.5 we present a detailed experimental characterization of XSQ.

9.1 Experimental Setup

We conducted our experiments on a PC-class machine with an Intel Pentium III 900MHZ processor with 1 GB of main memory running the Redhat 7.2 distribution of GNU/Linux (kernel 2.4.9-34). The maximum amount of memory the Java Virtual Machine (JVM) could use was set to 512 MB. For the purpose of comparison, we selected a set of systems that process XPath or XPath-like queries. These systems are outlined in Figure 28. As the figure suggests, these systems vary considerably in their design goals and features, and many do not support streaming. We have discussed Galax [Fernandez and Simeon 2002] (version 0.1α), XQEngine [Katz 2002] (version 0.56), XMLTK [Avila-Campillo et al. 2002] (version 0.9), Saxon [Kay 2002] (version 6.5.2), and Joost (version 20020828) [Becker 2002] in Section 7. Some systems use query languages that are supersets or variations of XPath. For such systems, we issued queries that are equivalent to the XPath queries in our experiments. In many cases, the results are enclosed by different container elements but the contents are the same.

One of the goals of our experimental study is comparing different systems for the throughput and the memory usage, which are very important metrics of a query engine. However, we also wish to characterize these XPath processors in terms of the relation between the performance and the underlying features of the systems. We wish to gain insights into the cost to supporting certain XPath features such as closures and to study which systems and features are best suited to a given environment. For example, if we only want to use a simple XPath fragment without predicates, we do not need a full-flavored XQuery engine such as Galax. However, if we need to express complicated queries that involve joins or constructing new elements, we need to use systems such as Galax.

In our experiments, we use both real and synthetic **datasets** that differ in size and characteristics. We use four real datasets [Avila-Campillo et al. 2002]: an XML-ized version of Shakespeare's plays (SHAKE); the NASA ADC XML dataset (NASA) [Borne 2002], bibliographic records from the DBLP site (DBLP) [Ley], and the PIR-International Protein Sequence Database (PSD) [Wu et al. 2002]. We also use synthetic datasets that are generated using IBM's XML Generator [IBM 2001] and Toxgene [Barbosa et al. 2002]. Since the real datasets have relatively shallow structures, we generated two datasets using IBMGEN with deeper document structure to explore features related to such data. They are named as *RECURS* and

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	

Fig. 28: System Features

Name	Size (MB)	Text size (MB)	Number of elements (K)	Avg/Max depth	Average tag length	Parsing Time(s) Xerces	Parsing Time (s) Expat
SHAKE	7.89	4.94	180	5.77/7	5.03	1.42	0.43
NASA	25.0	15.1	477	5.58/8	6.31	4.35	1.50
DBLP	119	56.4	2,990	2.90/6	5.81	27.6	7.53
PSD	716	286	21,300	5.57/7	6.33	170	66.4
RECURS	10.4	8.78	95.6	22.3/26	5.31	1.65	0.43
RECURB	121	105	963	26/30	5.31	13.0	4.82

Fig. 29: Dataset Descriptions

RECURB. Some characteristics of these datasets, such as size, number of elements, depth, and parsing time are listed in Figure II.

For a text-based data format such as XML, **parsing** the input is often a substantial component of the running time. The parsing times listed in Figure II are generated using two parsing programs, named **PureParsers**, in C and Java. The PureParser in C uses the Expat 1.2 parser that is used by XMLTK. The PureParser in Java uses Xerces 1.0 for Java, which is used in XSQ-NC, XSQ-F, XQEngine, Saxon, and Joost in the experiments. The PureParsers parse the XML data but do nothing else. We note that the C parser is generally faster than Java parser since parsing involves a large number of string operations, which are implemented more efficiently in C. For example, for the 119MB DBLP dataset, the C PureParser finishes parsing in 7.53 seconds and the Java PureParser uses 27.6 seconds.

In our experiments, we executed each query on a dataset 30 times to get the mean value of the result we need. We also computed the 95% **confidence intervals** of the values to make sure our comparisons are statistically significant. We found that in all cases the 95% confidence interval is of width less than 1% of the mean value being measured (throughput, memory usage, etc.). Since it is difficult to display this small interval graphically, the usual error-bars are omitted in the graphical results that follow.

9.2 Throughput

We measure throughput as the rate at which a streaming query engine consumes input data (megabytes per second). Since this rate may vary over time (perhaps depending on the structure of the data, or as a result of periodic reorganization of data structures in a streaming system), we measure the average throughput as the size of the input divided by the time required to process it. Although this measure of throughput is useful for understanding the end-to-end performance of

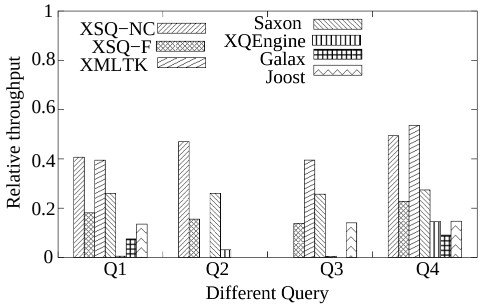


Fig. 30: Relative throughputs for different queries on the SHAKE dataset

Q1: `/PLAY/ACT/SCENE/SPEECH/SPEAKER/text()`

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

Q3: `//ACT//SPEAKER/text()`

Q4: `/PLAY/TITLE/text()`

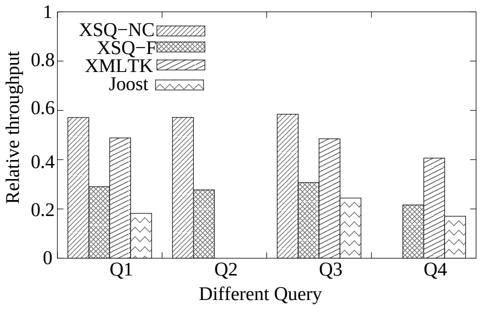


Fig. 31: Relative throughputs for different queries on the DBLP dataset

Q1: `/article/title/text()`

Q2: `/article[year>1990]/title/text()`

Q3: `/article{@key}`

Q4: `//title`

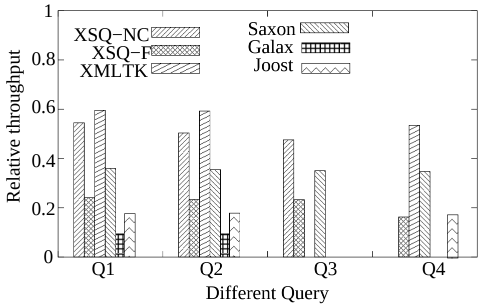


Fig. 32: Relative throughputs for different queries on the NASA dataset

Q1:`/dataset/reference/source/other/title/text()`

Q2:`/dataset/title/text()`

Q3:`/dataset[altname@type='ADC']/title/text()`

Q4:`//dataset//title/text()`

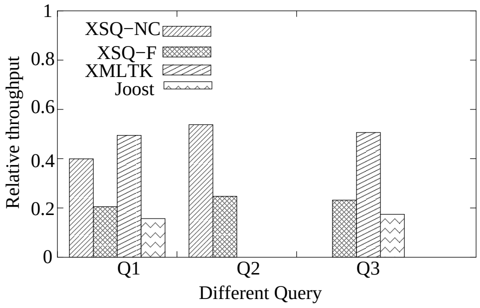


Fig. 33: Relative throughputs for different queries on the PSD dataset

Q1:`/ProteinEntry/reference/refinfo/authors/author/text()`

Q2:`/ProteinEntry[sequence]/protein/name/text()`

Q3:`//sequence`

a streaming query engine, it is not a good metric for our goal of understanding the ramifications of different system designs and features for two reasons. The first is that the systems we study use different programming languages and environments, and different parsers. Since the performance of the parser is a dominant factor in the performance of XPath processors, results based on only end-to-end throughput measurements are likely to be determined more by the features of the parser and programming language libraries than by the query engine proper. The second reason is that different datasets may lead to different parsing performance for the same parser. We can see in Figure II that the parsing times are influenced by not only the size of the file, but also by the number of elements in the file. For the two datasets DBLP and RECURB, the sizes are similar but the parsing times differ substantially since DBLP has more elements than RECURB. In order to study the throughput of query engines on different datasets, it is important to factor out the effects of the varying difficulties of parsing such datasets.

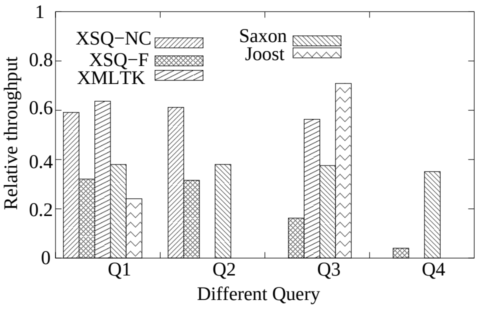


Fig. 34: Relative throughputs for different queries on the RECURS dataset
Q1:`/pub/book/title/text()`
Q2:`/pub/book[year]/author[email]/name/firstname/text()`
Q3:`//proceedings/@category`
Q4:`//pub[year=14]//paper[@id=13]/title`

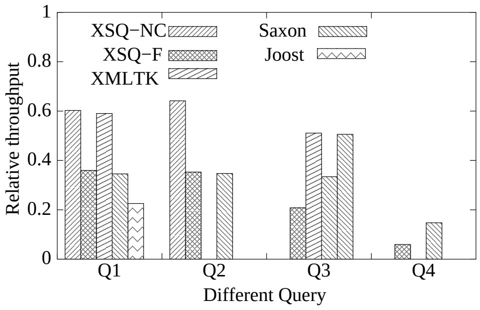


Fig. 35: Relative throughputs for different queries on the RECURB dataset
Q1:`/pub/book/title/text()`
Q2:`/pub/book[year]/author[email]/name/firstname/text()`
Q3:`//proceedings/@category`
Q4:`//pub[year=14]//paper[@id=13]/title`

Since all the systems in Figure 28 use the SAX API to parse the data, the throughput of the PureParser, which parses the data but does nothing else, gives an upper bound of the throughput for any XML query system. Therefore, instead of comparing systems using their raw throughput, we compare them using their throughput normalized with respect to their parsers. That is, we define **relative throughput** to be the throughput of the complete system divided by the throughput of the parser used by that system. Note that Galax implements its own parser in OCaml. Since we were unable to find a SAX parser implemented in OCaml, we used the Java PureParser to normalize the throughput of Galax. However, we believe that the OCaml parser is faster than the Java PureParser; thus this switch does not put Galax at a disadvantage.

Figures 29, 30, 31, 32, 33, and 34 summarize our experiments comparing the relative throughputs of the systems over different datasets and queries. Results for several combinations of queries and datasets are missing for one or more systems because either the system does not support queries with certain features (e.g., closures, predicates) or the dataset is too large for the implementation. For example, XMLTK, Galax, and Joost do not support query Q2 in Figure 29. Similarly, many systems do not work with the large PSD dataset of the experiment summarized by Figure 32.

We observe that, in general, XMLTK and XSQ-NC are the fastest two systems when we use simple queries that they support. However, since XMLTK does not handle predicates and XSQ-NC does not handle closure axes, they can use more efficient methods for query evaluation. One reason for this efficiency is that they do not need to handle the multiple matchings between the query and the data. Therefore, they have fewer extra operations for each element. Another reason is they use deterministic automata. The HPDT used in XSQ-NC is deterministic, which means there is only one current state at any point in time. For each input event, there is at most one transition arc that accepts the input for the current state. Therefore, even when processing the same query without closure, XSQ-NC is faster than XSQ-F since XSQ-F uses a non-deterministic automaton. For example, when

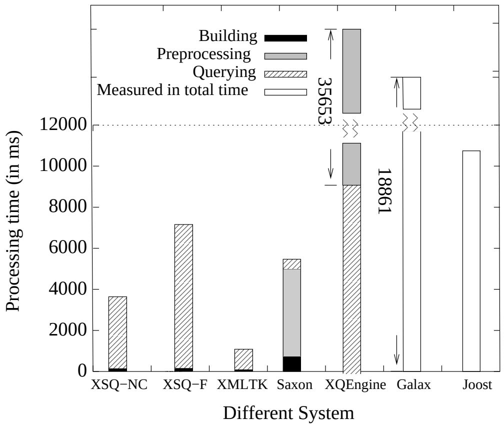


Fig. 36: Preprocessing time, query processing time, and total querying time
Dataset: SHAKE Query: `/PLAY/ACT/SCENE/SPEECH/SPEAKER/text()`

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

searching for a matching transition in the automaton, XSQ-NC can stop searching after it finds one match. In contrast, XSQ-F has to go through all the transition of the current state to make sure every transition is handled.

Figures 29, 31, and 33 suggest that Saxon is faster than XSQ-F when they process XML data that can fit 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 processing in main memory. In-memory processing is efficient 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 four to five times the size of the dataset. Thus, it is difficult to scale the Saxon approach to large XML files and to streaming data.

Figure 35 summarizes our experiments measuring the components of the overall query-processing time. The dark bar represents the query compilation time, which usually includes parsing the query and building the data structures used by the run-time query engine. The gray bar represents the preprocessing time. For example, the preprocessing stage of Saxon loads all the data into memory to build the DOM tree before it can evaluate the queries. Similarly, XQEngine preprocesses data by building a full-text index on the data before evaluating any queries. Figure 35 highlights an important advantage of streaming systems: They return results incrementally while still reading the input. The availability of some results early is a useful feature in general, and especially important when the input data stream is unbounded or very large. The non-streaming systems have to wait until all the preprocessing finishes before they can begin evaluating the queries. However, as long as the preprocessed data in these systems remains in memory, subsequent queries can be evaluated very efficiently by reusing the preprocessed data.

9.3 Memory Usage

The main memory required by a streaming query engine is an important metric and often determines the feasibility of using that engine for a dataset. Further, it is often possible to increase throughput by increasing the memory footprint. Figures

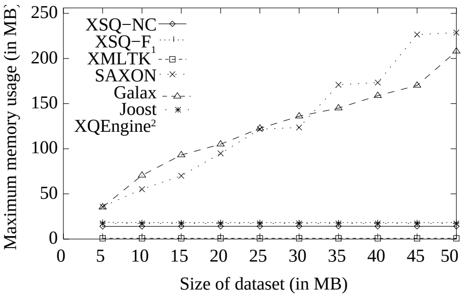


Fig. 37: Memory usage for DBLP-based datasets of different sizes
Query:`/dblp/inproceedings[author]/title/text()`
1. The query for XMLTK :`/dblp/inproceedings/title/text()`
2. XQEngine could not be tested because it currently supports only 32K elements per document.

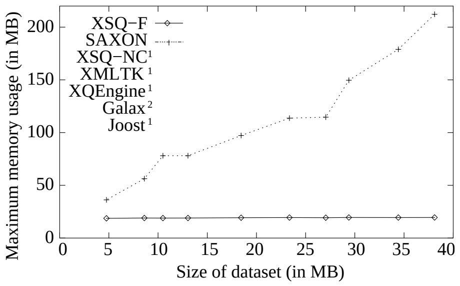


Fig. 38: Memory usage for synthetic datasets of different sizes
Query:`//pub[year]//book[@id]/title/text()`
1. The system cannot handle the query in the dataset.
2. Galax reports a "stack overflow" error when we try the query.

36, 37, 38, 39, 40, and 41 summarize the results of our experiments comparing the memory used by the systems we study. We observe that, as expected, the streaming systems typically use much less memory than the non-streaming systems. For example, although Saxon (a non-streaming system) is faster than XSQ-F when both systems handle the queries in these figures (Figures 31, 33, and 34), it also uses much more memory than XSQ-F. We also note that, for different datasets, the streaming systems use almost the same amount of memory. This fact suggests that the streaming systems need a small amount of memory which is only weakly dependent on to the size of the datasets in our study. For systems such as XMLTK and Joost, which do not support predicates, this observation is always true since they do not buffer anything in the data. However, systems that support predicates, such as XSQ-NC and XSQ-F must buffer data and the amount of buffered data may be large, depending on the dataset and query. Recall, however, that any data buffered by XSQ must also be buffered by any streaming query engine for XPath. That is, the need for a potentially large amount of buffering in this case is a result of XPath features and not system design. Further experiments studying this aspect of XSQ are described in Section 9.5.

Memory usage is also an important determinant of the scalability of streaming systems. Since non-streaming systems need to load the whole dataset into memory, they need memory that grows at least linearly with the size of the input. In contrast, streaming systems need to store only a small fraction of the stream. Figure 36 shows the memory usage reported for the queries over datasets ranging in size from 5MB to 50MB. All the datasets are excerpts of the DBLP dataset. For example, the 10MB dataset contains the first 10MB data of the DBLP dataset. (The size is approximate since we need to include the closing tags of elements near the 10MB offset in order to obtain well-formed XML.) Figure 36 indicates that Saxon and Galax use memory roughly linear in the size of the input data. Linear growth in memory usage, with a constant factor of 4 to 5, makes DOM-based systems unsuitable for large XML files.

We also used the XML Generator program to generate datasets of varying size

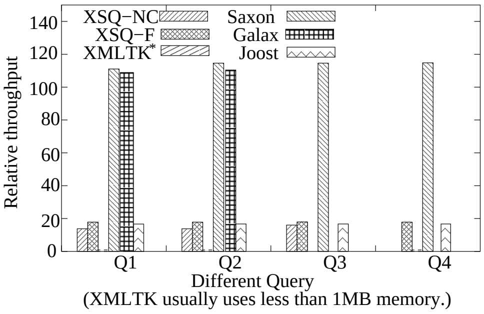


Fig. 39: Memory usage for different queries on the NASA dataset

Q1:`/dataset/reference/source/other/title/text()`

Q2:`/dataset/title/text()`

Q3:`/dataset[altname@type='ADC']/title/text()`

Q4:`//dataset//title/text()`

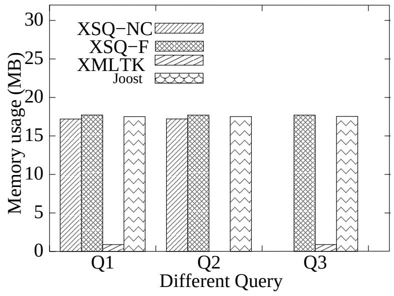


Fig. 40: Memory usage for different queries on the PSD dataset

Q1:`/ProteinEntry/reference/refinfo/authors/author/text()`

Q2:`/ProteinEntry[sequence]/protein/name/text()`

Q3:`//sequence`

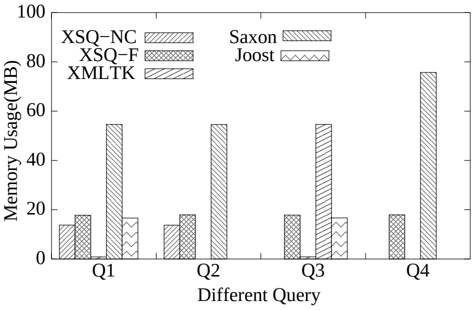


Fig. 41: Memory usage for different queries on the RECURS dataset

Q1:`/pub/book/title/text()`

Q2:`/pub/book[year]/author[email]/name/firstname/text()`

Q3:`//proceedings/@category`

Q4:`//pub[year=14]//paper[@id=13]/title`

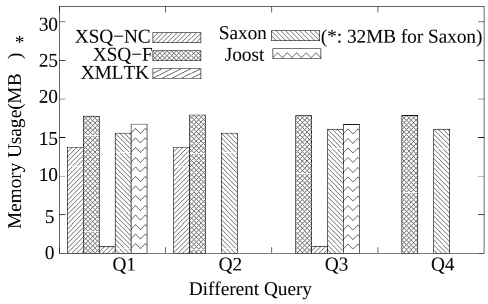


Fig. 42: Memory for different queries on the RECURB dataset

Q1:`/pub/book/title/text()`

Q2:`/pub/book[year]/author[email]/name/firstname/text()`

Q3:`//proceedings/@category`

Q4:`//pub[year=14]//paper[@id=13]/title`

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 37 we note that even with highly recursive data and queries with closures, the memory used by XSQ-F is constant. Recall, from Section 6, that XSQ-F 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 specified in the first location step (when the HPDT returns to the highest layer BPDT), the maximum amount of memory the XSQ needs does not exceed the maximum size of the elements in the stream.

9.4 Characterizing the XPath Processors

Recall, from our discussion in Section 6, that XSQ and other streaming query engines need to buffer data items when they cannot immediately decide whether the data items belong to the result. In general, the relative ordering in a dataset of XML elements to which a query refers influences the amount of buffering required for that dataset. In order to study this effect, we generated a 10 MB dataset using Toxgene, by applying the following template repeatedly to generate new a elements with successive id attributes.

```
<a id="1">
   <prior> 1 </prior>
   <foo> 1 </foo>
   <!--- 10,000 foo elements --->
   <foo> 1 </foo>
   <posterior> 1 </posterior>
</a>
```

We evaluated the following three queries on this dataset:

```
Q1:  /a[prior=0]
Q2:  /a[posterior=0]
Q3:  /a[@id=0]
```

All three queries have empty results on the above dataset because their predicates, which test for an text contents or attributes with value 0, are not satisfied by the test data, in which all content has value 1. However, the queries differ in the location of the data item used in the predicate relative to the data item to which the predicate applies.

Figure 42 summarizes the results of running XSQ-NC, XSQ-F, and Saxon on these queries. (XMLTK and Joost cannot handle queries that need explicit buffering of the data. Galax reports an "Internal Error" when evaluating the queries on the synthetic data. XQEngine is not tested in the following experiment since the version we use can process only XML files that have less than 32,767 elements.) We observe that the throughput of the Saxon system is essentially the same for all three queries. This result is not surprising because Saxon 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 a query, the document order of the elements traversed is not important. However, the throughput of XSQ-NC is 30% higher for Q3 than for the other two queries. When processing Q3, XSQ-NC is able to determine at the beginning of the a element that all the contents in this element can be ignored. For the other two queries, on the other hand, the content of every a element must be buffered because the prior and posterior elements may occur anywhere before the closing tag of the a element. We also observe that XSQ-F is not as sensitive as XSQ-NC to the element order. Recall from Section 6 that even if XSQ-F determines that an item is in the result set, it cannot output the item right away because there may be items in the global queue whose memberships in the result are as yet undetermined and that lie ahead of this item in the queue. Thus, XSQ-F must first mark the item with an output flag and check if the item is the head of the global queue. This process of marking and checking every result item slows down the XSQ-F system and reduces its sensitivity to the order of the elements. (However, this process is necessary since the closure axes in the query imply that the result membership of items in the buffer cannot always be determined in document order.)

We also studied the sensitivity of system throughput to the size of the query result. The degree to which system throughput depends on result size varies across

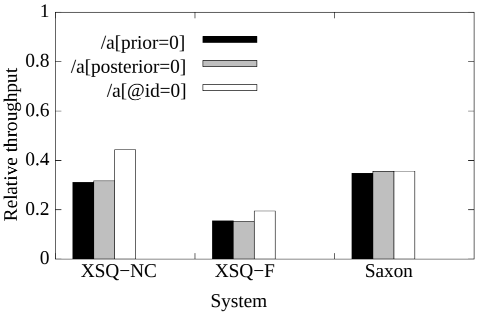


Fig. 43: Effect of data ordering on throughput

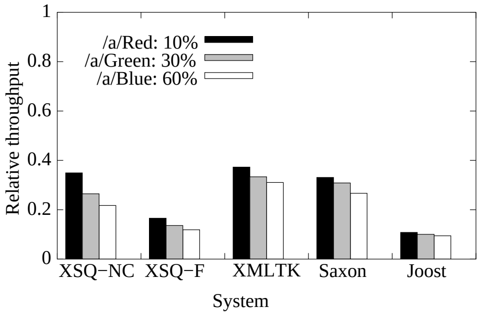


Fig. 44: Effect of the result size on throughput

the systems we studied. For example, the XQEngine is slower than the other systems in Figure 35 because the query returns a large portion of the dataset. However, if the query contains a tag that is not in the data, XQEngine returns the empty result set very quickly because it has access to an inverted file index on tags. The other systems, lacking such an index, spend similar amount of time on the query irrespective of whether the tags in the query appear in the document.

We used Toxgene to generate a test dataset of 10 MB consisting of a mix 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 single character. We used this dataset with three queries: `/a/red`, `/a/green`, and `/a/blue`, generating query results that are roughly 1 MB, 3 MB, and 6 MB in size, respectively. Figure 43 indicates the relative throughputs of the systems on these queries. (XQEngine and Galax are not tested for the same reason as described in the previous experiment.) We observe that XSQ-NC's throughput is quite sensitive to the size of the result. 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 constitute a large portion of the running time of XSQ-NC. We also note that XSQ-F is not as sensitive as XSQ-NC. As described in Section 6, XSQ-F always keeps the item first, irrespective of whether it is in the result, and checks the queue after all transition arcs are handled. The difference between the treatment of elements in and not in the result is therefore not as large as in XSQ-NC. Saxon's throughput is not very 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. Similarly, the low sensitivity of XMLTK's throughput to the result size is because the difference is only in the time required to output the result. However, it is not clear why Joost's throughput is not more sensitive to the result size.

9.5 Characterizing XSQ-F

In this section, we study the effect of different query features on the performance of XSQ-F. In particular, we study the effect of the number of closure axes in the query, the number of predicates in the query, and the length of the query.

In the first experiment, we executed a set of queries that return the same result

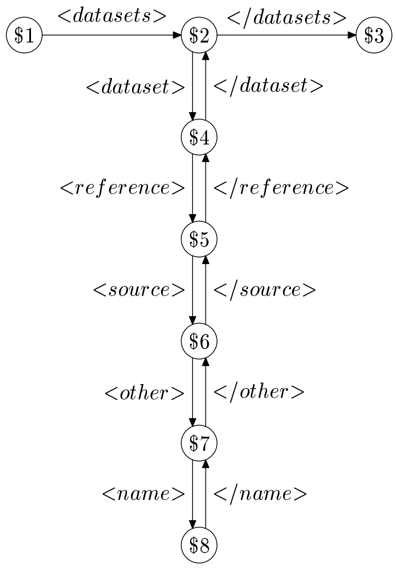
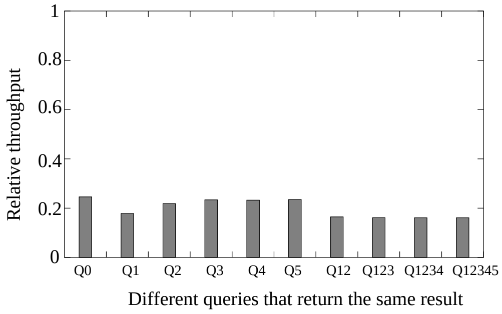


Fig. 45: HPDT generated for query /dataset/reference/source/other/-name/text()



Q0:`/dataset/reference/source/other/name/text()`
Q1:`//dataset/reference/source/other/name/text()`
Q2:`/dataset//reference/source/other/name/text()`
Q3:`/dataset/reference//source/other/name/text()`
Q4:`/dataset/reference/source//other/name/text()`
Q5:`/dataset/reference/source/other//name/text()`
Q12:`//dataset//reference/source/other/name/text()`
Q123:`//dataset//reference//source/other/name/text()`
Q1234:`//dataset//reference//source//other/name/text()`
Q12345:`//dataset//reference//source//other//name/text()`

Fig. 46: Effect of closure axes in the queries on NASA dataset

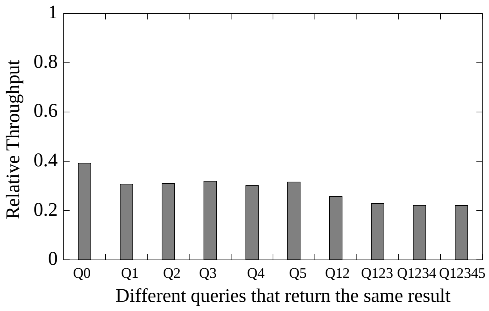


Fig. 47: Experiment of Figure 45 using a modified NASA dataset

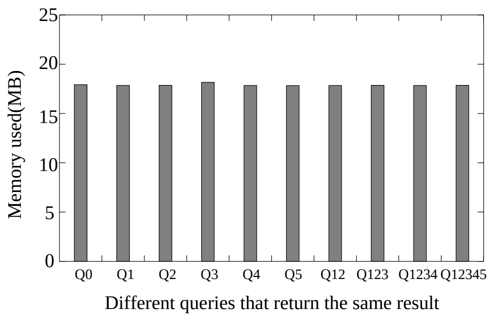


Fig. 48: Memory usage of queries with closure axes on NASA dataset

set but with different number of closure axes in the query. In Figure 45, Q_S, where $S \subseteq \{1,2,3,4,5\}$, is the query in which the ith location step has a closure axis for all $i \in S$. For example, the query Q_{123} has closure axes in the 1st, 2nd, and 3rd location steps. (The remaining location steps have the child axis.) All these queries return the same result when applied to the NASA dataset. The memory used by XSQ-F when processing these queries is summarized in Figure 47. The HPDT generated for the query `/dataset/reference/source/other/name/text()` is depicted in Figure 44. The HPDTs for other queries have a similar structure, with self-closure transitions and closure transitions in the appropriate places, following the scheme of Section 6.3.

Figure 47 indicates that although the number of closure axes and their locations vary among the queries, resulting in varying sizes of the set of current states, the memory used for the different queries does not vary much. As discussed earlier, this insensitivity is due to the fact that the memory used by the HPDT states is only a very small amount in the total memory used by the system. The buffers and other system components are responsible for most of the memory usage.

Figure 45 summarizes the throughput on the above queries. We observe that the throughput is lower for queries with a closure axis in the first location step than for queries with a child axis in the first location step. (The differences in the histogram bars, though small, are statistically significant; here, as in our other experimental results the 95% confidence intervals are smaller than 1% of the values shown.) From the DTD of the dataset [Borne 2002], we know that all the top level elements in the NASA dataset are `dataset` elements. If we have closure axis in the first location step, then after the HPDT (Figure 44) makes the transition from state \$2 to \$4, it will also keep state \$2 in its current state set. Then, the HPDT needs to check whether each incoming event is a `dataset` element, which involves string comparisons. In contrast, if the first location step has a child axis, state \$2 does not remain current. Therefore, only for all the subelements of the `dataset` elements does the HPDT check the begin events by comparing the name of the element with the label. It ignores all elements that are not descendants of both `dataset` and `reference` by simply checking the depth of those events, an operation much faster operation than the string comparison used for the earlier case.

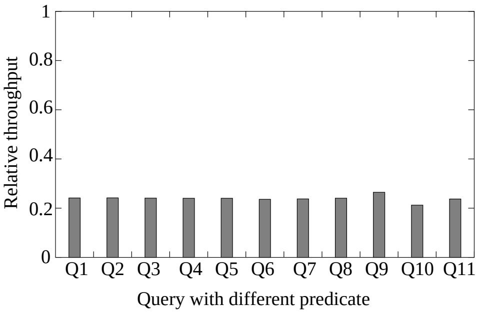
It is not the position of the closure axes in the query alone that determines the throughput. On examining the dataset closely, we note that the evaluation time is significantly affected by the *selectivities* of each location step. Consider the i'th location step of a query and let S be the set of elements that match the first $i-1$ location steps. Let S' be the children of nodes in S. We define the selectivity of location step i (for a given dataset) to be the fraction of the nodes in S' that match the first i location steps. If the i'th location step uses the closure axis, we use descendants instead of children in identifying the set S' in this definition. For the query and dataset of this experiment, each `dataset` element contains one `reference` child, which corresponds to 10%–20% of the total number of events for one `dataset` element. We also ran these queries on a dataset obtained by removing all subelements of `dataset` elements other than the `reference` subelements (which means the selectivity of the second location step changed from around 20% to 100%). The result is summarized in Figure 46. We can see that the closure axis in the first location step no longer has a significant impact on the throughput. (The throughput of query Q1 is not significantly smaller than throughputs of query Q2, Q2, Q3, and Q5, all of which contain one closure axes but in different location steps.) The reason is that the extra work done by Q1 (checking descendants of subelements other than `reference`) on the original dataset no longer exists when the system evaluates the queries on the new dataset since the `dataset` elements in the new dataset have only `reference` subelements. In general, when the selectivity of a location step is small, closure axes preceding this step result in a performance penalty because the non-result descendants cannot be eliminated by depth comparisons and incur the cost of more expensive string comparisons.

In the previous experiment, we used queries with only closure axes but without predicates. We also performed an experiment using queries with predicates of different types and in different positions. The dataset used for this experiment is the NASA dataset. The results are summarized in Figure 48. (We abbreviate dataset as `d` in the queries; similarly, we abbreviate other tags by their first letter.) The first eight queries have the same result although they have different types and numbers of predicates. The last three queries have empty results. We note that the throughputs for the first eight queries are similar because the number of comparisons needed to determine the results of their predicates does not vary much across these queries. For example, although the `dataset` elements typically have several `altname` subelements, the first `altname` subelement usually has the attribute `type` that has value `ADC`. Therefore, the query Q3 and Q4 will both check the first `altname` subelement and ignore the remaining `altname` elements. However, for query Q10, although the result set is empty, resulting in less time spent on output operations, all the `altname` subelements of `dataset` elements must be checked. Therefore, its throughput is lower than those of queries Q3 and Q4. We also observe that the query Q9 has the largest throughput among all the queries used in the experiment. The reason is that the predicate in this query `[@subject=test]` can be evaluated to false at the beginning of the `dataset` elements. Thus, all the descendants of the `dataset` elements can be ignored. This experiment demonstrates that XSQ is able to save on comparisons for predicates that have already been evaluated.

In the next experiment, we used queries of different lengths (query sizes). The results are summarized in Figure 49. The query Q5 and Q6 return the same result set of size 747 KB and the others return the same result set of size 16.7 KB. The bars in Figure 49 plot relative throughput: striped bars for queries with no predicates, gray bars for queries with a predicate in every location step, and white bars for queries with a predicate in only the first location step. The predicates all evaluate to false. For example, for Q3 for the gray bar is for the query `//source[test]//other[test]//title[test]/text()` while for Q1 the white bar is for the query `//source[test]//other//title/text()`. The memory usage for these queries is shown in Figure 50. The figures indicate that queries with predicates in every location step use almost the same amount of memory as the queries without predicates. The throughputs are also similar.

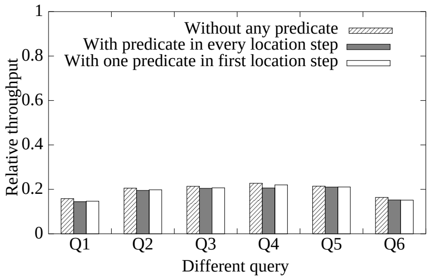
Although Figure 49 suggests that longer queries generally have lower throughputs, we notice an exception: Q6 has smaller throughput than Q4 and Q5 although it returns the same result set as Q5 and has the same query length as Q4. Q6 is slower than Q4 because the selectivity of the second location step of Q6 is much smaller than the selectivity of the second location step of Q4. (That is, the fraction of the descendants of `dataset` elements that have tag `title` is much smaller than the fraction of the descendants of `other` elements that have tag `title`.) Recall, from our discussion earlier in this section, that a location step with a low selectivity results in a larger number of string comparisons resulting from a closure axis in the previous location step. Thus, Q6 incurs a larger number of string comparisons than Q4, resulting in lower throughput.

We also note that Q5 has a higher throughput than Q6 because the HPDT



Q1:`/d[@subject=astronomy]/r/s/o/n/text()`
Q2:`/d[@subject]/r/s/o/n/text()`
Q3:`/d[altname]/r/s/o/n/text()`
Q4:`/d[altname@type=ADC]/r/s/o/n/text()`
Q5:`/d/r/s/o[publisher]/n/text()`
Q6:`/d[altname@type=ADC]/r/s/o[publisher]/n/text()`
Q7:`/d[altname]/r/s/o[publisher]/n/text()`
Q8:`/d[@subject]/r/s/o[publisher]/n/text()`
Q9:`/d[@subject=test]/r/s/o/n/text()`
Q10:`/d[altname@type=test]/r/s/o[publisher]/n/text()`
Q11:`/d/r/s/o[test]/n/text()`

Fig. 49: Effect of predicates in the queries on NASA dataset



Q1:`//dataset//reference//source//other//title/text()`
Q2:`//reference//source//other//title/text()`
Q3:`//source//other//title/text()`
Q4:`//other//title/text()`
Q5:`//title/text()`
Q6:`//dataset//title/text()`

Fig. 50: Effect of query length on throughput for the NASA dataset

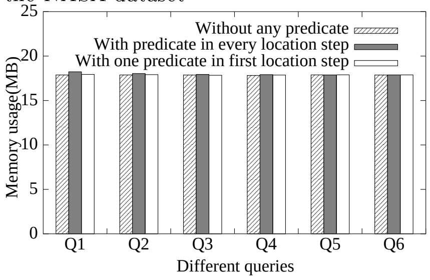


Fig. 51: Effect of query length on memory usage for the NASA dataset

evaluating Q5 has smaller current state set. The HPDT evaluating Q5 has one current state that stays active during the whole process (to check whether the next event is the begin event of a `title` element) while the HPDT evaluating Q6 has two current states that stay active (to check the begin events of the `dataset` and the `title` elements). Therefore, for the begin event of every descendant of the `dataset` elements, the HPDT for Q6 performs two string comparisons, while the HPDT for Q5 only performs one.

We noted in the Section 9.3 that the maximum amount of data that XSQ needs to buffer is no greater than the size of largest element in the input. To verify our implementation, we generated an XML file of size 31.5 MB, containing 11 top-level elements `chunk`. We put a `test` attribute within the open tag of each `chunk` element. All the `test` attributes have value 1. We also put two `test` subelements inside the `chunk` elements. The first one is put right after the open tag of each `chunk` element and its content is set to 0. The second one is placed right before the close tag of each `chunk` element and its content is set to 1. We ran two sets of queries on the dataset. The memory usage of the experiments is summarized in Figure 53. The first set of queries contains similar patterns but with different predicates in their first location steps. A location step `/a` is inserted between the

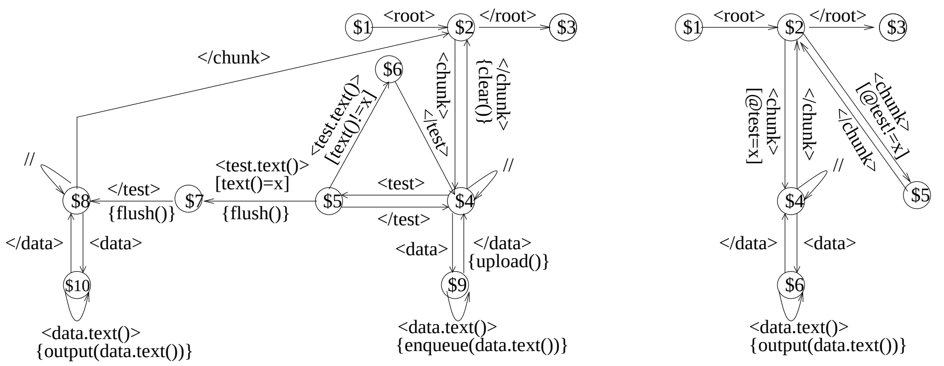


Fig. 52: HPDT generated for the query `/chunk[test=x]//data/text()` and `/chunk[@test=x]//data/text()`

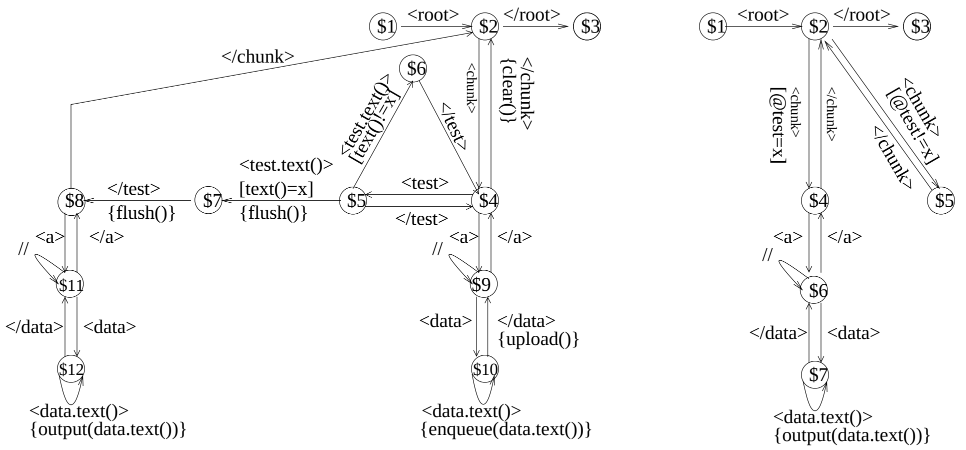
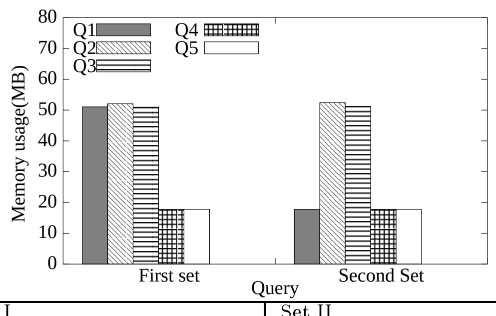


Fig. 53: HPDT generated for the query `/chunk[test=x]/a//data/text()` and `/chunk[@test=x]/a//data/text()`



	Set I	Set II
Q1	/chunk[test=0]//data/text()	/chunk[test=0]/a//data/text()
Q2	/chunk[test=1]//data/text()	/chunk[test=1]/a//data/text()
Q3	/chunk[test=2]//data/text()	/chunk[test=2]/a//data/text()
Q4	/chunk[@test=0]//data/text()	/chunk[@test=0]/a//data/text()
Q5	/chunk[@test=1]//data/text()	/chunk[@test=1]/a//data/text()

Fig. 54: Memory usage of queries

first location step and second location step of each query in the first set to form the queries in the second set. However, since each `chunk` element has only one `a` subelement and all the `data` descendants of the `chunk` element are inside the `a` subelement, the corresponding queries in the two sets always return the same result set on this synthetic dataset. Moreover, Q1, Q2, and Q5 of the two sets return the same result set of size 23.4MB while Q3 and Q4 of both query sets return the empty result set. The HPDTs generated for the first set queries are depicted in Figure 51, and the HPDTs generated for the second set of queries are depicted in Figure 52.

For the two sets of queries, Q2 and Q3 almost use the same amount of memory while Q4 and Q5 use much less memory, although Q2 and Q4 returns a result set of size 23.4MB and Q3 and Q5 return an empty result set. The memory usage for Q2 and Q3 is similar because both of them require buffering the text contents of all `data` subelements since the results of the predicates in both queries are determined only at end of every `chunk` element. (The results of query Q2 are sent to output while the results of query Q3 are cleared from the buffer.) Queries Q4 and Q5 use much less memory than Q2 and Q3 because neither requires the contents to be buffered since the results of their predicates are determined at the beginning of the `chunk` element. For Q4, the HPDT sends all the text contents of the `data` subelements directly to output, while for Q5 all the `data` subelements are discarded as they are encountered.

The difference in the memory usage of the two Q1 queries is due to the different structures of the corresponding HPDTs. If we follow the reasoning of the previous paragraph, it seems reasonable to expect that both Q1 queries have memory usage similar to that of queries Q4 and Q5 because the result of the predicates in the queries can be determined at the beginning of the `chunk` elements. However, it is clear in Figure 53 that the memory usage of Q1 in the first set is close to that of Q2 and Q3, while the memory usage of Q1 in the second set is close to that of Q4 and Q5. The HPDT generated from Q1 in the first set is depicted in Figure 51. Even when the predicate has been satisfied, this HPDT keeps the state $4 active because of the self-closure transition on $4. The HPDT continues to enqueue the text contents of the `data` descendants (using the enqueue operation on state $9), which will never be used because the same data item will be sent to output right away (by the output operation on state $10). However, since we cannot explicitly clear the buffer until the end of a `chunk` element, these items stay in memory until the end of the `chunk` element. Thus, the memory usage of this Q1 query is almost the same as that of Q2 and Q3. In contrast, the state $4 in the HPDT for Q1 in the second set (Figure 52) does not have a self-closure transition. Therefore, when the predicate has been satisfied, only state $8 is active. The text contents of the `data` elements will be only output by the operation on the state $12. The *enqueue* operation on state $10 will never be executed.

10. CONCLUSION

The XSQ system provides an efficient implementation of XPath for streaming XML data. It supports XPath queries that have multiple predicates, closure axes, and output functions that permit extraction of portions of the stream. We have illustrated the challenges posed by these XPath features to query processing in a streaming environment and described the soulution used by XSQ. All the methods described in this paper have been fully implemented in the XSQ system, which is freely available at `http://www.cs.umd.edu/projects/xsq/`. The implementation is based on a clean system design that centers on a hierarchical arrangement of pushdown transducers augmented with buffers and auxiliary stacks. A notable feature of XSQ is that at any point during query processing, the data that is buffered by XSQ must necessarily be buffered by any streaming XPath query engine. We have described the results of a detailed experimental study of XSQ and similar systems. In addition to demonstrating the ability of XSQ to maintain a high throughput with modest memory requirements, even for large datasets and complex queries, our experimental study provides a valuable characterization of the performance implications of XPath features and system designs, as embodied in the systems we studied.

Acknowledgment

We would like to thank Jerome Simeon and Mary Fernandez for providing the Galax system; Howard Katz for XQEngine; the XMLTK team (Iliana Avila-Campillo, Demi Raven, T.J. Green, Ashish Gupta, Yana Kadiyska, Makoto Onizuka, and Dan Suciu) for the XMLTK system and for pointers to datasets; Michael Kay for Saxon; Denilson Barbosa and Alberto Mendelzon for ToXGene; Angel Luis Diaz and Douglas Lovell the XML Generator; Oliver Becker for the Joost program and assistance with the code; the Graphviz team (John Ellson, Emden Gansner, Eleftherios Koutsofios, John Mocenigo, Stephen North, and Gordon Woodhull)for the Graphviz program used to display HPDTs; Mukund Raghavachari for bringing to our attention work on stream processing of backward axes; and Bertram Ludascher and Yannis Papakonstantinou for providing an early version of their paper on XSM.

REFERENCES

Abiteboul, S., Buneman, P., and Suciu, D. 2000. *Data on the Web*. Morgan Kaufmann.

Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. 1996. The Lorel query language for semistructured data. *Journal of Digital Libraries 1,* 1 (Nov.), 68–88.

Altinel, M. and Franklin, M. J. 2000. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *The 26th International Conference on Very Large Data Bases*. 53–64.

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

Avnur, R. and Hellerstein, J. M. 2000. Eddies: Continuously Adaptive Query Processing. In *The 19th ACM SIGMOD International Conference on Management of Data*. 261–272.

Barbosa, D., Mendelzon, A., Keenleyside, J., and Lyons, K. 2002. ToXgene: a template-based data generator for XML. In *The 5th International Workshop on the Web and Databases*. Madison, Wisconsin, 49–54.

Barton, C. M., Charles, P. G., Goyal, D., Raghavachari, M., Josifovski, V., and Fontoura, M. F. 2003. Streaming XPath Processing with Forward and Backward Axes. In *The 18th International Conference on Data Engineering*.

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

Becker, O., Cimprich, P., and Nentwich, C. 2002. Streaming Transformations for XML. `http://www.gingerall.cz/stx`.

BOAG, S., CHAMBERLIN, D., FERNANDEZ, M. F., FLORESCU, D., ROBIE, J., AND SIMEON, J. 2002. XQuery 1.0: An XML query language. W3C Working Draft. http://www.w3.org/TR/2002/WD-xquery-20021115/.

BORNE, K. D. 2002. ADC Dataset, GSFC/NASA XML Project. http://xml.gsfc.nasa.gov/archive/.

BOSAK, J. 2002. The Plays of Shakespeare in XML. http://www.oasis-open.org/cover/bosakShakespeare200.html.

BRAY, T., PAOLI, J., AND SPERBERG-MCQUEEN, C. 1998. Extensible markup language (XML) 1.0. World Wide Web Consortium Recommendation. Available at http://www.w3.org/TR/REC-xml.

BUNEMAN, P., DAVIDSON, S., HILLEBRAND, G., AND SUCIU, D. 1996. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Montréal, Québec, 505–516.

CARNEY, D., CETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. 2002. Monitoring Streams: A New Class of Data Management Applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*. 215–226.

CHAN, C. Y., FELBER, P., GAROFALAKIS, M. N., AND RASTOGI, R. 2002. Efficient Filtering of XML Documents with XPath Expressions. In *The 18th International Conference of Data Engineering*. 235–244.

CHANDRA, A. K., KOZEN, D. C., AND STOCKMEYER, L. J. 1981. Alternation. *Journal of the ACM (JACM) 28,* 1, 114–133.

CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. 2000. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *The 19th ACM SIGMOD international conference on Management of data*. 379–390.

CHEN, Z., JAGADISH, H. V., KORN, F., KOUDAS, N., MUTHUKRISHNAN, S., NG, R. T., AND SRIVASTAVA, D. 2001. Counting Twig Matches in a Tree. In *The 17th International Conference of Data Engineering*. 595–604.

CHERNIACK, M., BALAKRISHNAN, H., BALAZINSKA, M., CARNEY, D., CETINTEMEL, U., XING, Y., AND ZDONIK, S. 2003. Scalable Distributed Stream Processing. In *The First Biennial Conference on Innovative Database Systems*.

CHOI, B. 2002. What Are Real DTDs Like. In *The 5th International Workshop on the Web and Databases*. Madison,Wisconsin, 43–48.

CLARK, J. AND DEROSE, S. 1999. XML path language (XPath) version 1.0. W3C Recommendation http://www.w3.org/.

DEUTSCH, A., FERNANDEZ, M., FLORESCU, D., LEVY, A., AND SUCIU, D. 1998. XML-QL: A query language for XML. Available at http://www.w3.org/xml/.

DIAO, Y., FISCHER, P., AND FRANKLIN, M. J. 2002. YFilter: Efficient and Scalable Filtering of XML Documents. In *The 18th International Conference of Data Engineering*. 341–344.

FERNANDEZ, M. AND SIMEON, J. 2002. Galax. http://db.bell-labs.com/galax/.

FERNANDEZ, M. F., FLORESCU, D., KANG, J., LEVY, A. Y., AND SUCIU, D. 1997. STRUDEL: A Web-site management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, J. Peckham, Ed. Tucson, Arizona, 549–552.

GANSNER, E. R. AND NORTH, S. C. 2000. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience 30,* 11 (September), 1203–1233.

GNU 1991. GNU general public license. Free Software Foundation, Inc. http://www.gnu.org/copyleft/gnu.html. Version 2.

GOTTLOB, G., KOCH, C., AND PICHLER, R. 2002. Efficient algorithms for processing XPath queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. Hong Kong, China.

GREEN, T. J., MIKLAU, G., ONIZUKA, M., AND SUCIU, D. 2003. Processing XML streams with Deterministic Automata. In *The 9th International Conference on Database Theory*. Siena, Italy, 173–189.

HOFFMANN, C. M. AND O'DONNELL, M. J. 1982. Pattern matching in trees. *Journal of the ACM (JACM) 29,* 1, 68–95.

HOPCRAFT, J. AND ULLMAN, J. 1979. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley.

IBM. 2001. XML Generator. `http://www.alphaworks.ibm.com/tech/xmlgenerator`.

KATZ, H. 2002. XQEngine. `http://www.fatdog.com`.

KAY, M. H. 2002. SAXON: an XSLT processor. `http://saxon.sourceforge.net/`.

KILPEL, P. 1992. Tree matching problems with applications to structured text databases. Ph.D. thesis, Dept. of Computer Science, University of Helsink.

LAKSHMANAN, L. V. AND SAILAJA, P. 2002. On Efficient Matching of Streaming XML Documents and Queries. In *The 8th International Conference on Extending Database Technology.* Prague, Czech Republic, 142–160.

LEY, M. Computer Science Bibliography. `http://dblp.uni-trier.de/xml/`.

LIU, L., PU, C., AND TANG, W. 1999. Continual Queries for Internet Scale Event-Driven Information Delivery. *Knowledge and Data Engineering 11,* 4, 610–628.

LIU, L., PU, C., AND TANG, W. 2000. Webcq-detecting and delivering information changes on the web. In *9th International Conference on Information and Knowledge Management.* 512–519.

LUDASCHER, B., MUKHOPADHAYN, P., AND PAPAKONSTANTINOU, Y. 2002. A Transducer-Based XML Query Processor. In *The 28th International Conference on Very Large Data Bases.* Hong Kong, China, 227–238.

MADDEN, S. AND FRANKLIN, M. J. 2002. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *The 18th International Conference of Data Engineering.*

MEGGINSON, D. ET AL. 2002. Simple API for XML. `http://www.saxproject.org/`.

MIKLAU, G. AND SUCIU, D. 2002. Containment and Equivalence for an XPath Fragment. In *The 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* Madison,Wisconsin, 65–76.

OLTEANU, D., KIESLING, T., AND FRANOIS BRY. 2002. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams. Tech. Rep. PMS-FB-2002-12, Institute for Computer Science, Ludwig-Maximilians University, Munich. May.

OLTEANU, D., MEUSS, H., FURCHE, T., AND BRY, F. 2002. XPath: Looking forward. In *Workshop on XML-Based Data Management (XMLDM) at the 8th Conference on Extending Database Technology.* Springer-Verlag, Prague, 109–127.

PENG, F. AND CHAWATHE, S. S. 2003. The XSQ project. `http://www.cs.umd.edu/projects/xsq/`.

PLALE, B. AND SCHWAN, K. 2000. dQUOB: Managing Large Data Flows by Dynamic Embedded Queries. In *The 9th IEEE International Symposium on High Performance Distributed Computing.* Pittsburgh, Pennsylvania, 263–270.

PLALE, B. AND SCHWAN, K. 2003. Dynamic querying of streaming data with the dquob system. *IEEE Transactions on Parallel and Distributed Systems 14,* 4 (April), 422–432.

SEGOUFIN, L. AND VIANU, V. 2002. Validating Streaming XML documents. In *The 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* Madison, Wisconsin, 53–64.

TUCKER, P. A., MAIER, D., AND SHEARD, T. 2003. Applying punctuation schemes to queries over continuous data streams. *Bulletin of the Technical Committe on Data Engineering, IEEE Computer Society 26,* 1 (March), 33–40.

W3C XSL WORKING GROUP. 2002. XSL Transformations (XSLT) Version 2.0. W3C Working Draft, W3C, `http://www.w3.org/TR/xslt20/`. April.

WIDOM, J. 1996. The starburst active database rule system. *IEEE Transactions of Knowledge and Data Engineering 8,* 4 (August), 583–595.

WU, C. H., HUANG, H., ARMINSKI, L., ET AL. 2002. The Protein Information Resource: an integrated public resource of functional annotation of proteins. Nucleic Acids Ressearch 30,35-37.

XER 2000. The Xerces Java parser readme. `http://xml.apache.org/`.

XSL WORKING GROUP AND THE XML LINKING WORKING GROUP. 2000. Document Object Model Level 2 Core Specification. W3C Recommendation, W3C, `http://www.w3c.org/DOM/`. November.

Zdonik, S., Stonebraker, M., Cherniack, M., Cetintemel, U., Balazinska, M., and q, H. B. 2003. The aurora and medusa projects. *Bulletin of the Technical Committe on Data Engineering, IEEE Computer Society 26,* 1 (March), 3–10.

A. DTD USED IN IBM XML GENERATOR FOR THE SYNTHETIC DATA

```
<!ELEMENT root (pub*)>

<!ELEMENT pub (year,book*,paper*, proceedings*)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT book (title, author, publisher?, year?,
                price?)>
<!ATTLIST book id ID #REQUIRED>
<!ELEMENT paper (title, author, pages?,
                 proceedings?, year?)>
<!ATTLIST paper id ID #IMPLIED>

<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name, institute, email, pub?)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT pages (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT proceedings (name, place, time, paper*)>
<!ATTLIST proceedings category CDATA #IMPLIED>

<!ELEMENT email (#PCDATA)>
<!ELEMENT institute (#PCDATA)>
<!ELEMENT name (first, last)>
<!ELEMENT place (country, city)>
<!ELEMENT time (#PCDATA)>

<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT city (#PCDATA)>
```

B. TEMPLATE FILE USED IN TOXGENE

The template file used in Toxgene to generate the synthetic dataset for the experiment in Figure 43 is shown in Figure 54.

C. QUERIES AND COMMANDS USED FOR SOME SYSTEMS

C.1 Galax

For GALAX, the query.xq file is like following:

```
<result> {
$shake/root/PLAY/ACT/SCENE/SPEECH/SPEAKER/text()
}</result>;
```

The context file is like following:

```
define global $shake {
treat as document (document("shake.xml"))
}
```

The command is like following:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE tox-template SYSTEM 'http://www.cs.toronto.edu/tox/toxgene/ToXgene1_1.dtd'>
<tox-template>
  <tox-distribution name="c4" type="uniform" minInclusive="1" maxInclusive="6">
  </tox-distribution>
  <tox-list name="steps" readFrom="input.xml">
        <element name="chunk">
          <complexType><element name="step" type="byte"/></complexType>
        </element>
  </tox-list>
  <tox-document name="result">
        <element name="root">
          <complexType>
                <tox-foreach path="[steps/chunk]" name="s">
                  <element name="chunk">
                        <complexType>
                          <element name="a" maxOccurs="unbounded" tox-recursionLevels="c4">
                                <complexType mixed="true">
                                  <tox-scan path="[$s/step]">
                                        <attribute name="count">
                                          <simpleType>
                                                <restriction base="byte">
                                                  <minInclusive value="01"/>
                                                  <tox-number sequential="yes"/>
                                                </restriction>
                                          </simpleType>
                                        </attribute>
                                        <tox-alternatives>
                                          <tox-option odds="10">
                                                <element name="prior">
                                                  <tox-expr value="2"/>
                                                </element>
                                                <element name="red" maxOccurs="10000">
                                                  <tox-expr value="[!]"/>
                                                </element>
                                                <element name="posterior">
                                                  <tox-expr value="2"/>
                                                </element>
                                          </tox-option>
                                          <tox-option odds="30">
                                                <element name="prior">
                                                  <tox-expr value="2"/>
                                                </element>
                                                <element name="green" maxOccurs="10000">
                                                  <tox-expr value="[!]"/>
                                                </element>
                                                <element name="posterior">
                                                  <tox-expr value="2"/>
                                                </element>
                                          </tox-option>
                                          <tox-option odds="60">
                                                <element name="prior">
                                                  <tox-expr value="2"/>
                                                </element>
                                                <element name="blue" maxOccurs="10000">
                                                  <tox-expr value="[!]"/>
                                                </element>
                                                <element name="posterior">
                                                  <tox-expr value="2"/>
                                                </element>
                                          </tox-option>
                                        </tox-alternatives>
                                  </tox-scan>
                                </complexType>
                          </element>
                        </complexType>
                  </element>
                </tox-foreach>
          </complexType>
        </element>
  </tox-document>
</tox-template>
```


Fig. 55: Template file used in Toxgene

```
time -f "%U" xmlquery -pic -verbose -context galax
_context.xq galax.nc.xq > galax.nc.out
```

C.2 Saxon

For SAXON, the style-sheet file is like following:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL
                           /Transform" version="1.1">
<xsl:template match="/">
 <result>
  <xsl:for-each select="/root/PLAY/ACT/SCENE/SPEECH
                        /SPEAKER">
    <xsl:value-of select="."/>
  </xsl:for-each>
 </result>
</xsl:template>
</xsl:stylesheet>
```

The command is like following:

```
java com.icl.saxon.StyleSheet -x org.apache.xerces.parsers.  SAXParser
-t shake.xml saxon.nc.xsl > saxon.nc.out
```

C.3 Joost

For Joost, the transformation file is like following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<stx:transform xmlns:stx="http://stx.sourceforge.net
                          /2002/ns" version="1.0">
  <stx:template match="/root/PLAY/ACT/SCENE/SPEECH
                       /SPEAKER/text()">
        <stx:copy />
  </stx:template>
</stx:transform>
```

The command is like following:

```
time -o -f "%U" java -Dorg.xml.sax.driver=org.apache.xerces.  parsers.SAXParser
net.sf.joost.Main shake.xml joost.nc.stx > joost.nc.out
```

C.4 XMLTK

Note that we have modified the xrun program so that it reports the running time.
The command is like following:

```
xrun "/root/PLAY/ACT/SCENE/SPEECH/SPEAKER/text()" shake.xml > xrun.nc.out
```

C.5 XQEngine

For XQEngine, the command to query the SHAKE dataset is like following:

```
java XQE a_and_c.xml   cymbelin.xml  hen_vi_1.xml
```

```
j_caesar.xml  merchant.xml  pericles.xml t_night.xml
all_well.xml  dream.xml     hen_vi_2.xml  john.xml
m_for_m.xml taming.xml   troilus.xml as_you.xml
hamlet.xml    hen_vi_3.xml  lear.xml      much_ado.xml
r_and_j.xml   tempest.xml  two_gent.xml com_err.xml
hen_iv_1.xml  hen_viii.xml  lll.xml       m_wives.xml
rich_iii.xml  timon.xml    win_tale.xml coriolan.xml
hen_iv_2.xml  hen_v.xml     macbeth.xml   othello.xml
rich_ii.xml   titus.xml "/PLAY/ACT/SCENE/SPEECH/SPEAKER"
>> XQE.nc.out
```