XSQ: A Streaming XPath Engine

FENG PENG and SUDARSHAN S. CHAWATHE
University of Maryland

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 augmented finite state automata. A design goal of XSQ is buffering data for the least amount of time possible. We present a detailed experimental study that characterizes the performance of XSQ and related systems, and that 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*. 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. 2000]. We focus in this article on XML data that are in streaming format. *Streaming data* are available for reading only once and 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 streaming data include real-time news feeds, stock market data, sensor data, surveillance feeds, and data from network monitoring equipment. Some data are available in only streaming form because they have a limited lifetime of interest to most consumers. For example, articles in a topical news feed are not likely to retain their value for very long. Moreover, the data source may lack resources to provide nonstreaming access. For example, a network router that provides real-time packet counts, error reports, and security alerts is typically unable to fulfill the processing or storage requirements of providing non-streaming access to such data.

This material is based upon work supported by the National Science Foundation (NSF) under grants IIS-9984296 (CAREER) and IIS-0081860 (ITR).
Authors' address: Department of Computer Science, University of Maryland, College Park, College Park, MD; Correspondence, email: chaw@cs.umd.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2005 ACM 0362-5915/05/0600-0577 $5.00

```
1.  <!-- begin document -->
2.  <pub>
3.    <book id="1">
4.      <price> 12.00 </price>
5.      <name> First </name>
6.      <author>A </author>
7.      <price type="discount"> 10.00 </price>
8.    </book>
9.    <book id="2">
10.     <price> 14.00 </price>
11.     <name> Second </name>
12.     <author> A </author>
13.     <author> B </author>
14.     <price type="discount"> 12.00 </price>
15.   </book>
16.   <year> 2002 </year>
17. </pub>
18. <!-- end document -->
```

Fig. 1. Input Fragment 1.

There have been a number of recent proposals for query languages for XML and XML-like data models [Abiteboul et al. 1996; Fernández et al. 1997; Buneman et al. 1996; Deutsch et al. 1998; Clark and DeRose 1999; Boag et al. 2003]. Of these proposals, XPath and XQuery have emerged as the standards recommendations that are likely to receive broad support. In this article, we focus on XPath. However, since XPath forms an important core of XQuery, the methods we describe are also useful for XQuery engines. We now present two examples that illustrate some of the challenges of XPath evaluation in a streaming environment. (A brief description of XPath appears in Section 2.)

Example 1. Consider the following query on the input fragment depicted in Figure 1: `/pub[year > 2000]/book[price < 11]/author`. Intuitively, it returns the `authors` of the `books` that have been published after `year` 2000 and that have a `price` less than 11.

When we encounter the first `author` element on line 6 in the stream, it is easy to deduce that the sequence of its ancestor elements matches the pattern `/pub/book/author` (since the `pub` and `book` elements have been encountered earlier and are still open). The predicate `[year > 2000]` is not satisfied by the `pub` element (line 2) because we have not encountered any `year` child elements. However, qualifying child elements may occur later in the stream. Therefore, we cannot yet conclude that the predicate is false. For the `book` element on line 3, we have encountered the first `price` element (line 4), which does not satisfy the predicate `[price < 11]`. Again, we cannot yet conclude that the predicate is false for this `book` element because it may have additional `price` child elements later in the stream. Thus, at line 6 in the stream, we cannot determine whether the `author` element belongs to the result. The element must therefore be buffered.

When we encounter the `price` element on line 7, we can check that it satisfies the predicate for its parent `book` element. However, we still cannot determine whether the `pub` element on line 2 satisfies the predicate `[year > 2000]`. Consequently, it is still unknown whether the `author` element on line 6 belongs

to the result. Therefore, we must continue to buffer the author element and record the fact that the second predicate has been satisfied but not the first one. Similarly, the two author elements on lines 12 and 13, which belong to the second book element, have to be buffered as well. At this point in the stream (line 13), there are three author elements in the buffer: two with value A and one with value B.

When we encounter the price element on line 14, we note that it does not satisfy [price < 11]. Since its parent book element is still open, we cannot yet conclude that the book element fails to satisfy the predicate. That conclusion can only be made when we encounter </book> on line 15. At this point in the stream (line 15), the two author child elements of this book element should be removed from the buffer. The other author element (with value A) remains in the buffer because its first predicate may be satisfied by data encountered later in the stream.

When we encounter the year element on line 16, we may determine that the pub element on line 2 satisfies the predicate [year > 2000]. Recalling that this pub element is the ancestor of the author element remaining in the buffer, which has already satisfied the other predicate, we determine that this author 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 items that are potentially in the result before we encounter the items required to evaluate their predicates. We need to buffer such potential result items. Second, buffered items have to be distinguished so that, after the evaluation of a predicate, only the items that are affected by that predicate are processed. Third, in order to buffer items for the least amount of time possible, we need to check whether pending buffer items can be output as soon as some predicate is satisfied. Finally, predicates access different portions of the data. Some should be evaluated when the start-tag is encountered, while others may only be evaluated upon encountering the text content. (There are other forms of predicates, discussed later.)

Example 2. Consider the query //pub[year>2000]//book[author]//name for the input fragment depicted in Figure 2. This example introduces some problems not seen in Example 1. Since the closure axis // is used in this query, an element and its descendants may match the same location step. For instance, the pub elements in lines 1 and 9 match the node test in the first location step. There are three ways in which the name in line 11 matches the pattern of the query (ignoring predicates). Each matching yields a different result for the predicates, as summarized in the following table.

pub	book	[year > 2000]	[author]	name
line 2	line 7	true	false	line 11
line 2	line 10	true	true	line 11
line 9	line 10	false	true	line 11

As indicated by the table, only the matching of the second row satisfies both predicates. However, the predicate results of these different matchings may arrive in different orders and need further consideration.

```
1.  <!-- begin document -->
2.  <pub>
3.    <book>
4.      <name> X </name>
5.      <author> A </author>
6.    </book>
7.    <book>
8.      <name> Y </name>
9.      <pub>
10.       <book>
11.         <name> Z </name>
12.         <author> B </author>
13.       </book>
14.       <year> 1999 </year>
15.     </pub>
16.   </book>
17.   <year> 2002 </year>
18. </pub>
19. <!-- end document -->
```

Fig. 2. Input Fragment 2.

When we encounter `</pub>` on line 15, we know that this `pub` element (of line 9) fails the predicate `[year > 2000]`. However, we cannot remove the `name` element on line 11 from the buffer because it is still possible that this item satisfies the query due to a subsequent `year` child of the other `pub` element on line 2. A similar situation occurs when we encounter `</book>` on line 16. Only when all the possible matchings have failed to satisfy the predicates can we remove the item from the buffer.

When multiple matchings evaluate all predicates to true, we must remove duplicate results. For example, if there were an additional `author` element between lines 8 and 9, the matching indicated by the first row of the above table would also satisfy both predicates. The `name` element, however, should be outputted only once.

The XSQ system uses an automaton-based method to evaluate XPath queries over XML streams. The automaton, called an HPDT (Section 3.1), is a finite state automaton augmented with a buffer. For every input XPath query, we construct an HPDT hierarchically using a template-based method. Using the HPDT as a guide, a runtime engine (Section 4) responds to the incoming stream and emits the query result. The multiple matching problem (Example 2) is solved by associating with every buffer item its matching with the query and a flag the indicates the current predicate results. We note that the HPDT is used simply as convenient conceptual machinery to describe our methods. The expressiveness and theoretical complexity of the automata are not our focus in this article.[1]

Organization. Some preliminaries, including a brief description of XPath, are covered in Section 2. Section 3 introduces how we compile an XPath query into an HPDT. In Section 4, we describe how the runtime engine processes the

[1]A brief description of our methods and the results of a preliminary experimental study of XSQ appear in Peng and Chawathe [2003].

$$
\begin{aligned}
\mathbf{Q} &::= \mathbf{N}^{+}[/\mathbf{O}] \\
\mathbf{N} &::= \{/|//\}\mathbf{nodetest}\ [\mathbf{P}] \\
\mathbf{P} &::= [\ \mathbf{F}[\mathbf{OP\ constant}]] \\
\mathbf{F} &::= @\mathbf{attribute} \mid \mathbf{nodetest}[@\mathbf{attribute}] \mid \text{text}() \\
\mathbf{O} &::= @\mathbf{attribute} \mid \text{text}() | \text{count}() | \text{sum}() \\
\mathbf{OP} &::= \ > \ | \geq | = | \ < \ | \geq |! =
\end{aligned}
$$

Fig. 3. EBNF for an XPath subset.

incoming stream using the HPDT as a guide. We also discuss the correctness and complexity of the method, along with a few key implementation details. Related work is discussed in Section 5. Section 6 presents our experimental study of XSQ and related systems. We conclude in Section 7.

2. PRELIMINARIES

A static XML document is usually modeled as a tree (e.g., a DOM tree [Hors et al. 2000]). We model the input XML stream as a sequence of **events**, modeled after SAX [SAX Project Organization 2001] events. Each event e is a quadruple of the form (n, al, t, d): (1) The string n is the name of the element that generates the SAX event. (2) The list al contains pairs of the form (a, v), indicating that the element has attribute a with value v. Since elements are not permitted to have multiple attributes with the same name, the attribute name a uniquely identifies a pair in the list. 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. (3) The type t 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 $(\textit{text}(), v)$, indicating that v is the text content of the element. (4) Finally, the integer d is the depth of the element in the document tree. The root of the document tree, also called *document root*, has depth 0. The attr and text nodes have the same depth as their parent nodes.[2] A SAX parser generates a *start-document* event (S-DOC) when it begins parsing an XML document and an *end-document* event (E-DOC) when it finishes parsing the document. It is convenient to regard the S-DOC and E-DOC events as the begin and end events, respectively, of the document root.

A simplified grammar for **XPath** is depicted in Figure 3. An XPath query is an expression of the form of $N_1 N_2 \ldots N_k[/O]$, which consists of a **location path**, $N_1 N_2 \ldots N_k$, and an optional **output function** O. Each **location step** N_i is of the form $/a_i::n_i\,[p_i]$ where a_i is an **axis**, n_i is a **node test** that specifies the name of elements N_i can match, and p_i is an optional **predicate** that is specified syntactically using square brackets.

An XPath query is interpreted as follows: Each location step selects a set of nodes in the document tree. For every node x selected by N_{i-1}, N_i selects a set of nodes using x as the *context node*. The set of nodes selected by the last location step consists of the *result set* of the query. In more detail, there is an

[2]Strictly speaking, SAX events do not include this depth component. Instead, this information is added by XSQ by wrapping SAX events and maintaining a depth counter internally.

implicit zeroth location step N_0 that always selects the document root. Thus, N_1 is always evaluated using the document root as the context node. The axis in a location step N_i specifies the relation between a node y selected by N_i and the context node x in which N_i is evaluated. In the simplified grammar, `/` is shorthand for the `/child::` axis, which specifies that y must be x's child. Similarly, `//` is shorthand for the `/descendant-or-self::node()/` axis, which specifies that y must be a descendant of x (not necessarily a proper descendant). After N_k is evaluated, the output function O is applied to every node in the result set to produce the final output. The output function may specify an attribute or the text value of an element. It may also use an aggregation function such as `sum()` or `count()`. If no output expression is specified in the query, the elements in the result set are returned as the query result.

In a streaming environment where no DOM tree [Hors et al. 2000] is built, the above interpretation is not convenient. Instead, we use the following equivalent interpretation based on the SAX model [SAX Project Organization 2001]. A **matching** between an element e_m and a location path $N_1 N_2 \cdots N_m$ is a sequence of elements $(e_0, e_1, \ldots, e_m)$ such that (1) e_0 is always the document root that matches the implicit location step N_0 described above, (2) for all $i \in [1, m]$, e_i's name matches n_i, the node test of N_i, and (3) for all $i \in [1, m-1]$, e_i is the parent of e_{i+1} if a_{i+1}, the axis of N_{i+1}, is `/` and the ancestor if a_{i+1} is `//`. In this case, we can also say e_m *matches location step* N_m. If m is k, the size of the query, we say e_k *matches the query*. If p_i tests the content or existence of a child with name `c`, an element e *matches predicate* p_i if and only if e's name is `c` and e's parent has a matching with N_i. An element e is in the result set of a query $N_1 N_2 \cdots N_k$ if and only if there exists a matching between e and the query that satisfies all the predicates. If there are multiple such matchings, as illustrated in Example 2, the output function is applied to e only once. This definition of the result set is equivalent to the traditional step-by-step evaluation scheme, as can be verified easily by induction on the number of location steps.

If a predicate contains no value comparison, it tests the existence of specified object. For example, `book[price]` tests whether a `book` element has a `price` child. Predicates with value comparisons are evaluated as follows. First, when an element's attribute value or the text content a is compared with a literal v, XPath semantics specify that, if v is a number, a must be coerced to a numeric type. The comparison then proceeds with the usual numeric semantics. If the coercion fails, the predicate returns false. Second, a predicate such as `[price=10]` is interpreted as `[price/string()=10]`, where `price/string()` returns the aggregation of the text content within the `price` element. For ease of presentation, we assume in this article that `string()` function be replaced by the `text()` function and that there is at most one text event for any element. (The `text()` function returns the set of text children of a node. For example, `string()` on a `price` element `<price>10<note>sale</note><price>` returns `10 sale`, while `text()` returns `10`.) However, our method easily supports `string()` and multiple text events within an event by buffering all the text events and delaying predicate-processing for an event to its end, after all text events have been encountered.

The XPath subset we study in this article does not include nonforward axes, such as *sibling* and *ancestor*, Boolean connectives in predicates, or position functions such as *pos()* and *last()*. These features pose additional challenges for streaming evaluation and are the subject of continuing work. For example, the value of the *last()* function cannot be determined until the entire result set is available.

3. COMPILING XPATH QUERIES

In XSQ, an XPath query is first compiled into an *HPDT*, which is used by the runtime engine (Section 4) to evaluate the query on a streaming XML input. The HPDT is built in a layered manner with overlapping groups of states called *BPDT*s. We begin by describing HPDTs in Section 3.1. In Section 3.2, we describe the BPDT templates that form the basis of our method for building HPDTs. This method is described in detail in Section 3.3. Finally, Section 3.4 describes how aggregation functions are implemented in XSQ.

3.1 HPDT

The HPDT is a nondeterministic finite-state automaton augmented with a buffer. Its transitions are optionally associated with predicates and buffer operations. A transition is taken only if its predicate, if any, is satisfied. The buffer operation, if any, on a transition is executed when that transition is taken.

Transitions. The input to an HPDT is a sequence of SAX events, each of which takes the form (n, al, t, d), where n is the name, al is the attribute list, t is the type, and d is the depth. (See Section 2.) On transition arcs, we specify events as (n, t), where n specifies an element name and t specifies a SAX event type. Besides the three types (B, E, and T) introduced in Section 2, t can also be $\bar{*}$, a **catchall** type that matches all three types of events. A transition x with symbol (n, t) **matches** an input event e if n matches $e.n$ and t matches $e.t$. The attribute list al and depth d of an event are used in predicate evaluation and output composition, as described later. When a transition x emerging from a state matches the current event e in the input stream, we say this state **accepts** e. However, if x has a predicate then x is taken only if e satisfies the predicate, as described in Section 2. In the figures that depict state transition diagrams, we use an XML-like notation: `<n>` for (n, B), `</n>` for (n, E), and `<n.text()>` for (n, T). In our description, we use **Element(e)** to denote the XML element that generates event e. We use (e_i, B) to denote the begin event of element e_i, (e_i, E) to denote its end event, and (e_i, T) to denote its text event.

After an HPDT takes a transition x, the set of active states is determined not only by x's target state, but also by the type of x. There are four types of transitions: (1) **self-closure transitions**, identified in state transition diagrams using the symbol `//` next to arrows; (2) **closure transitions**, identified using $=$ or `||` on arrows; (3) **catchall transitions**, identified using $\bar{*}$; and (4) **regular transitions**, identified by the absence of special markings. These transitions differ in their effects on the runtime engine, as described in Section 4.

BPDT. The states in an HPDT are organized in overlapping groups, each of which is called a BPDT. In each BPDT, we specify a START state, a TRUE state,

an optional NA state, and an optional FALSE state. Intuitively, a BPDT contains a group of states that evaluate a location step of the XPath query. The START state is the entry point into the BPDT. The TRUE (FALSE) state indicates the predicate of this location step has evaluated to true (respectively, false). The NA (not available) state indicates that the data required to determine the truth value of a predicate has not yet been encountered in the stream. The BPDTs are connected by overlapping the START state of one BPDT with the TRUE or NA state of another. The TRUE, NA, and FALSE states are called P-VALUE states (because they indicate the result of predicate evaluations). The other states, excluding START, are called P-EVAL states (because they are used to evaluate a predicate).

Buffer. The buffer of an HPDT is used to hold potential result items. We associate with each buffer item a $(k + 1)$-bit flag, where k is the query length (number of location steps). The ith bit of the flag (counting from the left, starting with 0) denotes the current state of the predicate (perhaps trivial) of the ith location step: 1 for true and 0 for pending. Recall the zeroth location step always matches the document root and has no predicate. Thus, the zeroth bit of the flag is always 1 (We use $(k+1)$-bit flags instead of k-bit flags for better correspondence with BPDT identifiers, described in Section 3.3.) We use f_i to denote the ith bit of a flag f. If all bits in a flag are 1, we say the flag is a *true flag*.

An HPDT uses buffer operations `set`, `remove`, and `add`. We describe these operations only informally here, deferring the details to our discussion of the runtime engine in Section 4. In that section, we also describe how the runtime engine applies `set` and `remove` operations selectively to only a subset of buffer items. However, for ease of presentation in the rest of this section, we assume that these operations apply to all items in the buffer. The *set(i)* operation sets (to 1) f_i for every buffer item. The *remove(i)* operation removes all buffer items having $f_i = 0$. The *add(f, a)* operation creates a buffer item with flag f using the *feature* a of the event e. The feature a may be an attribute name (including "`text()`"), in which case $e.a$ is added. It may also be the catchall symbol $\bar{*}$, in which case the serialized (string) representation of e is appended, including all its attributes. For example, for the begin event (`book`, {(`id`,`"1"`)}, B, 1), the operation `add`($\bar{*}$, f) creates a buffer item that contains the string `<book id="1">` and has flag f. We do not use an explicit output operation. Rather, when the flag of a buffer item becomes a true flag (all 1s), the item is ready for output. The document order among the output items is preserved (as required by XPath) by using a global queue, as described in Section 4.4.

The following example illustrates how an HPDT can be used to evaluate an XPath query. A special BPDT, called the **root BPDT**, is used in the HPDT to process the start-document (S-DOC) and end-document (E-DOC) events, which are generated for the document root.

Example 3. We can use the HPDT H depicted in Figure 4 to evaluate the query: `/pub/book[author]/price/text()`. We use rounded boxes to enclose the BPDTs, which are numbered using the scheme described in Section 3.3. All the transitions in H are regular transitions. Note that the START states of BPDTs $b(2, 3)$, $b(3, 6)$, and $b(3, 7)$ are TRUE or NA states of other BPDTs (TRUE state of $b(1, 1)$, NA state of $b(2, 3)$, and TRUE state of $b(2, 3)$, respectively). Such a shared

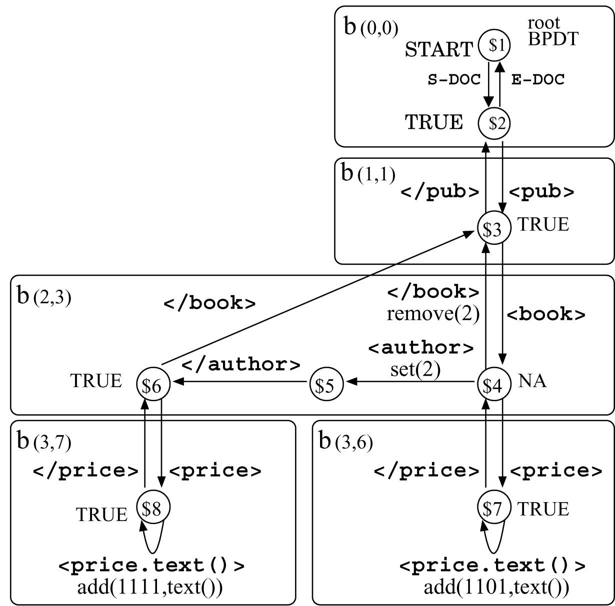


Fig. 4. An HPDT for query `/pub/book[author]/price/text()`.

state belongs to both the BPDT suggested by its enclosing box and the BPDT below it. Let us consider the first a few actions of H on the input fragment of Figure 1. After processing the begin event of the `price` element on line 4, state $7 is active. The transition on the text event adds the text content, 12.00, of this `price` element to the buffer, with flag 1101. When H encounters the begin event of the `author` element on line 6, it sets f_2 to 1 for the buffer items and transits to state $5 (and state $6 at the end event of this `author` element). Since the buffer item with value 12.00 now has its flag set to all 1's, it is emitted as output. When H encounters the next `price` element (line 7 of Figure 1), it transits to state $8. The transition from $8 on the text event results in the addition of 10.00 to the buffer, with flag 1111, which in turn causes 10.00 to be sent immediately to the output. (Since this `price` element's `book` parent has already satisfied the predicate `[author]`, it should be immediately output.)

3.2 Templates for BPDT

We generalize the BPDT $b(2,3)$ of Example 3 to the template depicted in Figure 5. We instantiate BPDTs from this template to evaluate location steps of the form of `/n[c]`. In general, we classify location steps into five categories for the purpose of template-based generation of BPDTs. In the following descriptions, we consider only the `/` axis. The modifications needed for the `//` axis are made separately after the templates are instantiated. During the instantiation of a template for a location step N_i, the parameter l used by the buffer operations in the template is replaced by i. The instantiation procedure is described further in Section 3.3. The design of these templates is guided

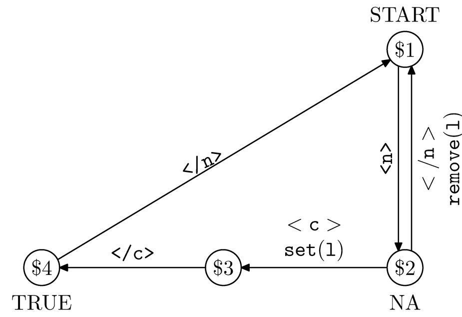


Fig. 5. Template BPDT for: /n[c].

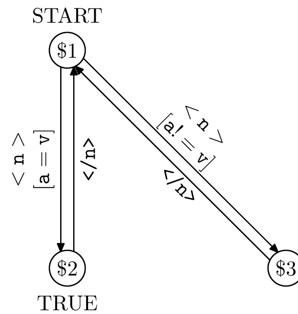


Fig. 6. Template BPDT for: /n[c@a = v].

by the existential semantics of XPath predicates. Once a predicate's result has been determined as true or false, the automata transit to states in which further data that could be used to evaluate the predicate is skipped. Buffered items are always processed, using the *set* or *remove* operations, at the earliest time that a predicate's result can be determined.

Template 1. Location steps of the form `/n`, `/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 3), and `v` is a literal: Figure 6 illustrates the template for `/n[@a = v]`. For `/n[@a]`, the test of the attribute value is replaced by a test for the existence of the attribute. For `/n`, state $3 and the transitions connected to it are not used and there is no test for the attribute. This template does not include an NA state because the result of the predicate is always known for each element as it is encountered. If the result is false, the BPDT enters state $3 that accepts nothing but the end event of the same element. Otherwise, the BPDT enters the TRUE state $2, which indicates that the predicate has been satisfied.

Template 2. Location steps of form `/n[text() op v]`, which include a predicate on the text content of matching elements: Figure 7 illustrates the template for `/n[text() = v]`. Since we assume that there is only one text event in each element, we compare the text event with the literal `v` only once. If the element `n` has no text contents (which can be determined only at the end of the element),

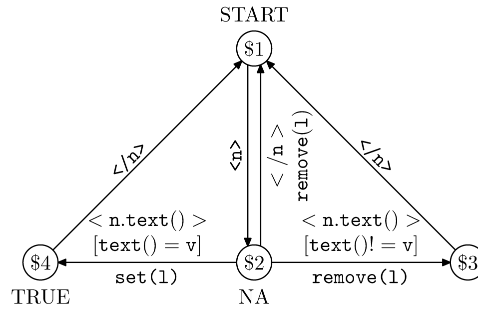


Fig. 7. Template BPDT for: /n[text() = v].

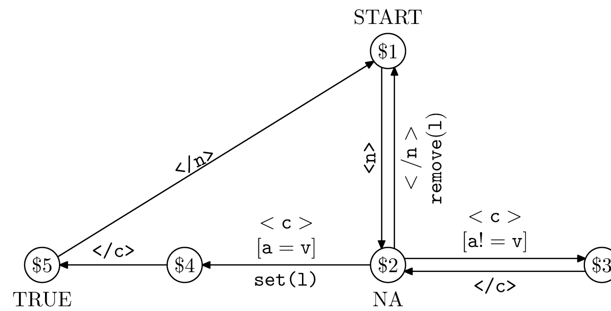


Fig. 8. Template BPDT for: /n[c = v].

the BPDT returns to the START state removing the buffer items that are waiting for this predicate. If the element n contains some text content, the BPDT transits from state \$2 to \$4 if the content satisfies the condition, otherwise it transits from state \$2 to \$3. Once state \$3 is active, it remains active until the end of this n element. State \$2 is the NA state since the predicate is pending when it is active.

Template 3. Location steps of form /n[c], which test the existence of c-children: Figure 5 illustrates the template for /n[c]. The template encodes the existential semantics of XPath predicates: After one c-child element of n satisfies the predicate, state \$4 becomes active and no other c-child is tested. Only when the end of n is encountered and no c-child is encountered do we conclude that the predicate is false.

Template 4. Location steps of the form /n[c@a] and /n[c@a op v], which include predicates that reference attributes of children. Figure 8 illustrates the template for /n[c@a = v]. For /n[c@a], the test of the attribute value is replaced by a test for the existence of that attribute: This template encodes the existential semantics of predicates in a manner similar to that of the template for /n[c]. However, here a c-child may not satisfy the predicate, in which case state \$3 becomes active and this c-child is ignored.

Template 5. Location steps of the form /n[c op v], which include predicates that test the values of the child elements. Figure 9 illustrates the template for /n[c = v]. Recall, from Section 2, that the predicate [c op v] is interpreted

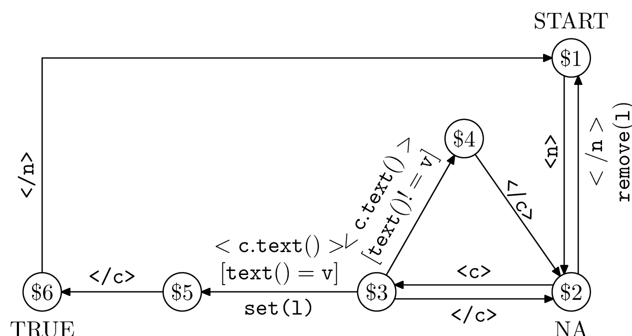


Fig. 9. Template BPDT for: `/n [c = v]`.

as `[c/text() op v]`: This template is similar to that in Figure 5, but includes transitions to process the text events of `c`-children.

3.3 Building HPDTs from XPath Queries

Consider a query $Q = N_1 N_2 \cdots N_k$, where $N_i = /a_i :: n_i[p_i]$. The HPDT H for Q is is generated in a layered manner. Every BPDT is assigned a two-dimensional **identifier** (l, m) and is denoted as $b(l, m)$, where l is the layer and m is its position in the lth layer. We use the notation $b(x, y)$.START to denote the START state of BPDT $b(x, y)$ (and similarly for the TRUE and NA states). We first create the **root BPDT** $b(0, 0)$ (Figure 10 as the only BPDT in the zeroth layer. This BPDT does not depend on the XPath query and corresponds to the implicit zeroth location step of a query, which matches the document root. Its START state, denoted as s_0, is also the START state of the HPDT. Layer l, for $l \in [1, k]$ is generated as follows: For every BPDT $b(l-1, m)$ in the $(l-1)$th layer, we create a **child BPDT** $b(l, 2m+1)$, by instantiating the BPDT template that matches N_l. The TRUE state of $b(l-1, m)$ is merged with the START state of $b(l, 2m+1)$. If $b(l-1, m)$ has an NA state, we create another child BPDT, $b(l, 2m)$, by instantiating the template for N_l (again). The start STATE of $b(l, 2m)$ is merged with the NA state of $b(l-1, m)$. When instantiating a template, we set the parameter of the `set` and `remove` operations to the layer number, l.

We summarize in Listing 1 the procedure for creating an HPDT. The **AddBPDT(b,N,s)** procedure instantiates a BPDT using the template that matches location step N and sets the s state (either START or NA) of b as the START state of the new BPDT. After BPDT $b(l, m)$ is created, the **PostProcess** procedure as depicted in Listing 2, is applied to it to perform the following three modifications. First, if a_l (the axis of location step N_l) is `//`, the procedure adds a *self-closure transition* from $b(l, m)$.START to itself, labeled `//`. We then use the **LocateTrans** function to locate all the transitions that emerge from the START state and match the begin event with name n_l (the node test of N_l). We mark them as *closure transitions*. (As discussed in Section 4, these newly marked transitions cause the HPDT to remain in $b(l, m)$.START in order to accept any descendants that also match N_l.)

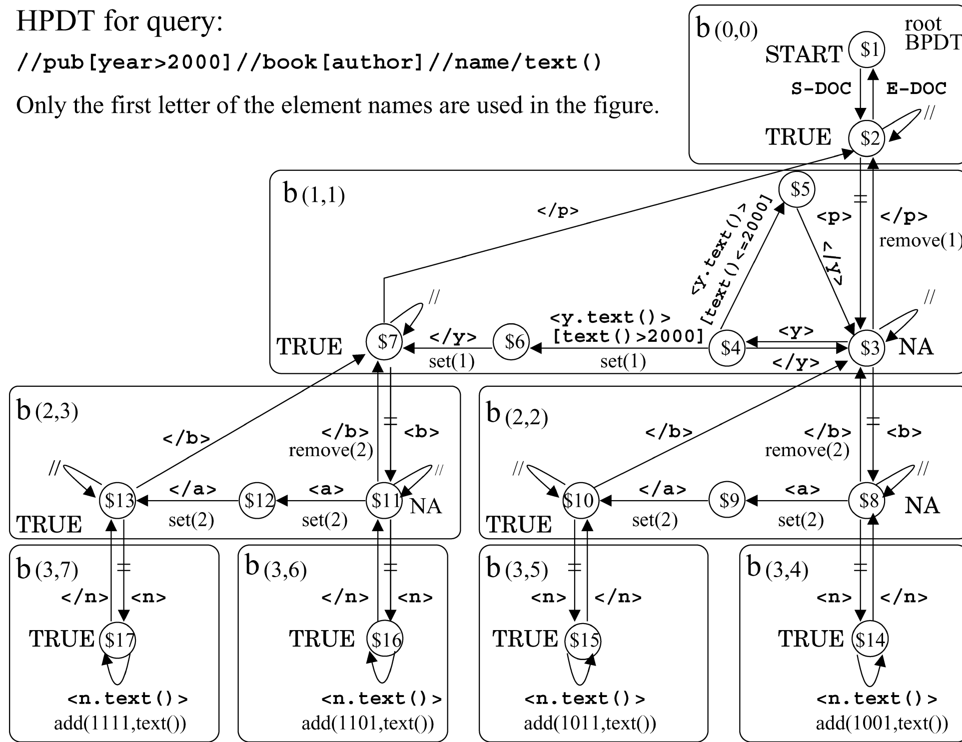


Fig. 10. An HPDT example.

Listing 1: GenerateHPDT(Q)

/* *Build an HPDT from a query* $Q = N_1 N_2 \cdots N_k/O$, *where* $N_i = /a_i :: n_i[p_i]$. */

1 $b(0,0)$ = CreateRootBPDT()
2 **For** $l \leftarrow 1$ ***to*** k **do**
3 **For** $m \leftarrow 0$ ***to*** $2^{l-1} - 1$ **do**
4 $b \leftarrow b(l-1, m)$;
5 **If** $b \neq$ NULL **then**
6 $b(l, 2m+1) \leftarrow$ AddBPDT(b, N_l, TRUE);
7 PostProcess($b(l, 2m+1), Q$);
8 **If** b. NA $\neq$ NULL **then**
9 $b(l, 2m) \leftarrow$ AddBPDT(b, N_l, NA);
10 PostProcess($b(l, 2m), Q$);
end
end
end
end

Second, when a_{l+1} is `//` and p_l (the predicate of N_l) tests a child element, an extra `set` operation is added in $b(l, m)$ by the **AddExtraSet** procedure. This extra `set` operation is used to process descendants that are nested inside the child elements tested by p_l. This modification is needed only for BPDTs

Listing 2: PostProcess(BPDT b, Query Q)

```
   /*Modify b according to Q = N1N2...Nk/O, where Ni = /ai :: ni[pi] .*/
1  l ← b.layer;
2  If al = // then
3     NewTrans(b. START, b. START, "B", //, NULL, SELF-CLOSURE);
4     X ← LocateTrans(b. START, "B", nl);
5     ForEach x ∈ X do x.type ← CLOSURE
   end
   /* Add an extra set operation if needed. */
6  If al+1 = // then AddExtraSet(b);
   /* Add output to the lowest layer BPDTs. */;
7  If l = k then AddOutput(b, Nk, O);
```

Listing 3: AddOutput(BPDT b, Location Step N_k, Output Function O)

```
   /*Translate O to operations in BPDT b, which is created from Nk = /ak :: nk[pk].*/
1  m ← b.position;
2  switchO.feature do
3     case ATTRIBUTE:
4        X ← LocateTrans(b. START, 'B', nk);
5        if pk = NULL then
6              foreach x ∈ X do AddOp(x, add(2m, @attrname));
7              else foreach x ∈ X do AddOp(x, add(2m + 1, @attrname));
8     case TEXT:
         /* Add self-transitions to NA and TRUE states for TEXT events of nk. */
9        if pk = NULL then
10             NewTrans(b. NA, b. NA, 'T', nk, add(2m, text()), REGULAR);
11       NewTrans(b. TRUE, b. TRUE, 'T', nk, add(2m + 1, text()), REGULAR);
12    case CATCHALL:
13       X ← LocateTrans(b. START, 'B', nk);
14       if pk = NULL then
15             foreach x ∈ X do AddOp(x, add(2m, *̄));
16       else foreach x ∈ X do AddOp(x, add(2m + 1, *̄));
17       if b. NA ≠ NULL then
18          NewTrans(b. NA, b. NA, '*̄', '*̄', add(2m, *̄), CATCHALL);
         end
19       NewTrans(b. TRUE, b. TRUE, '*̄', '*̄', add(2m + 1, *̄), CATCHALL);
20       X ← LocateTrans(b. TRUE, 'E', nk);
21       foreach x ∈ X do AddOp(t, add(2m + 1, *̄));
         /* add extra flush operations in the BPDT if needed*/
22       AddExtraSet(b);
   end
```

generated using the templates in the following Figures (with the affected transitions in parentheses): Figure 8 ($\$4 \rightarrow \$5$), Figure 5 ($\$3 \rightarrow \$4$), and Figure 9 ($\$5 \rightarrow \$6$).

Third, for every BPDT $b(k, m)$ in the last (k'th) layer, the **AddOutput**(b, m) procedure translates the output function O into operations in BPDT $b(k, m)$. This procedure is summarized in Listing 3, in which the **NewTrans**(s_1, s_2, e, n, o, t) function is used to create a new transition of type t, from state s_1 to state s_2, on event e of element n, with buffer operation o.

If the query's output function O specifies outputting an attribute of the element that matches N_l, an add operation is 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 with an add operation is added to the TRUE state (and NA state if there is any) in b. If O specifies outputting the whole element, we add a catchall transition labeled with $\bar{*}$ from the TRUE state (and NA state if there is any) to itself together with the add operation. These two transitions match the descendant elements and text contents of the current element. The operation add is also added to both the transition that emerges from the START state that processes the begin event of the element and the transition from the TRUE state to the START state that processes the end event of the element.

The **initial flag** of the add operation in $b(k, m)$ is determined as follows. If p_k, the predicate of the nth step N_k, is empty or p_k tests an attribute, the initial flag is always $2m + 1$. If p_k tests a child element or the text content, then the initial flag is $2m + 1$ if the operation is on a transition whose source or target state is the TRUE state, and $2m$ otherwise. We will see in Section 4.3 that such an initial flag correctly encodes the current state of every predicate for the matching between the current element and the query.

3.4 Aggregations

In order to support aggregates in XPath queries, XSQ uses a statistics buffer called **stat**. This buffer is organized as a map and contains one entry for each aggregation function. The entry's key is the name of the aggregation function and its initial value is *null*. There are two operations on this buffer: The first, `update(aggr)`, updates the entry for aggregation function *aggr* in *stat*. For example, `update(COUNT)` counts the number of buffer items with true flags and adds that number to *stat* entry for *count*; `update(SUM)` adds the numerical value of every buffer item with a true flag to the entry for *sum*. The second operation, `print(aggr)`, outputs the value of the *stat* entry for *aggr*.

For example, consider the following query, which differs from the query of Example 2 only in its use of output function `count()`:

```
//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 10. We replace all occurrences of `set(`l`)` with `update(COUNT)`. The `add(`f, a`)` operation performs the update operation automatically if f is a true flag. We also place a `print(COUNT)` operation on the transition from \$2 to \$1 in the root BPDT.

We may also modify the semantics of the `update()` operation so that it emits a new value whenever the number in the buffer is updated. This change makes preliminary results of aggregation queries available in an online manner. This feature is especially useful when we process aggregation queries over unbounded streams.

Listing 4: EventHandler(Event e, Matching Record Set R)

```
   /* R is the set of matching records, each of the form (s, M).*/
1 for r ∈ R do
2     T ← LocateTrans(r.s, e.t, e.n);
3     for x ∈ T do
4         M′ ← MatchDepth(r.M, e, x);
5         if M′ ≠ NULL ∧ Evaluate(x.predicate, e) = true then
6            R ← R + {(x.target, M′)};
7            if x.type = REGULAR then R ← R − {r} ;
8            if x.op ≠ NULL then Execute(x.op, e, r.M);
         end
      end
   end
```

4. RUNTIME ENGINE

The runtime engine maintains a set of *matching records*, which are described in Section 4.1 below. These records encode matching information and predicate results for buffered items. Using the HPDT as a guide, the runtime engine responds to every input SAX event, updates the set of matching records, and executes the buffer operations. Buffer operations are described in Section 4.2. We discuss correctness in Section 4.3. We describe some implementation techniques in Section 4.4 and analyze our method's complexity in Section 4.5.

4.1 Matching Records

The runtime engine for HPDT H maintains a set R of **matching records**. Each matching record has the form (s, M), where s is a state identifier from H and M is a matching between an element and a location step. (Recall the definition of a matching from Section 2.) Listing 4 summarizes the method for updating R in response to an event e in the input. Initially, R contains a single matching record with the start state of H and an empty matching. For every incoming event e, the engine performs the following operations on every matching record $r = (s, M)$.

First, the engine uses the *LocateTrans* function to locate the set of transitions that emerge from state s and match event e. The engine performs no further operation for r if no such transitions exist. Next, for every matched transition x, the engine compares e with $r.M$ based on the type of the transition, $x.type$. The rules of the comparison are summarized in Listing 5. The `MatchDepth` function returns a new matching M'. If M' is empty, no further operation is performed for this transition, otherwise, the engine uses e to evaluate the predicate, if any, on transition x. If the predicate evaluates to false, no further operation is performed for x. Finally, a new matching record $r' = (s', M')$ is added to R, where s' is x's target state. If x is a regular transition, r is removed from R. If there is a buffer operation associated with transion x, it is executed as described in Section 4.2.

In the above scenario, when r' is added to R, we say that r **takes** the transition x on event e and **activates** r'. We also say that event e triggers the

Listing 5: MatchDepth(Matching M, Event e, Transition x)

```
   /* Returns the matching sequence M′ of the target state. */
 1 M′ ← null;
 2 switch x.type do
 3     case SELF-CLOSURE:   if e.t = B ∧ e.d > last(M).d then M′ ← M;
 4     case CLOSURE:
 5             if e.t = B ∧ e.d > last(M).d then M′ ← append(M, Element(e));
 6     case CATCHALL:
 7             if e.d > last(M).d then M′ ← M;
 8             if e.t = T ∧ e.d = last(M).d then M′ ← M;
 9     case REGULAR:
10          switch e.t do
11              case B:   if e.d = last(M).d + 1 then M′ ← append(M, Element(e));
12              caseE:   if e.d = last(M).d then M′ ← removelast(M);
13              case T:   if e.d = last(M).d then M′ ← M;
            end
        end
14 returm M′;
```

transition x and the runtime engine reaches state s'. We call a matching M **viable** if there is no element e_i in M such that p_i is false before the $(e_i, \texttt{B})$ event. The method described in `EventHandler` and `MatchDepth` is motivated by the following two properties, which establish the relationship between matching records and matchings.

Property 1. If an element e_i has a viable matching $M = (e_0, e_1, \ldots, e_i)$ with the ith location step N_i, then, immediately after the runtime engine has processed the begin event of e_i, R contains a matching record $r = (s, M)$, where s is the NA or TRUE state of a BPDT in the ith layer.

Consider the event sequence $S_e = (e_0, \texttt{B}), (e_1, \texttt{B}), \ldots, (e_i, \texttt{B})$. For every $j \in [0, i]$, $(e_j, \texttt{B})$ triggers a transition that goes down to a NA or TRUE state in a lower-layer BPDT. When that transition occurs, e_j is appended to the matching by `MatchDepth`. Let (s', M') be the matching that is activated when $(e_{j-1}, \texttt{B})$ is processed. Consider a begin event that occur between $(e_{j-1}, \texttt{B})$ and $(e_j, \texttt{B})$. If its matching end event also occurs before $(e_j, \texttt{B})$ then this pair of events either leads back to s' or, if s's is an NA state of BPDT b, leads from s' to the TRUE state of b. Thus, every element appended to M' between $(e_{j-1}, \texttt{B})$ and $(e_j, \texttt{B})$ by such begin events is removed from M' by its matching end event. If there are unmatched begin events between $(e_{j-1}, \texttt{B})$ and $(e_j, \texttt{B})$, then the query must specify e_j to be e_{j-1}'s descendant. In this case, every unmatched begin event is processed by the self-closure transition on s'. Such a transition always leads back to s' and appends nothing to the matching. (Recall that, if the jth axis is `//`, then the START state of every jth layer BPDT has a self-closure transition.)

Property 2. For every $r = (s, M)$ in R with $M = (e_0, e_1, \ldots, e_i)$ $(i > 0)$, either (1) s is a P-VALUE state of a BPDT in the ith layer and M is a matching between e_i and N_i or (2) s is a P-EVAL state of a BPDT in the $(i-1)$th layer and M is a matching between e_i and p_{i-1}.

Suppose s is a P-VALUE state. If s is the START state of a BPDT b, s must also be a P-VALUE state in b's parent b'. Suppose the START state of b' is s'. Consider the transition sequence from s' to s. According to the templates, there is a unique transition in the sequence that accepts an unmatched begin event: the one going out from s'. Since e_i must be appended when that transition is taken, the matching record that accepts $(e_i, \texttt{B})$ must be $r' = (s', M')$, where $M' = (e_0, e_1, \ldots, e_{i-1})$. If $i = 1$, the property holds since s must be the TRUE state of the root BPDT and M contains the document root (e_0) that matches the implicit N_0. If $i > 1$, using induction we can assume that M' is a matching between e_{i-1} and N_{i-1} and s' is a P-VALUE state in the $(i-1)$the layer. Since s' is also the START state of b', b' is in the ith layer and thus s is a P-VALUE state in the ith layer. Moreover, since b' is instantiated from N_i, the transition going out from s' accepts only the begin events of elements matching N_i. Since $(e_i, \texttt{B})$ is accepted by that transition, e_i must match N_i. Therefore, the property holds for r as well.

Now suppose s is a P-EVAL state. We can trace back from r to the matching record whose state is a P-VALUE state of the same BPDT. For example, if s is instantiated from state \$3 in Template 5, we can infer that it must be $r' = (s', M')$ that activates r, where s' is instantiated from \$1 in Template 5. Moreover, M' must be $(e_0, e_1, \ldots, e_{i-1})$ and e_i must be a child of e_{i-1} that evaluates p_{i-1}. If Property 2 holds for r', then it must also hold for r.

4.2 Buffer Operations

Recall that an element may have multiple matchings with a query; the element belongs to the result if at least one matching satisfies all predicates. When the runtime engine buffers an element (or its text content or attributes, as indicated by the query's output function), it stores one copy of the element for each matching. Each copy is associated with a (*matching, flag*) pair.

We now describe in more detail the buffer operations introduced in Section 3.1. Consider a buffer operation that is invoked when matching record $r = (s, M)$ activates matching record $r' = (s', M')$ on event e. Let $\texttt{max}(M, M')$ denote the longer one of M and M'. As before, the `last(M)` operation returns the last element in a matching M, while the `removelast(M)` function returns a new matching containing all but the last element of M.

- —Operation **add(f, a)** creates a new buffer item whose content is the feature a of event e. The matching-flag pair associated with this item is $(\texttt{max}(M, M'), f)$.
- —Operation **set(i)** sets f_i (the ith bit of the flag) for every buffer item whose matching contains the *target element* e_x. If s is an NA state, e_x is $\texttt{last}(M)$; otherwise, e_x is $\texttt{last(removelast(}M\texttt{))}$.
- —Operation **remove(i)** removes all buffer items with $f_i = 0$ and a matching that contains the target element $\texttt{last}(M)$.

The target element is determined by using Property 1 and considering all transitions on which the operation could reside. We use the `set(i)` operation as an

example. Suppose the `set(i)` operation is executed when $r = (s, M)$ activates $r' = (s', M')$ on event e. According to `BuildHPDT`, every `set(i)` operation is on a transition x in the ith layer. According to the templates, the source state of x, s, could be the NA state. Also, s could be a P-EVAL state that is reached from the NA state via the begin (and optionally an additional text) event of the child `c`. In the first case, according to Property 2, M must be a matching between `last(`M`)` and N_i. Since the transition x can only accept the begin event of a child of `last(`M`)`, it must be `last(`M`)` that satisfies p_i on this event e. Therefore, we should operate on `last(`M`)`. In the second case, according to Property 2, M must be a matching between `last(`M`)` and p_i. Since `last(`M`)` can only evaluate the predicate p_i for its parent, we should operate on the parent of `last(`M`)`, which is `last(removelast(`M`))`. Thus, the target element e_x for an operation *set*(i) is the element that has just safisfied its predicate.

Another important feature of the buffer operations is that the flags are always set at the earliest possible moment. First, from the templates we can see that the `remove` operation is always invoked when a predicate evaluates to false. Next, a `set(i)` operation is always invoked when an element e_i satisfies its predicate if e_i has a matching with N_i. Let us consider the event e that evaluates the predicate to true for e_i. This event e could be $(e_i, \texttt{B})$, $(e_i, \texttt{T})$, $(c_i, \texttt{B})$, or $(c_i, \texttt{T})$, where c_i is the child of e_i that satisfies its predicate. In accordance with Property 1, $(e_i, \texttt{B})$ must be processed by the engine. In all four cases, since there are no unmatched begin events between $(e_i, \texttt{B})$ and e, e must be processed as well and thus the `set(i)` operation is executed.

Example 4. Consider the runtime engine, with the HPDT in Figure 10 for the query `//pub[year > 2000]//book[author]//name/text()`, operating on the stream of Figure 2. When the begin event of the `name` element on line 11 is encountered, there are three matching records with the state \$8, which accepts this begin event, and different matchings:

M_1: document root, `pub` on line 2, `book` on line 7
M_2: document root, `pub` on line 2, `book` on line 10
M_3: document root, `pub` on line 9, `book` on line 10.

We use M_i^- to denote the prefix of M_i without the last `book` element and M_i^+ to denote the longer matching obtained by appending to M_i the `name` element on line 11. When the text content of the `name` element is buffered, three copies of it are created with three different matching-flag pairs: (M_1^+,1001), (M_2^+,1001), and (M_3^+,1001), which are also used below to refer to the buffer items.

On encountering the begin event of the `author` element on line 12, both matching records (\$8, M_2) and (\$8, M_3) process this event and execute the `set(2)` operation on the copy whose matching contains the `book` element on line 10 (the tail element of M_2 and M_3) . Therefore, the three copies are now (M_1^+,1001), (M_2^+,1011), and (M_3^+,1011). Two new matching record, (\$10, M_2) and (\$10, M_3) are activated after the end event of the `author` element is processed.

On encountering the end event of the `book` element on line 13, (\$10, M_2) and (\$10, M_3) both take the transition from \$10 to \$3. The two matching records

(\$3, M_2^-) and (\$3, M_3^-) are already in R, since they activated (\$8, M_2) and (\$8, M_3) at the begin event of the book element on line 10. They stayed in R because of the self-closure transition on \$3.

On encountering the end event of the pub element on line 15, only matching record (\$3, M_3^-) takes the transition from \$3 to \$2 and the `remove(1)` operation is executed. Since the buffer item (M_3^+,1011)'s matching contains the pub element on line 9 (the tail element of M_3^-) and its f_1 is 0, it is removed from the buffer. The other two copies stay in the buffer.

On encountering the end event of the book element on line 16, only matching record (\$8, M_1) takes the transition from \$8 to \$3 and the `remove(2)` operation is executed. Since the buffer item (M_1^+,1001)'s matching contains the book element on line 10 (the tail element of M_1) and its f_2 is 0, it is removed from the buffer. However, the buffer item (M_2^+,1011) is not removed since neither does its matching contain the book element on line 10 nor is f_2 0. The item is updated to (M_2^+,1111) on encountering the text event of the year element on line 17 and consequently sent to output.

4.3 Correctness

We now outline the correctness of the above method. To simplify the description, we assume the buffer items are created for whole elements instead of their features (such as attributes). We wish to show that an element e_k has a matching M that satisfies all the predicates if and only if there exists a buffer item that is created for e_k and is associated with the pair $(M, 1^*)$ (where 1^* denotes the true flag). The following property is useful for this purpose.

Property 3. Suppose element e_k has a matching $M = (e_0, e_1, \ldots, e_k)$ with N_k. A matching record $(b(k,m).\text{START}, f)$ is active upon encountering $(e_k, \texttt{B})$ if and only if e_i has satisfied p_i for all $i \in [0, k-1]$ such that $m_i = 1$.

Consider a BPDT $b(i, j)$ in the ith layer. Its position, j, has i bits since $j \in [0, 2^i - 1]$. Consider now the two child BPDTs $b(i+1, 2j)$ and $b(i+1, 2j+1)$. The first i bits of their positions are copied from j and the last bits are determined by the state by which they overlap with $b(i, j)$. (If $j = (j_0 j_1 \cdots j_{i-1})_2$, $2j = (j_0 j_1 \cdots j_{i-1} 0)_2$ and $2j+1 = (j_0 j_1 \cdots j_{i-1} 1)_2$.) It is easy to see that $b(k, m)$ copies the ith bit in its position (m) from the ancestor b in the $(i+1)$th layer: If b.START is its parent's TRUE state, $m_i = 1$; otherwise, $m_i = 0$. In other words, $m_i = 1$ if and only if a TRUE state in the ith layer is reached during the transition sequence from the START state of the HPDT to the START state of $b(k, m)$. Therefore, when the $b(k, m)$.START is reached and associated with a matching $M' = (e_0, e_1, \ldots, e_{k-1})$, we know that, for every $m_i = 1$, a TRUE state in the ith layer has been reached. Moreover, since only the elements in M' and their children (used for predicate evaluation) may trigger the transitions, we know $m_i = 1$ only if e_i has satisfied p_i.

Suppose e_i satisfies p_i before $(e_k, \texttt{B})$. Let e be the event that satisfies p_i for e_i. An examination of the templates reveals that if the transition that accepts e is not connected directly to the TRUE state in the BPDT the the TRUE state must be reached later. Thus, for every e_i that satisfies p_i before $(e_k, \texttt{B})$, a TRUE state in the ith layer must be reached before $(e_k, \texttt{B})$. Therefore, upon encountering

(e_k, B), the engine reaches the state $b(k,m)$.START, where $m_i = 1$ if e_i has satisfied p_i.

Output ⇒ Result. Suppose a buffered element e_k is associated with a true flag and a matching $M = (e_0, e_1, \ldots, e_k)$. We wish to show that (1) M is a matching between e_k and N_k and (2) e_i ($i \in [0, k]$) satisfies p_i, the predicate of N_i. Result (1) follows directly from Property 2 and an enumeration of the possible output functions and the corresponding translated operations. If the ith bit of the flag of e_k, f_i, is 1, either the ith bit of initial flag of the buffer item is already 1 or f_i is set by a `set(i)` operation. We have shown in Section 4.2 that `set(i)` set f_i for e_k (with matching M) only if M contains e_x and e_x has just satisfied p_i. Therefore, we only need to show that the ith bit of the initial flag for e_k (with matching M) is 1 only if e_i satisfies p_i. In accordance with the definition of `add` operation, in BPDT $b(k,m)$, the initial flag is $2m+1$ if the transition on which the operation reside is connected to a TRUE state, or $2m$ otherwise. For the kth bit of the initial flag, it is 1 only if p_k evaluates to true for `last(`M`)` (otherwise, the TRUE state would not be reached). For any other bit f_i, it is 1 if only if $m_i = 1$. In accordance with Property 3, since the START state of $b(k,m)$ has been reached, m_i is 1 only if e_i satisfies p_i.

Result ⇒ Output. Suppose an element e_k has a matching $M = (e_0, e_1, \ldots, e_k)$, e_i ($i \in [0, k]$) that satisfies p_i. We wish to show that there exists a buffer item created form e_k with matching M and a true flag. By Property 1, e_k will be processed by the engine using (the transitions in) a kth layer BPDT and be added to the buffer. Let us consider the event e that evaluates p_i to true for e_i. If e_k is buffered before e, f_i for e will be set for e_k when e is processed. If e_k is buffered after e, there are two cases. First, e_k is buffered before the TRUE state in the ith layer is reached (but e_i has satisfied p_i). This case could happen only if e_k is a descendant that is nested in c_i. The f_i is set for e_k by the extra `set` operation added by the `AddExtraSet` procedure. Second, e_k is buffered after the TRUE state in the ith layer is reached. By Property 3, upon encountering (e_k, B), the engine has a matching record with the START state of a BPDT $b(k,m)$, where $m_i = 1$. Therefore, e_K must be buffered by an `add` operation with an initial flag $2m$ or $2m+1$, both of which have $f_i = 1$.

4.4 Implementation

4.4.1 *Depth Stack.* Instead of storing a matching as a sequence of the elements, we use a *depth stack*: a stack consisting of the depths of the matching's elements. In Listing 5, when an element is to be appended to the matching, we push its depth onto the depth stack. When the rightmost element of a matching is to be removed, we pop the top item off the stack. In `MatchDepth()`, we only need to compare the depth of the rightmost element in a matching with the depth of the current event. Thus, using a depth stack is equivalent to using a matching for this function. Recall that when the engine executes a buffer operation `set(i)` or `remove(i)`, it operates only on the buffer items whose matchings have a specified element e_i that matches N_i. Every element after e_i in the matching must be closed because no event from e_i's descendant can invoke `set(i)`. Further e_i is always the element that is being processed so that every element

before e_i in the matching is currently open. Since no two open elements can have the same depth, the depth stack uniquely specifies the useful prefix of the matching. Thus, all the necessary operations on matchings can be performed on their depth-stack representations instead.

Depth stacks are stored as integers and operations on the depth stacks are implemented as bitwise operations on the integer representations. For example, if the depth stack is $(0, 1, 2, 5)$, the integer representation is 111001. That is, the ith ($i \geq 0$) 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.

4.4.2 *Global Queue.* XSQ maintains a global queue that contains a single copy of each buffered data item, irrespective of the number of times the item has been buffered. Recall that an item may be buffered multiple times, with different (matching,flag) pairs. In such buffer entries, XSQ stores a pointer to the corresponding items in the global queue. When the flag of any such buffer entry becomes a true flag, the corresponding data item in the global queue is marked for output and no further operations are performed on it. The document order of result items is preserved (as required by XPath) by outputting data items only when they are at the head of the global queue. That is, even if an item is marked for output, it is not emitted as output until the items ahead of it in the global queue are either removed or emitted.

4.4.3 *Buffer Segmentation.* Consider buffer item b_1 with matching $(e_0, e_1, \ldots, e_i, e_{i+1}, \ldots, e_k)$ and flag $f = (f_0 f_1 \cdots f_i f_{i+1} \cdots f_k)_2$ and, similarly, b_2 with matching $(e_0, e_1, \ldots, e_i, e'_{i+1}, \ldots, e'_k)$ and flag $f = (f_0 f_1 \cdots f_i f'_{i+1} \cdots f'_k)_2$. If $f_i = 0$ and $f_j = f'_j = 1$ for $j > i$, then any future `set(x)` or `remove(x)` operations will be always applied to b_1 and b_2 at the same time. To take advantage of this feature, we group buffer items (pointers) based on the longest prefixes of their matchings that have the last element's predicate pending. Since we store depth stacks instead of matchings, we use function **remain(**ds, f**)** to return the prefix of ds of length $i + 1$ where i is the largest value such that $f_i = 0$. If a pointer is associated with the pair (ds, f), the pointer belongs to a group with key `remain`(ds, f). The group is also associated with a single flag f, called the *group flag*. When a buffer item is first created with depth stack ds and initial flag f, it is added to a group with the key `remain`(ds, f). When a `set(i)` operation is executed on the buffer items whose matchings have an element e_i that, in turn, has a matching ds with N_i, we simply set f_i for the group flag f of the groups with ds. Since the result flag f' has a new right-most zero-bit, the whole group is appended to anther group with key `remain`(ds, f'). For a `remove(i)` operation, we simply delete the group with key ds. In our implementation, all the groups are organized as a hash table. The key is the depth stack and the pointers in the group is organized as linked list. All the marking operations are executed on groups of pointers.

4.5 Complexity

A detailed experimental study of the time and space efficiency of XSQ appears in Section 6. Below, we provide a very simple analysis of the construction-time and runtime complexity. For the construction-time complexity, we assume that the input query is in a parsed form and that string operations take unit time. The dominant factor in the construction is the number of BPDTs in the HPDT. For the runtime complexity, we assume that each depth-stack operation takes unit time. (See Section 4.4.) The function `remain`(ds, f) can be computed in constant time as follows. Since the rightmost nonzero bit of every flag f can be computed in advance, given the depth stack ds stored as an integer, the `remain`(ds, f) function is a simple bitwise operation. Target groups in the buffer can be located using the hash table in constant expected time. Thus, buffer operations, such as appending an item, deleting a group, and appending a group to another group, can be performed in constant expected time. Strictly speaking, some depth-stack operations, which are implemented using bitwise operations, and some buffer operations, which operate on only pointers in groups, may require non-constant time given arbitrary inputs. (For example, XSQ's implementation of depth stacks using constant-size integers does not work if the input XML has truly unbounded depth.)

4.5.1 *Construction-Time.* Recall the construction of HPDTs summarized in Listing 1. The worst case occurs when every location step has a predicate. In this case, the construction creates 2^{k-1} BPDTs for an XPath query with k location steps. Creating a BPDT requires constant time for the tasks of finding the matching template, initializing a constant number of states and transitions, and adding and changing a constant number of operations. (The number of templates, states, transitions, and the number of items to check for template matching, are all bounded by a small constant.) Therefore, the space and time cost of construction is bounded by $O(2^k)$. Although the exponential dependence on query length may seem problematic at first, the space cost of the HPDT is typically completely dwarfed by the space cost of buffering data at runtime.

4.5.2 *Runtime.* Recall the runtime actions summarized in Listing 3.4. First, by examining the templates, we note that given a source state and an event, the `LocateTrans` function returns at most two transitions. Listing 5 suggests that the `MatchDepth` function also requires only constant time. The main determinant of the complexity is the number of matching records that need to be processed in the outer for-loop of Listing 3.4. If the query contains no // axis, the HPDT is free of closure and self-closure transitions. In this case, there is only one matching between a location path and an element. There can be only one or two (in the case of catchall transitions) matching records at any time. Therefore, each event can be processed in constant time. If the query contains // axes, there will be multiple matching records because of the closure and self-closure transitions. Since there are at most 2^{i-1} BPDTs generated to process the ith location step N_i, we have at most 2^i states (2 states in each BPDT) associated with a matching of length $i+1$. The number of ways that an element at depth d can match N_i is bounded by $\binom{d}{i}$. Therefore, the number of matching records

(and thus the processing time per event) is bounded by $\sum_{i=1}^{k} 2^i \binom{d}{i}$, where k is the query length and d is the maximum depth of an element. However, typical query-data combinations do not permit all the combinations for matchings assumed by the above calculation; thus the bound is not likely to be reached in practice. (See Section 6.)

5. 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; Gupta and Suciu 2003]. 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. The *XFilter* system [Altinel and Franklin 2000] uses an indexed automaton-based method for filtering a stream of documents by a large number of XPath filter expressions. The *YFilter* system [Diao et al. 2002] addresses a similar problem and uses a single, combined automaton to evaluate all submitted filter expressions. The *XTrie* data structure indexes XPath queries based on common substrings [Chan et al. 2002]. To support predicates and boolean operators in XPath filters, the XPush machine [Gupta and Suciu 2003] uses *alternating automata* [Chandra et al. 1981], in which each state has a flag indicating the acceptance or rejection. The flag of a state is computed from its offspring states based on the type of the states: *universal*, *existential*, or *negating*.

Green et al. [2003] introduce a lazy deterministic finite-state automaton that may be used to obtain the speed benefits of deterministic automata (fast matching of input events) without incurring high memory costs. The main idea is to add states at runtime to the finite state automaton obtained directly from the XPath filters. Their method provides an upper bound on the size of the runtime automaton. Although our method may also be characterized as a lazy method, there are some important differences in the manner of lazy instantiation. In the lazy automaton approach, when an unknown tag is encountered in the stream, the automaton forks a new state that is used to process the new element. In our method, when an unknown tag is encountered, it is ignored and no operation is performed (unless it is accepted by a self-closure transition or a catchall transition). In other words, the closure transition is implemented using dynamic states in the lazy automaton, while is enforced by the new types of transitions in XSQ.

The problem of *query labeling* can be also viewed as a type of filtering problem because it labels the data in the stream with query identifiers. The *requirements index* and a framework for efficiently organizing it have been proposed for this task [Lakshmanan and Sailaja 2002]. The problem of validating XML streams using pushdown automata [Segoufin and Vianu 2002] is also relevant because validation may be thought of as filtering for documents that match a given *Document Type Definition (DTD)* [Bray et al. 2000].

The above methods address the problem of *filtering* using XPath and are not directly applicable to *querying*. However, it may be possible to extend some of the filtering ideas to querying. (For example, the XMLTK system [Avila-Campillo et al. 2002] uses lazy DFAs to evaluate XPath queries.) There are some similarities between the methods for filtering and querying. For example, both XPush and XSQ encode predicate results in automaton states. XPush uses the flags associated with states while XSQ uses the hierarchical arrangement of BPDTs to encode this information in the states themselves. However, an important distinction, which stems from their different design goals, is that filtering systems keep track of the matching status of every *document*, while querying systems keep track of the matching status of every *element*. Thus, a filtering system does not need to buffer different elements of a document, as a querying system must do. Further, problems due to multiple matchings, such as those highlighted in Example 2, so not arise in filtering systems. Our work in this article, therefore, may provide a method to support features such as closures and multiple predicates when applying the filtering methods to a querying system.

The *XML Stream Machine (XSM)* [Ludascher et al. 2002] is a transducer-based approach to evaluating *XQuery* queries on XML streams. A network of XSMs, each generated from a subexpression of a decomposed query, is merged into a single XSM that can be optimized if the DTD for the input is available. The transducer network model called *SPEX* [Olteanu et al. 2002] follows a similar approach: each transducer in the network is generated from a regular path expression construct. XSM supports XQuery constructors while XSQ supports only XPath (no constructors). On the other hand, XSM does not support XPath features such as aggregations, closures, and multiple predicates. DTD-based optimizations could be applied in XSQ. For example, if the DTD suggests that the `author` is the child of the `book` while a query contains `book//author`, we can rewrite the query into `book/author` to reduce the number of closure transitions in the HPDT. Currently, we do not have a systematic method for such optimizations.

The *XAOS* system [Barton et al. 2002, 2003] for streaming XML supports XPath's reverse axes, such as `parent` and `ancestor`. To to reduce the amount of streaming data buffered in a *matching structure*, XAOS uses two data structures: a *X-tree*, which is the parse tree of the XPath expression with reverse axes permitted, and a *X-dag*, which is the equivalent XPath representation with reverse axes removed. The X-dag is used as a pattern to filter the incoming stream to remove irrelevant nodes. The relevant nodes are stored in the matching structure based on their relations in the X-tree. When the stream ends, results are produced by traversing the matching structure. Unlike XAOS, XSQ buffers only potential result items and outputs a result item as soon as its membership in the result is determined. (At the time of writing, XSM and XAOS were not available for testing and they are therefore omitted from our study in Section 6.)

A recent study of the query complexity of XPath includes a main-memory algorithm (nonstreaming) with polynomial combined complexity (query and data) [Gottlob et al. 2002]. The paper identifies a subset of XPath, called *Core XPath*,

that can be evaluated in linear time. *Core XPath* has boolean operators and axes other than / and //, which are not in the subset used in XSQ, but has no value comparisons and aggregates, which are used in XSQ. The algorithm is based on reducing every axis to two primitive axes: *first-child* and *next-sibling*. Although the polynomial complexity of this method is attractive (compared to the exponential complexity of XSQ), the algorithm requires several passes over the data as it evaluates nodes in the XPath parse tree in a bottom-up manner. Thus, it is not clear how it could be extended to a streaming environment. In general, it is not straightforward to compare streaming and nonstreaming algorithms because streaming systems are typically limited to an smaller subset of XPath.

A follow-up paper [Gottlob et al. 2003] further shows that the combined complexity of Core XPath, and therefore of XPath, is P-hard. It also analyzes the complexity of various subsets of XPath and shows that negations and scalars (numerical and string functions) in the query are the sources of difficulty in evaluation. Without these features, and without some other complex XPath constructs, such as nested predicates of the form of `[p1]...[pn]`, XPath queries are highly parallelizable. Another paper [Benedikt et al. 2003] addresses the expressiveness of different subsets of XPath and the containment relations among them. It also provides rules for different subsets to simplify XPath queries into a normal form defined by that paper. These papers suggest that most of XPath can be evaluated very efficiently. Nonetheless, streaming evaluation requires further study, especially if result items are to be emitted as early as possible and if only potential result items are to be buffered.

The features of queries and data in a streaming environment have been investigated in a general setting [Babcock et al. 2002]. Some of the more general issues are also applicable to semistructured data and XML. For example, we may wish to consider different semantics for streaming XPath queries with aggregations (up-to-date result, a window-based result, or an approximation). Some ideas suitable for relational data streams may not be directly applicable to XML streams. For example, the dispatching of incoming data to multiple applications based on selections is easy in relational data. However, since evaluating XPath queries usually requires information about the document tree (to get the context of an element), the dispatching task is more complex for XML streams. For example, we may need to add the ancestors of a target element to it when it is dispatched.

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 is when the provided input includes constraints on forthcoming data. For relational data, there are methods for embedding *punctuations* in streaming data, facilitating the streaming evaluation of queries that include blocking operators such as *group by* [Tucker et al. 2003]. It should be interesting to use similar ideas for streaming XML to support XPath queries that include axes such as *following*.

Several systems provide methods for querying non-streaming XML data. *Galax* [Fernández and Siméon 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. Another topic closely related to XPath query processing is *XML transformation*. *XSLT* is a standard template-based language for transforming XML [Kay 2003]. Since XSLT uses XPath to specify patterns in its rules, XSQ and other methods for XPath processing have applications in XSLT processors. It is difficult to adapt these systems for streaming data because they usually require an in-memory materialization of the entire XML document. However, they provide some baselines for our experimental study in Section 6.

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. A method that generate STX templates equivalent to an XPath queries that require buffering is not immediately apparent. However, this approach is an alternative to our automaton-based approach and deserves further attention.

Prior work [Miklau and Suciu 2002] has noted that there are important differences between XPath evaluation and the classical problems of *tree pattern matching* [Hoffmann and O'Donnell 1982; Chen et al. 2001] and *unordered tree inclusion* [Kilpel 1992]. In particular, the problem of unordered tree inclusion is NP-hard (by direct reduction from SAT), 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, an algorithm described for the classical tree pattern matching problem [Hoffmann and O'Donnell 1982] 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 the order of siblings in the data matches the their order in the pattern. In contrast, 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.

6. EXPERIMENTAL EVALUATION

The goals of this experimental study include validating the XSQ implementation, characterizing its features and performance, and providing an *exploratory* description of the features and performance of systems that are related to XSQ.

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	—
Joost	STX	X		X	X	

Fig. 11. System features.

We stress that our experiments are not designed for a head-to-head micro-benchmark-style comparison of the systems we study. Given the diversity of the systems in goals, supported query languages and features, implementation language and environment, state of development, etc., such a comparison would not be easy. Rather, we wish to gain some qualitative insights into the cost of supporting certain XPath features such as closures and to study which systems and features are best suited to a given environment.

We begin by describing our experimental setup in Section 6.1. We describe results on throughput in Section 6.2, latency in Section 6.3, and memory usage in Section 6.4. Section 6.5 presents a broader study of a set of query engines aimed at characterizing their features and performance. Section 6.6 presents an experimental characterization of XSQ.

6.1 Experimental Setup

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. The former implementation uses a deterministic automaton leading to performance benefits. We conducted our experiments on a PC-class machine with an Intel Pentium III 900 MHz processor with 1 GB of main memory running the Redhat 7.2 distribution of GNU/Linux (kernel 2.4.9). To ensure the evaluation is performed only in the main memory, 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 11. As the figure suggests, these systems vary considerably in their design goals and features. Many do not support streaming evaluation, and many are main memory systems that evaluate the query on the document tree of the data built in memory. We have discussed XQEngine [Katz 2002] (version 0.56) and XMLTK [Avila-Campillo et al. 2002] (version 0.9) in Section 5. An implementation of STX, Joost [Becker 2002] (version 20020828), and an implementation of XSLT, Saxon [Kay 2002] (version 6.5.2), are also studied here. 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.

Name	Size (MB)	Text size (MB)	Num. of elements (K)	Depth avg	Depth max	Avg. tag length	Parsing Time(s) Xerces	Parsing Time(s) Expat
SHAKE	8	5	180	5.77	7	5.03	1.42	0.43
NASA	25	15	477	5.58	8	6.31	4.35	1.50
DBLP	119	56	2,990	2.90	6	5.81	27.60	7.53
PSD	716	286	21,300	5.57	7	6.33	170.00	66.40
RECURS	10	9	96	22.30	26	5.31	1.65	0.43
RECURB	121	105	963	26.00	30	5.31	13.00	4.82

Fig. 12. Dataset descriptions.

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 2003], and the PIR-International Protein Sequence Database (PSD) [Wu et al. 2002]. Since these datasets have relatively shallow structures, we generated two synthetic datasets, *RECURS* and *RECURB*, using IBM's XML Generator with deeper document structure to explore features related to such data. Some characteristics of these datasets are listed in Figure 12. In Section 6.5, We also use Toxgene [Barbosa et al. 2002] to generate synthetic datasets that contain specified number of designated elements.

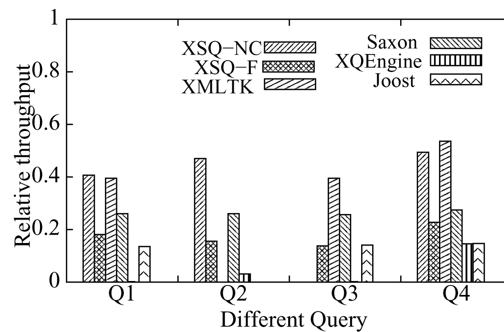
To the best of our knowledge, there are no standard or widely used benchmarks for XPath queries. Therefore, following other work on this topic (e.g., Ludascher et al. [2002] and Barton et al. [2003]), we conduct our experimental study using **queries** that vary in a variety of features that are likely to influence performance, such as query length, number of predicates, and types of axes. The queries used for each experiment are listed near the figures summarizing the results.

For a text-based data format such as XML, **parsing** the input typically accounts for a substantial fraction of the running time. The last two columns of Figure 12 list the parsing times for our sample datasets. We also note that parsing times vary widely across systems, depending on the parser and programming environment. In order to prevent these differences from masking the effects of query processing, we normalize the running time of each system using its parsing time.

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 such tight confidence intervals graphically, the conventional error-bars are omitted in the graphical results that follow.

6.2 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



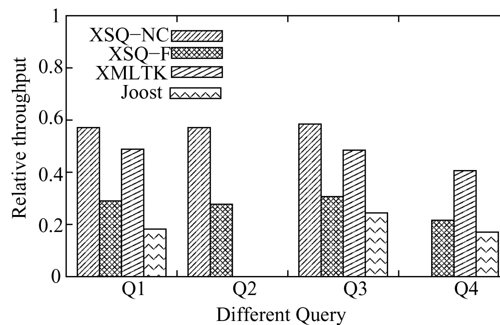
Q1: `/PLAY/ACT/SCENE/SPEECH/SPEAKER/text()`

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

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

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

Fig. 13. Relative throughputs for different queries on the SHAKE dataset.



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

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

Q3: `/article@key`

Q4: `//title`

Fig. 14. Relative throughputs for different queries on the DBLP dataset.

(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. (For infinite streams, the average throughput at a point in the stream is obtained by dividing the amount of data processed up to that point by the amount of processing time expended up to that point.)

As noted earlier, parsing often accounts for a significant fraction of the processing time and may mask the differences due to query processing proper. Therefore, we define **relative throughput** of a system to be its throughput divided by the throughput of the parser used by that system.

Figures 13, 14, 15, 16, 17, and 18 summarize our experiments comparing the relative throughputs of the systems over different datasets and queries. Results

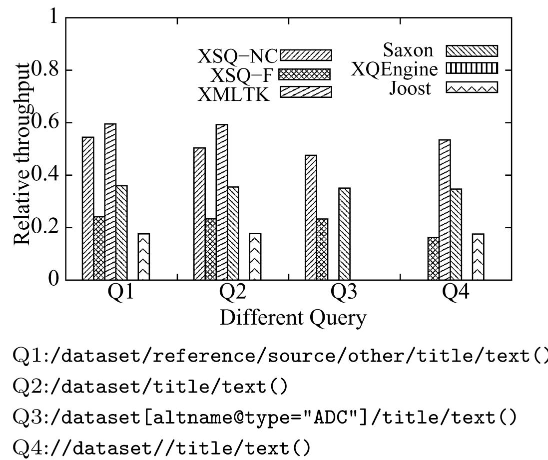


Fig. 15. Relative throughputs for different queries on the NASA dataset.

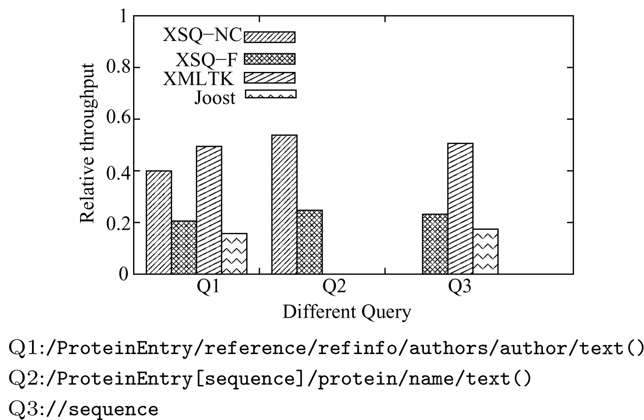
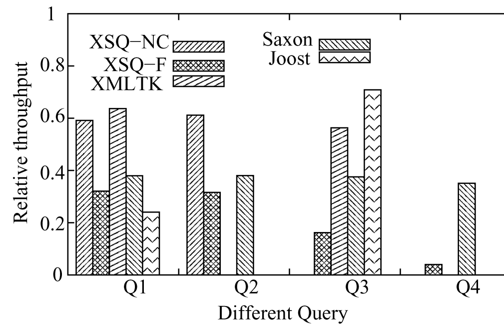


Fig. 16. Relative throughputs for different queries on the PSD dataset.

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.

We observe that, in general, XMLTK and XSQ-NC are the two fastest systems when we use simple queries that they support. XMLTK supports only predicates that can be evaluated at the time a potential result element is encountered, such as a predicate on that element's attribute. Therefore, XMLTK can always output a result item at the time it appears in the stream. The HPDT used in XSQ-NC is deterministic, which means there is only one active matching record and at most one matching transition for the incoming event. It is one reason that XSQ-NC is faster than XSQ-F since XSQ-F may have multiple active matching records and multiple matching transitions for an incoming event.

However, even for the same query without closure, XSQ-NC is faster than XSQ-F although XSQ-F also has only one active matching record in this case.



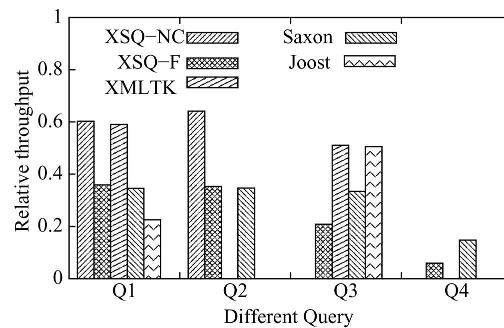
Q1:`/pub/book/title/text()`

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

Q3:`//proceedings/@category`

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

Fig. 17. Relative throughputs for different queries on the RECURS dataset.



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

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

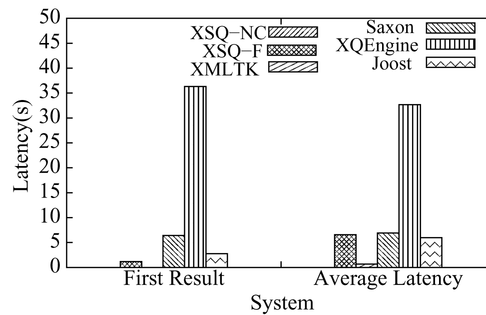
Q3:`//proceedings/@category`

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

Fig. 18. Relative throughputs for different queries on the RECURB dataset.

One reason is that XSQ-NC does not need mechanisms such as depth stacks to keep track of possible multiple matchings. Moreover, XSQ-F always buffers a potential result item b first even if b is known to be in the result when it comes in. XSQ-F then marks b as output, checks the queue, and outputs b if b is at the head of the queue. This mechanism is used only in XSQ-F since, due to the existence of closure axes, there may be other undecided items in the buffer before the current buffer item. Without closure axes, if we can determine b is in the result, we can always output b right away. In this case, we conclude that every b's ancestor matches a location step in the query and satisfies the predicate. Therefore, there cannot be any undecided buffered items, since a buffered item can only wait for an open element whose predicate is pending. We study these issues further in Section 6.5.

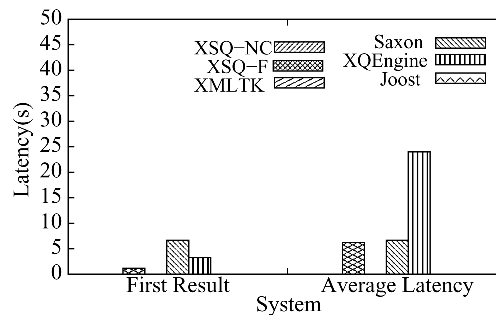
Figures 13, 15, and 17 suggest that Saxon is faster than XSQ-F when they process XML data that can fit into main memory. Saxon loads all the data



Query: `/PLAY/ACT/SCENE/SPEECH/SPEAKER`

Note: For XQEngine, the average latency bar has been scaled down by a factor of 10.

Fig. 19. Latency on the SHAKE dataset.



Query:`/PLAY/ACT/SCENE/SPEECH[LINE contains "love"]/SPEAKER`

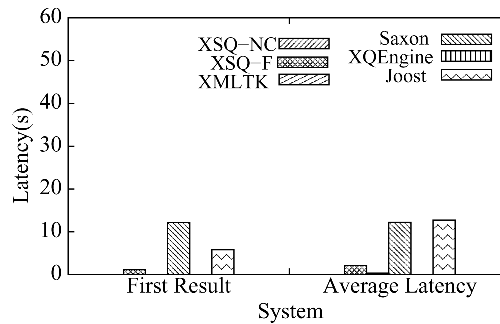
Fig. 20. Latency on the SHAKE dataset.

into the memory to build the DOM-tree before it evaluates the query. After parsing the data, Saxon performs all the processing in main memory. Such in-memory processing is efficient and can support more complex features such as the whole set of XPath axes. However, the main memory approach is not suitable for streaming data in general.

6.3 Latency

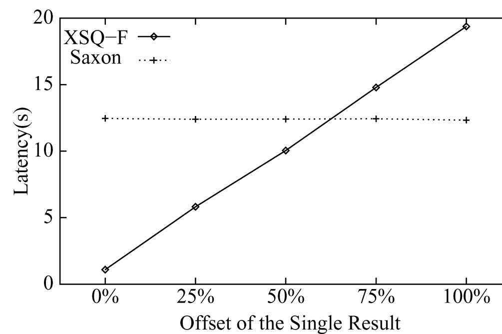
Output latency is an important property of streaming systems, and we measure it as follows. We let every system output the result to standard output. For a query that returns an element with name `N`, we monitor the standard output to detect the start-tag `<N>` and record the elapsed time when we receive each such tag. (The clock is started at the time the system begins evaluation.) For each result item, we refer to this time as its latency and define the latency of the query result to be the average latency for all items in the result.

Figures 19, 20, 21, and 22 summarize our results on the output latency. In the first three figures, the left parts illustrate the time when the first result elements are returned. The right parts of the figures illustrate the average output latency of result elements. We note from Figure 19 that the streaming



Query: `/datasets/dataset/reference/source/other/title`

Fig. 21. Latency on the NASA dataset.

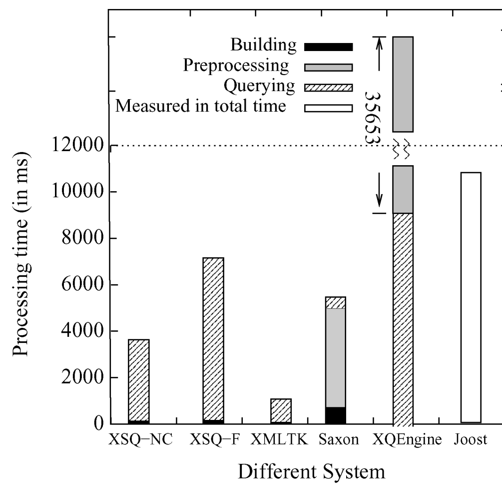


Query:`/datasets/dataset[altname=x]/title`
Note: The query returns a single result item at the offset in the document that we selected using different value of `x`.

Fig. 22. Latency on the NASA dataset.

systems usually output the first result item earlier than the nonstreaming systems. (XSQ-NC and XMLTK returned the first element immediately after the systems were invoked.) This result is as expected since the nonstreaming systems need to load all the data and build the document tree in memory before actual query evaluation begins. The average latency for the nonstreaming system Saxon is very close to its latencies for the first returned element. The reason is that it always evaluates the whole query first and then returns the result when the whole result set is available. Since the XQEngine version we tested cannot handle documents with more than 32,767 elements, we divided datasets into a sequence of smaller documents as needed to satisfy this constraint. Therefore, XQEngine returned the first result item after it finished processing the first small document. Also, we note that the bar depicting average latency for XQEngine has been scaled down 10-fold in order to fit in the chart.

In Figure 20, we used a query that contains a predicate testing whether the text content contains a string. Besides results similar to those in Figure 19, we



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

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

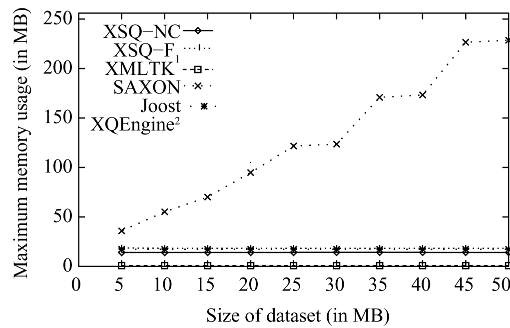
Fig. 23. Preprocessing time, query processing time, and total querying time.

notice that XQEngine returns the first result very quickly and that its average latency is also lower than that in Figure 19. This result is explained by recalling that XQEngine builds a full-text index for the XML document, and can therefore efficiently evaluate queries that require string lookups of this kind.

We used the NASA dataset for the next set of experiments. Figure 21 illustrates that for this larger (23MB) dataset, the latencies of the first result items in the streaming systems are much smaller than those in the nonstreaming systems. This result is as expected since the nonstreaming systems now need to load a larger dataset before they output the first result item. We also observe that the average latencies of XSQ and XMLTK are much smaller than those of the nonstreaming systems, while the average latency of Joost is still almost the same as that of Saxon. After examining the result, we discovered that Joost uses buffered output. Since the result size of this query is twice the buffer size, the result items are emitted in two groups.

We note that the nonstreaming systems may return results faster. In Figure 22, we used a query that returns a single element. By selecting the element at different positions in the stream, we observe that the latency for XSQ is almost proportional to the size of data before the result element. In contrast, Saxon's latency is almost constant since the position of the element is not important for its main-memory query evaluation.

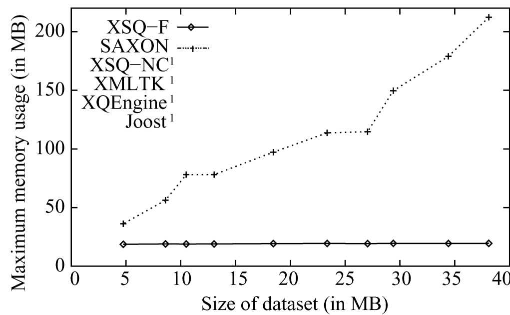
Figure 23 illustrates the result of measuring the components of the overall query-processing time. Although the figure depicts the result for one query, the results are similar for other queries we used. The dark bar represents the query compilation time, which usually includes parsing the query and building the data structures used by the runtime 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.



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 file.

Fig. 24. Memory usage for DBLP-based datasets of different sizes.



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

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

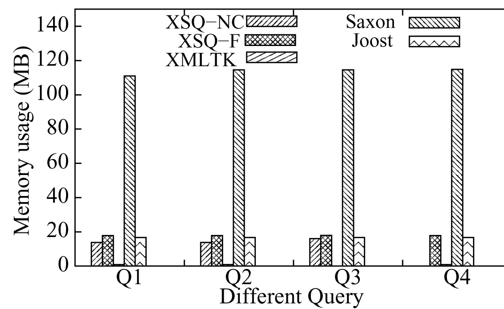
Fig. 25. Memory usage for synthetic datasets of different sizes.

Similarly, XQEngine preprocesses data by building a full-text index on the data before evaluating any queries.

In general, one benefit of the nonstreaming systems is that, as long as the preprocessed data in these systems remains in memory, subsequent queries can be evaluated very efficiently by reusing the preprocessed data.

6.4 Memory Usage

The main memory required by a streaming query engine is an important metric and often determines its feasibility for an application. Figures 24, 25, 26, 27, 28, and 29 summarize the results of our experiments comparing the memory usage. We observe that, as expected, the streaming systems typically use much less memory than the nonstreaming systems. We also note that, for different datasets, the streaming systems use almost the same amount of memory. This fact suggests that the amount of memory used by the streaming systems is



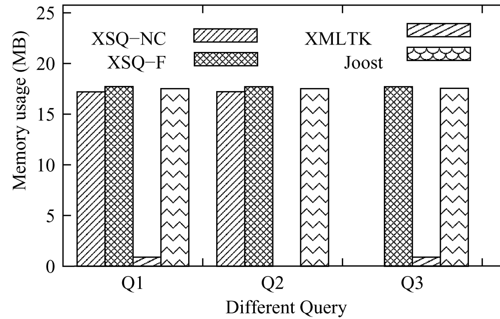
Q1:`/dataset/reference/source/other/title/text()`

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

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

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

Fig. 26. Memory usage for different queries on the NASA dataset.



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

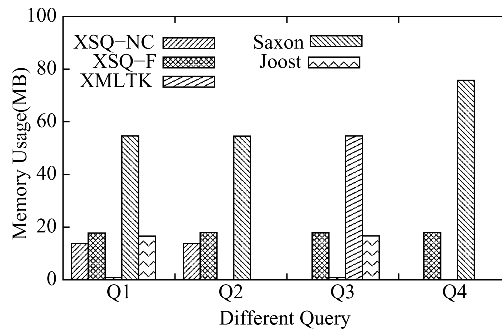
Q2:`/ProteinEntry[sequence]/protein/name/text()`

Q3:`//sequence`

Fig. 27. Memory usage for different queries on the PSD dataset.

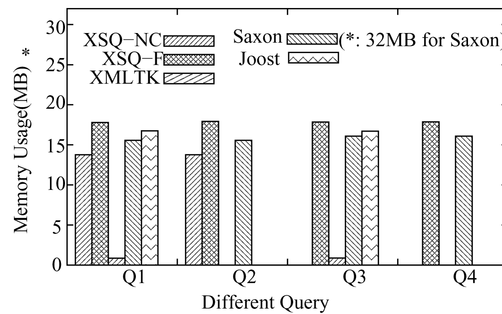
only weakly dependent on the size of the datasets. For systems such as XMLTK and Joost, this observation is always true since no data is buffered during the evaluation. However, systems that support predicates, such as XSQ-NC and XSQ-F must buffer data and the amount of buffered data may be large, depending on the dataset and query. Further experiments studying this aspect of XSQ are described in Section 6.6.

We also used the XML Generator program to generate datasets of varying size and recursiveness. For example, for the dataset of size 13 MB, the nested level parameter of the XML Generator program is set to 15 and the maximum repeats parameter is set to 20. From Figure 25 we note that even with highly recursive data and queries with closures, the memory used by XSQ-F is almost constant. Since all the items in the buffers can be determined when we encounter the end event of the element matching the first location step, the maximum amount of memory that XSQ needs does not exceed the size of the largest element in the stream.



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. 28. 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. 29. Memory for different queries on the RECURB dataset.

6.5 Characterizing the XPath Processors

Since streaming query engines need to buffer potential results items, the relative ordering of XML elements in a dataset may influence the amount of buffer space needed. To study the effect of element order, we generated a 10 MB dataset using Toxgene, by applying the template of Figure 30 repeatedly to generate new elements with successive `id` attributes. The result dataset contains 128 `A` elements, each of which has a non-zero `id` attribute, a `prior` child with value 1, a `posterior` child with value 1, and up to 10,000 `foo` children. There are 700,771 `foo` elements in total. We used the three queries in Figure 31. All three produce empty results on the dataset. However, the data items that are used to evaluate the predicates come from different locations in the element.

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

Fig. 30. Toxgene template.

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

Fig. 31. Synthetic queries.

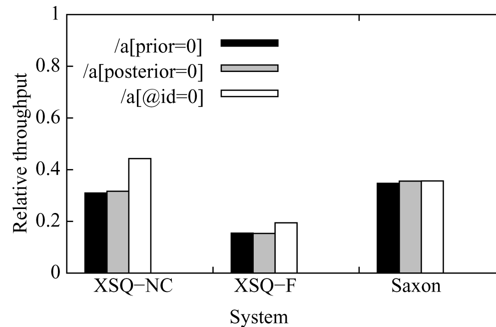


Fig. 32. Effect of data ordering on throughput.

Figure 32 summarizes the results of running XSQ-NC, XSQ-F, and Saxon on these queries. Saxon's throughput is essentially the same for all three queries since it always builds the whole DOM tree before the evaluation. When it traverses the DOM tree to evaluate the query, the document order of the elements is not important. However, the throughput of XSQ-NC is about 30% higher for Q3 than for Q1 and Q2. For Q3, XSQ-NC can determine at the beginning of an `A` element that all the contents in it should be ignored. For Q1 and Q2, on the other hand, the content of every `A` element must be buffered because the `prior` and `posterior` child elements may occur anywhere before the `</A>` tag. We also observe that XSQ-F is not as sensitive as XSQ-NC to the element order. Even if XSQ-F determines that an incoming item is in the result set, XSQ-F cannot output it right away since there may exist undecided queue items. Thus, XSQ-F must first mark the item as "output" and then check the queue, which reduces its sensitivity to the order of the elements.

We also studied the sensitivity of throughput to the result size, which varies across the systems. For example, XQEngine is slower than the other systems in Figure 23 where the query returns a large portion of the dataset. However, if a node test in the query is not in the data, XQEngine returns the empty result set very quickly because it builds an inverted-file index on all the strings in the data. The other systems, lacking such an index, spend similar amount of time on the query irrespective of whether the node tests in the query appear in the data.

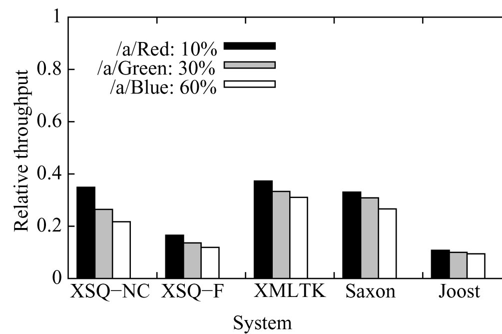


Fig. 33. Effect of the result size on throughput.

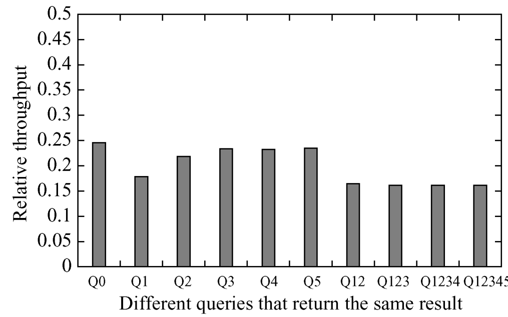
We used Toxgene to generate a 10-MB dataset consisting of a mix of three types of elements (besides a few top level elements): 10% of the elements have name `red`, 30% `green`, and 60% `blue`. The content of each such element is a single character. We used this dataset with three queries: `/a/red`, `/a/green`, and `/a/blue`, generating query results that are roughly 1 MB, 3 MB, and 6 MB in size, respectively. Figure 33 illustrates the relative throughputs of the systems on these queries. (XQEngine is not tested for the same reason as described in the previous experiment.)

We observe that XSQ-NC is quite sensitive to the result size. The different 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 by XSQ-NC. If there are more items in the result set, XSQ-NC will perform more matching record activations 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. XSQ-F always keeps the item first since there may be multiple transitions that process the item. Even if the item is not in the result, only when all the transitions finish can we throw it away. The difference between the treatment of elements in and not in the result is therefore not as large as the difference in XSQ-NC. Saxon's throughput is not very sensitive to the result size because, after it loads all data into main memory, all query evaluation is performed in main memory except for 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 sensitive to the result size.

6.6 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 query length (number of location steps).

In the first experiment, we executed a set of queries that return the same result set but have different number of closure axes. In Figure 34, Q_S, where $S \subseteq \{1, 2, 3, 4, 5\}$, is the query in which the ith location step has a closure axis for



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

Fig. 34. Effect of closure axes in the queries on NASA dataset.

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.) The memory usage of XSQ-F when evaluating these queries is summarized in Figure 36. The HPDT generated for the query `/dataset/reference/source/other/name` is depicted in Figure 35. 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 3.1.

Figure 36 indicates that the memory used for the different queries does not vary much. This insensitivity is due to the fact that the memory used for storing the HPDT and matching records is only a very small amount in the total memory used by the system. The buffers are responsible for most of the memory usage. Therefore, although different number and position of closure axes lead to different number of matching records at runtime, the difference in overall memory usage is very small.

Figure 34 summarizes the throughput on the above queries. We observe that the throughput is lower for queries with a starting closure axis than for queries with a starting child axis. The DTD of the dataset [Borne 2002] suggests that all the top level element are `dataset` elements. (The `datasets` element in the DTD is treated as the document root.) qIf the first location step has a closure axis, after the runtime engine (Figure 34) makes the transition from state $1 (with depth stacks omitted here) to $2, $1 keeps active. Then, the engine needs to check for every incoming element whether it is a `dataset` element, which involves string comparisons. In contrast, if the first location step uses a child axis, $1 does not remain active after the transition. Therefore, only for all the child elements of the `dataset` elements does the engine check their names. Any

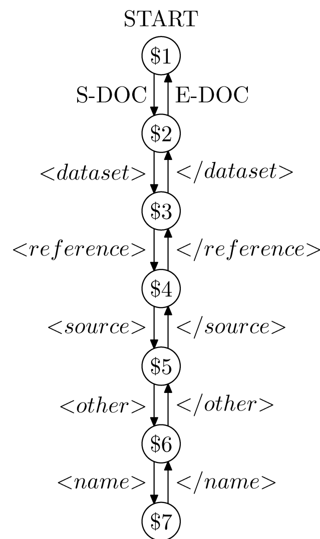


Fig. 35. HPDT generated for query `/dataset/reference/source/other/name`.

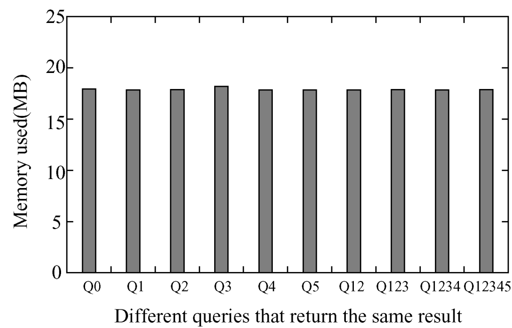


Fig. 36. Memory usage of queries with closure axes on NASA dataset.

element that is not a descendant of both `dataset` and `reference` is ignored after the engine checks its depth, which is much faster than string comparison.

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. Let S be the set of elements that match the $(i-1)$th location step and S' the set of children of nodes in S. We define the selectivity of the ith location step (for a given dataset) to be the fraction of the nodes in S' that match the ith location step. If the ith location step uses the closure axis, we use descendants instead of children in identifying S'. 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 child elements of `dataset` elements other than `reference` (which means the selectivity of the second location step changed from

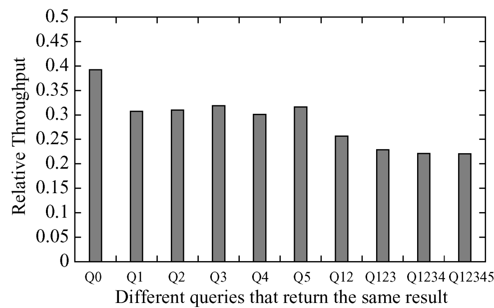
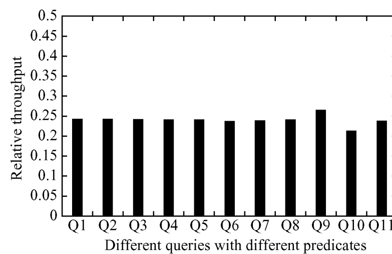


Fig. 37. Experiment of Figure 34 using a modified NASA dataset.



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. 38. Effect of predicates in the queries on NASA dataset.

around 20% to 100%). The result is summarized in Figure 37. We observe 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 queries 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 for Q1 (checking descendants of child elements other than `reference`) on the original dataset no longer exists since the `dataset` elements in the new dataset have only `reference` child elements. In general, when the selectivity of a location step is small, closure axes preceding this step result in a performance penalty because the descendants that are not in the result set 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. In the next experiment, we used queries on the NASA dataset with predicates of different types and in different positions in the query. The results are summarized in Figure 38. We abbreviate the node test `dataset` as

d in the queries. Similarly, we abbreviate other node tests 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 child elements, the first altname child element usually has the attribute type that has value *ADC*. Therefore, the queries Q3 and Q4 both check the first altname child element 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 child elements 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.

7. 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 solution used by XSQ. All the methods described in this article 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.

In continuing work, we are extending XSQ to support the simultaneous evaluation of multiple queries on a stream. The main idea is to use simple state machines to evaluate common *segments* among a group of queries and route the partial results among the smaller machines to construct the result of each query. Each segment consists of a pair of adjacent node tests (in a query) and the axis between them. For example, both A/B and A[B] contain segment A/B. A segment s may serve different *roles* in different queries. For instance, A/B can be used in the main trunk of the query, or B can be used in the predicate of A in the form of A[B]. For every (s, r) combination, where r is a role such as main-trunk

or predicate, the system records in a segment table a list of queries to indicate that the segment s is used in these queries as the role r. Given a large number of XPath queries, it is likely that a segment is used in many queries as the same role. The above operations can be performed for a large group of queries simultaneously. In our implementation, most of the bookkeeping information is stored using bitmaps, allowing efficient operations. Further details of this method appear in a technical report [Peng and Chawathe 2004]. We are also extending XSQ to take advantage of structural information (such as DTDs or XML Schemas) when it is available. Finally, we are investigating methods for reducing the overhead of parsing XML by using a framework in which the data source augments streams with *hints* that permit a consumer to skip uninteresting portions of the stream.

ACKNOWLEDGMENTS

We would like to thank Jérôme Siméon and Mary Ferández 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. We would also like to thank the anonymous referees for several suggestions that helped improve the article.

REFERENCES

ABITEBOUL, S., QUASS, D., MCHUGH, J., WIDOM, J., AND WIENER, J. 1996. The Lorel query language for semistructured data. *J. Dig. Lib. 1*, 1 (Nov.), 68–88.

ALTINEL, M. AND FRANKLIN, M. J. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)* (Cairo, Egypt), 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/.

BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND WIDOM, J. 2002. Models and issues in data stream systems. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)* (Madison,Wisc.), ACM, New York, 1–16.

BARBOSA, D., MENDELZON, A., KEENLEYSIDE, J., AND LYONS, K. 2002. ToXgene: A template-based data generator for XML. In *Proceedings of the 5th International Workshop on the Web and Databases* (Madison, Wisc.), 49–54.

BARTON, C., CHARLES, P., FONTOURA, M., GOYAL, D., JOSIFOVSKI, V., AND RAGHAVACHARI, M. 2002. An algorithm for streaming XPath processing with forward and backward axes. In *Proceedings of the PLAN-X Workshop on Programming Language Technologies for XML* (Pittsburgh, Pa).

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 *Proceedings of the International Conference on Data Engineering (ICDE)* (Bangalore, India). 455–466.

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.

BENEDIKT, M., FAN, W., AND KUPER, G. 2003. Structural properties of XPath fragments. In *Proceedings of the International Conference on Database Theory (ICDT)* (Siena, Italy).

BOAG, S., CHAMBERLIN, D., FERNÁNDEZ, M. F., FLORESCU, D., ROBIE, J., AND SIMÉON, J. 2003. XQuery 1.0: An XML query language 1.0. W3C Working Draft, W3C, http://www.w3.org/TR/xquery/. August.

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

BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C., AND MALER, E. 2000. Extensible markup language (XML) 1.0 (2nd Edition). World Wide Web Consortium Recommendation. 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 (SIGMOD)* (Montréal, Québec, Ont. Canada). ACM, New York, 505–516.

CHAN, C. Y., FELBER, P., GAROFALAKIS, M. N., AND RASTOGI, R. 2002. Efficient filtering of XML documents with XPath expressions. In *Proceedings of the International Conference on Data Engineering (ICDE)* (San Jose, Calif.). 235–244.

CHANDRA, A. K., KOZEN, D. C., AND STOCKMEYER, L. J. 1981. Alternation. *J. ACM 28*, 1, 114–133.

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 *Proceedings of the International Conference on Data Engineering (ICDE)* (Heidelberg, Germany). 595–604.

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

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

DIAO, Y., FISCHER, P., AND FRANKLIN, M. J. 2002. YFilter: Efficient and scalable filtering of XML documents. In *Proceedings of the International Conference on Data Engineering (ICDE)* (San Jose, Calif.). 341–344.

FERNÁNDEZ, 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)* (Tucson, Az.), J. Peckham, Ed. 549–552.

FERNÁNDEZ, M. F. AND SIMÉON, J. 2002. Galax. http://db.bell-labs.com/galax/.

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).

GOTTLOB, G., KOCH, C., AND PICHLER, R. 2003. The complexity of XPath query evaluation. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)* (San Diego, Calif.). 179–190.

GREEN, T. J., MIKLAU, G., ONIZUKA, M., AND SUCIU, D. 2003. Processing XML streams with deterministic automata. In *Proceedings of the International Conference on Database Theory (ICDT)* (Siena, Italy). 173–189.

GUPTA, A. K. AND SUCIU, D. 2003. Streaming processing of XPath queries with predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (San Diego, Calif.). ACM, New York, 419–430.

HOFFMANN, C. M. AND O'DONNELL, M. J. 1982. Pattern matching in trees. *J. ACM 29*, 1, 68–95.

HORS, A. L., HGARET, P. L., WOOD, L., NICOL, G., ROBIE, J., CHAMPION, M., AND BYRNE, S. 2000. Document object model level 2 core specification. W3C Recommendation, W3C, http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113. November.

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

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

KAY, M. 2003. XSL transformations (XSLT) version 2.0. W3C Working Draft, W3C, http://www.w3.org/TR/xslt20/. November.

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

LAKSHMANAN, L. V. AND SAILAJA, P. 2002. On efficient matching of streaming XML documents and queries. In *Proceedings of the 8th International Conference on Extending Database Technology* (Prague, Czech Republic). 142–160.

LEY, M. 2003. Computer science bibliography. `http://dblp.uni-trier.de/xml/`.

LUDASCHER, B., MUKHOPADHAYN, P., AND PAPAKONSTANTINOU, Y. 2002. A transducer-based XML query processor. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)* (Hong Kong, China). 227–238.

MIKLAU, G. AND SUCIU, D. 2002. Containment and equivalence for an XPath fragment. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)* (Madison,Wisc.). ACM, New York, 65–76.

OLTEANU, D., KIESLING, T., AND BRY, F. 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, Germany, May.

PENG, F. AND CHAWATHE, S. S. 2003. XPath queries on streaming data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (San Diego, Calif.). ACM, New York, 431–442.

PENG, F. AND CHAWATHE, S. S. 2004. XPaSS: A multi-query streaming XPath engine. Tech. Rep. CS-TR-4565 (UMIACS-TR-2004-10), Department of Computer Science, University of Maryland. May.

SAX PROJECT ORGANIZATION. 2001. SAX: Simple API for XML. `http://www.saxproject.org/`.

SEGOUFIN, L. AND VIANU, V. 2002. Validating streaming XML documents. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)* (Madison, Wisc.). ACM, New York, 53–64.

TUCKER, P. A., MAIER, D., AND SHEARD, T. 2003. Applying punctuation schemes to queries over continuous data streams. *Bull. Tech. Comm. Data Eng. 26*, 1 (Mar.), 33–40.

WU, C. H., HUANG, H., ARMINSKI, L., CATRO-ALVEAR, J., CHEN, Y., HU, Z. Z., LEDLEY, R. S., LEWIS, K. C., MEWES, H. W., ORCUTT, B. C., SUZEK, B. E., TSUGITA, A., VINAYAKA, C. R., YEH, L. S., ZHANG, J., AND BARKER, W. C. 2002. The protein information resource: An integrated public resource of functional annotation of protein. *Nuc. Acids Res. 30*, 35–37.

Received August 2004; revised December 2004; accepted January 2005