Streaming XPath Subquery Evaluation

Feng Peng
Department of Computer Science
University of Maryland, College Park
Maryland, USA, 20742
pengfeng@cs.umd.edu

Sudarshan S. Chawathe
Department of Computer Science
University of Maryland, College Park
Maryland, USA, 20742
chaw@cs.umd.edu

Abstract

We describe a method for the streaming evaluation of XPath queries that have subqueries in predicates. Our method rewrites XPath queries into a set of predicate-free labeled linear-form expressions (LFEs). These LFEs are used to generate a pushdown transducer that enables efficient management of a buffer and hierarchical index at runtime. To the best of our knowledge, our method is the first to support XPath subqueries in a streaming environment. Our method also provides optimal buffering, minimum-latency output, and optimal predicate evaluation. The method has been fully implemented and publicly released in the XSQ system. We present an experimental study of XSQ and related systems on both real-life and synthetic datasets, and investigate how subqueries and other features affect the performance of these systems.

1 Introduction

XML is now widely used as a standard format of information exchange in distributed computing environments such as Web services, data integration, and information dissemination. In many cases, data is transmitted among applications in streaming form. In *streaming XML query evaluation*, serialized XML is parsed by a SAX parser to generate a sequence of SAX events. The query engine processes the SAX events and emits the results as the data is streaming in. Streaming query evaluation usually requires less memory and provides higher throughput than the traditional approach, which evaluates the query on materialized data in main memory. However, streaming evaluation is also more difficult due to the restriction that every data item can be seen only once. Since seeking back in the streams is usually not feasible, we have to explicitly buffer data items that may be used in the future.

XPath is a succinct yet powerful path language used to address parts of an XML document. For example, the XPath expression `//book[price=10 and quantity>1]//author` selects the `authors` of all the `books` whose `price` is 10 and `quantity` is larger than 1. The *predicate*, such as `[price=10 and quantity>1]`, specifies the conditions that an element has to satisfy to be selected by the path expression. Each clause in the predicate is essentially a *subquery* that is evaluated within the context specified by the original query, e.g., the `price` element matched in the subquery `price=10` has to be a child of a `book` element that is selected by the original query. A subquery may be a complex XPath query itself. Unlike the main query, a subquery always returns a boolean result: true if the result set is not empty and false otherwise.

Our goal of this paper is to support subqueries in streaming XPath evaluation. Moreover, our method provides three very important guarantees for streaming XPath processors: *optimal buffering, minimum-latency output*, and *optimal predicate evaluation*. **Optimal buffering** requires the XPath processor only buffers the data items whose membership in the result set cannot be determined based on currently available data, and therefore must be buffered by all streaming systems. **Minimum-latency output** means the available result items are returned to the user as early as possible, i.e., at the exact moment when its membership in the result item can be determined by any streaming system. **Optimal predicate evaluation** means that only we only evaluate those undetermined subqueries whose evaluation may change the result of the predicate, which we call *necessary subqueries*. For example, if one of two subqueries that are connected by an OR operator has been evaluated to true, the other subquery becomes unnecessary.

It is a challenging task to evaluate XPath queries with predicates and closure axes (such as //, which

denotes the descendant-or-self relation between two elements) over streaming XML data. First, we may need to buffer the potential result items (called **candidates**) since the order in which the data arrives may not permit the query engine to immediately determine whether a data item belongs to the query result. We also have to keep track of the partial predicate results of every candidate since each of them may be pending on different predicates. Moreover, the combination of the closure axes and the predicates may result in *multiple matchings* between an element and the query, each of which may lead to different predicate results. Only when all the matchings fail the query can we decide the element is not in the result.

Providing the three guarantees poses extra challenges for streaming evaluation. Without the three guarantees, we can buffer arbitrary amount of data or determine at any time whether an element is in the result. Given enough memory or relax the latency requirement, such an approach may be suitable for bounded streams. However, even given enough memory, this approach may lead to smaller throughput for larger datasets (see Figure 24 and Figure 26). Moreover, it also leads to larger latency than streaming evaluation in general. In contrast, to satisfy these guarantees, we have to maintain the state for every candidate and always take action at the earliest moment a decision for a candidate can be made. Moreover, it is inefficient to record the partial result of every candidate separately, otherwise we have to update the candidates separately when a predicate is evaluated. With the presence of subqueries, such an approach is even more difficult since it is not straightforward to locate the candidates that are affected by the new evaluation result.

1.1 Motivating examples

The following examples highlight some of the difficulties of supporting subqueries in streaming evaluation. To denote the partial results of a predicate of an element, we use the term **state of an element** to denote which subqueries in its predicate are necessary, and therefore also implies which subqueries have been evaluated. Note that since subqueries may themselves be partially evaluated, the exact definition of the state is recursive (and will be given in Section 4).

Example 1 [Multiple Matchings] Consider evaluating the following XPath query (separated into four lines) on the XML stream in Figure 1. (For now, we can think of the query engine as reading the document line by line. Later, we will introduce the SAX model.)

```
1. <store>
2.   <name>Amazon</name>
3.   <book>
4.     <price type="sale">15</price>
5.     <title>XML</title>
6.   </book>
7.   <book>
8.     <title>Java</title>
9.     <author>John</author>
10.    <price>10</price>
11.    <related>
12.      <store>
13.        <name>BN</name>
14.        <book>
15.          <quantity>1</quantity>
16.          <author>Mike</author>
17.          <author>John</author>
18.          <title>JDBC</title>
19.          <price>15</price>
20.        </book>
21.      </store>
22.    </related>
23.    <price type='sale'>8</price>
24.    <quantity>2</quantity>
25.  </book>
26.</store>
```

Figure 1: Example XML Data

```
//store[//name="BN"]
  //book[not(author!="John")
          and (//quantity=1 or //price=10)]
   //title
```

The above query asks for the `title` descendants of the `book`s that do not have an `author` child that is not "John" *and* either have a `quantity` descendant with value 1 or a `price` descendant with value 10. The `book` element should be the descendant of a `store` element who has a `name` descendant with value "BN".

The `title` element in line 18 has three matchings with the query, as shown in Figure 2. In each matching, for each matched `store` element, there are 3 possible states. For each matched `book` element, there are 3^3 possible states since the predicate consists of three subqueries and each has three possible states. Therefore, we have 3^4 possible state combinations for every matching of this `title` element. Some of the combinations cannot be used in the evaluation since

store		book				title
line	[//name="BN"]	line	[not(author!="John")]	[//quantity=1]	[//price=10]	line
1	TRUE	7	NA	NA	TRUE	18
1	TRUE	14	FALSE	TRUE	NA	18
12	TRUE	14	FALSE	TRUE	NA	18

Figure 2: Combination of predicate results

the `store` element has to be the ancestor of the `book` element. Figure 2 lists the current state combinations for the three matchings at the time we encounter this `title` element in line 18.

For every possible combination of states, we have to define proper operations to satisfy the three guarantees. For example, if we want to guarantee optimal predicate evaluation, then: When we encounter the `quantity` element in line 15 we have to evaluate the predicate `quantity=1` since this `quantity` is the descendant of the `book` element in line 14, of which the three subqueries are all pending. Although this `quantity` element is also the descendant of the `book` element in line 7 we do not evaluate this predicate for that `book` since it is not necessary. We have seen a `price` with value 10 and these two subqueries are connected by an OR operator. (Therefore, the result is shown as NA in the first line in Figure 2 although it is TRUE at the time.)

We have to define all these operations for all the state combinations, since we cannot predict the future input. A small difference in the input may require a different set of operations for the same query. For example, if we exchange line 15 and line 16, we no longer need to evaluate subquery `quantity=1` when we encounter this `quantity` element. Although it is a descendant of two `book` elements, the subquery `quantity=1` is needed neither for the first `book` in line 7 (since we have seen a `price` with value 10), nor the second `book` in line 14 (since we have seen an `author` that is not "John"). The key challenge here is to define all these operations in a dynamic and systematic manner so that no matter in which combination of states, we can decide the proper operation. ∎

We also have to know the correct scope of a element e that evaluates a subquery to true or false. The **scope** of e is the set of elements to which the evaluation result will be applied. We also say that these elements in the scope are **affected by** e. If we limit the predicate to use a single descendant, the scope of a matched element is either its ancestors or its descendants. With the presence of subqueries, it is not easy to determine the scopes of an element, especially during runtime. Following example illustrates some of the difficulties.

Example 2 [Scopes of Elements] Consider the query used in Example 1. When we encounter the `name` element in line 13, we have to know that the predicate `[//name="BN"]` of both `store` elements in line 1 and 12 evaluates to true. Accordingly, we have to know the scopes of these two `store` elements. At this time, there are two items in the buffer: the `title` elements in line 5 and 8. These two elements are both in the scope of the first `store` element. However, only the first `title` should be sent to output at the time since the `book` element (in line 3) in the matching has satisfied the predicate. The second `title` will remain in the buffer since the `book` element (in line 7) in the matching still has the first part of the query `[not(author!="John")]` pending. ∎

Not only is it inefficient to maintain the states separately for every candidate, it is also difficult to do so in the presence of subqueries. Since the data may be nested, a subquery may be matched by an element e and one of its descendants e'. In this case, we have to keep track of the states for both e and e' when we evaluate the query. If either e or e' evaluates the subquery to true, we know that the result for the other element will no longer affect the result of the predicate. The following example illustrates the complexities.

Example 3 [Multiple Matchings in the Subquery] We modify the previous example query to use the `book` element in the predicate of the `store` element:

```
//store[//name="BN" and
       //book[not(author!="John")
            and (//quantity=1 or //price=10)]]
  //title
```

For the `store` element in line 1, it has two `book` descendants that matches the second subquery in its predicate. When we encounter the `book` element in line 14, another `book` element in line 7 also matches the second subquery and has not been determined yet. To denote this fact, we cannot simply record that

this `store` element may satisfy its subquery. We have to record the fact that it has two `book` descendants that may satisfy this subquery. Otherwise when one of them fails the subquery, we will erroneously determine that the `store` element fails the predicate. Essentially, even for a single element, its partial predicate results have to be organized as a tree to incorporate all the open and undecided elements matched by the subqueries. ∎

To address all the difficulties illustrated above, a novel approach is proposed in this paper using a *finite state transducer* augmented by a queue with a hierarchical index. The transducer is transformed from an XPath query after we decompose it into a set of simple queries free of predicates. The operations in the transducer carry out the semantics of the original query and provide the three guarantees for any incoming data.

The main **contributions** of this paper may be summarized as follows:

- We outline the challenges of streaming XPath subquery evaluation in a streaming environment. We characterize pure streaming evaluation of XPath queries using three properties: (1) Optimal buffering: At each point in time, every item in the buffer is necessary (for any method of query evaluation). (2) Minimum-latency output: As soon as it is logically possible to determine that an item belongs to the query result, it must be emitted as output. (3) Optimal predicate evaluation: No query engine can evaluate the query by skipping one or more predicate evaluations performed by this method.

- To the best of our knowledge, our method is the first pure streaming method to support XPath subqueries.

- The methods described in this paper are fully implemented in the XSQ system, which is freely available at `http://www.cs.umd.edu/projects/xsq` under the GNU GPL license.

- We provide a detailed experimental study of our method and those in several publicly available XPath processors. The study highlights the effects of not only system designs but also features of XPath queries (e.g., multiple predicates, subqueries) and XML datasets (e.g., depth, recursive elements).

Outline of the rest of the paper: In Section 2, we cover the basics of XPath, SAX, and XSQ. Section 3 outlines the system architecture that uses the pushdown transducers to evaluate the XPath queries on XML streams. In Section 4, we describe how to decompose an XPath query with subqueries into linear-form expressions. Section 5 presents our method for organize buffer items and partial predicate results. Section 6 describes how to map these linear-form expressions to a transducer that is used at runtime. Related work is discussed in Section 7. Section 8 presents the results of our experimental study of our method and several related systems. We conclude in Section 9.

2 Preliminaries

In this section, we first present a brief overview of the XPath query language, focusing on subqueries. Next, we outline the features of the SAX model used by our method. Finally, we briefly describe the XSQ streaming XML query system that is used by our implementation.

2.1 XPath with subqueries

An XPath expression consists of a sequence of **location steps**, which is called the **location path**, and an optional **output function**, which specifies the contents or functions of the selected elements that form the result. The ith location step is in the form of $A_iN_i[P_i]$. In this form, A_i denotes the **axis**, which is either the *child* axis (`/` for short) or *descendant-or-self* axis (`//` for short) in this paper. The N_i component denotes the **node test**, which is the name matching elements are required to have. The optional P_i component denotes a **predicate** that matched elements are required to satisfy.

In general, an XPath expression is of the form of $A_1N_1[P_1]\ \ldots\ A_kN_k[P_k]O$. We call the expression $A_1N_1 \ldots A_kN_k$ (obtained by ignoring predicates) the **main trunk** of the query. It specifies the pattern that result elements are required to match. A **matching** between an element e_k and the query determines a sequence of elements $e_1, \ldots, e_k$ such that (1) e_i matches the node test N_i of the ith location step and (2) e_{i-1} is the parent of e_i if A_i is `/` and the ancestor of e_i if A_i is `//` (using e_0 to denote the document root).

Each optional predicate P_i consists of either simple tests on the elements matching the location step to which it is attached (e.g., `[@type="sale"`, which tests the `type` attribute of matching elements) or a complex expression that may include boolean combinations of subqueries. There are two kinds of sub-

queries: **Absolute subqueries** begin with a `/` and are standalone XPath queries and are evaluated using the document root as the context. **Relative subqueries** begin with either no explicit axis or the `//` axis. A relative subquery is evaluated in the context of the elements that match the node-test of the location step to which it is attached. For example, the subquery `book/author` in the XPath query `//related/store[book/author]` is evaluated starting at one of the `store` elements that match match `//related/store`. (For the data of Figure 1, the subquery is evaluated starting at the `store` element in line 12, and matches the `author` elements in lines 16 and 17. Absolute and relative subqueries in XPath are analogous to uncorrelated and correlated subqueries, respectively, in SQL. Each absolute subquery is evaluated as a standalone XPath query. If its result is nonempty, the subquery is conceptually replaced with the `true()` predicate; otherwise, it is replaced with the `false()` predicate. Thus, such subqueries do not pose significant implementation challenges and we henceforth focus on relative subqueries.

Recall that, as illustrated by Example 1, the matching between an element and a query is not unique. Such multiple matchings also apply to predicates and subqueries. XPath semantics specify that an element belongs to the result if there is some matching (of elements to components in the main trunk as well as in the predicates) that evaluates to true.

The item in the query result generated for each matched element is specified by the output function O. For example, `/text()` denotes that the text content of each matched element forms the query result. Other choices for the output function include `/@attr`, denoting the value of the named attribute of each matched element, and functions such as `count()`, with the usual semantics. The output function may also be a boolean operation. For example, the query `//name="BN"` returns true if there is a `name` element with text content `"BN"`. If no explicit output function specified, the query result consists of each matched element, including its subelements (recursively).

Output functions in the subqueries have semantics different from those of output functions in the main query. For example, in query `//store[//book[author]/@id]`, the output function `@id` does not specify that the value of the `id` attribute is to be in the query result. Instead, it tests for the existence of the `id` attribute. Henceforth, we refer to output functions in predicates as **test functions**.

An XPath predicate may contain multiple subqueries connected using boolean operators, with the usual semantics. However, negation is supported as a function instead of as an operator; thus, we write `[not(P)]` instead of `[not P]`. Further, predicates `[P!=5]` and `[not(P=5)]` have different semantics. The former evaluates to true if and only if there is at least one item in the result set of `P` that does not equal 5, while the later evaluates to true if and only if all items in the result set of `P` are unequal to 5.

type	depth	name	attributes
begin	1	store	
begin	2	name	
text	2	name	(TEXT, "Amazon")
end	2	name	
begin	2	book	
begin	3	price	(type, "sale")
text	3	price	(TEXT, "15")
end	3	price	
begin	3	title	
text	3	title	(TEXT, "XML")
...	...	...	...

Figure 3: Sequence of SAX events

2.2 SAX

Streaming XML data is usually modeled using the SAX (Simple API of XML) model [20]. For each opening and closing tag of an element, the SAX parser generates, respectively, a **begin** and **end** event. The begin event of an element comes with an **attribute list** that encodes the names and values of attributes associated with the element. The text contents enclosed by opening and closing tags result in the SAX parser generating a **text** event. The sequence of the SAX events corresponds to a preorder traversal of the tree representation of the XML data, with attribute nodes combined with their parents.

Each SAX event is a 4-tuple consisting of **event type**, `event depth`, `element name`, and **attribute (name,value) pairs**. The first ten SAX events generated for the data of Figure 1 are depicted in Figure 3. We define the **depth of an event** to be the depth (in the XML tree) of the element that triggers the event. Standard SAX events do not include depth information. However, it is easily computed by maintaining a single counter in the SAX event handler code. For ease of presentation, we will henceforth assume that this depth information is part of the SAX event itself.

2.3 XSQ

XSQ [21], supports streaming evaluation of XPath queries that may include closures and multiple predicates, but predicates are restricted to using only one descendant or attribute. The method is based on mapping each location step of a query to a finite-state automaton augmented with a buffer (BPDT). Intuitively, the BPDT for a location step stores elements matching the location step's axis and node-test in its buffer. If the elements satisfy the location step's predicate, the BPDT enters a TRUE state; otherwise, it enters a *na* state. A copy of the BPDT for the k'th location step, $k > 1$, of a query is attached to each of the TRUE and NA states of the BPDT for the $k-1$'th location step. In the resulting hierarchically structured automaton (HPDT), each state's identifier encodes the predicates in the query that have been satisfied by items in the corresponding buffer and the predicates that remain to be satisfied. As elements satisfy additional predicates, they are propagated from a BPDT's buffer to the buffer of an ancestor BPDT with an unsatisfied predicate (or marked for output if all predicates are satisfied).

The above method cannot be easily extended to support subqueries. The mapping of location steps to BPDTs in XSQ is based on templates that use the fact that there are only a few different combinations of axis, node-test, and predicate. When predicates include subqueries, there is a much greater variety of location steps and this template-based approach is not practical. Further, an HPDT encodes the set of predicates satisfied by an element in a BPDT buffer using that BPDT's position in the hierarchy. Again, this approach is not practical when predicates contain subqueries. The naive generalization would be to include recursive HPDTs within BPDTs. However, the interconnection of such automata and the proper management of buffer items would be very complex.

3 Outline of Subquery Evaluation

In this section, we present a high-level description of our solution and introduce the main subtasks, which are described in the following sections.

3.1 Query evaluation by flag-marking

We use a finite state transducer (FST) augmented with a queue to evaluate XPath queries over XML streams. The transducer receives the SAX events generated from the incoming stream as the input.

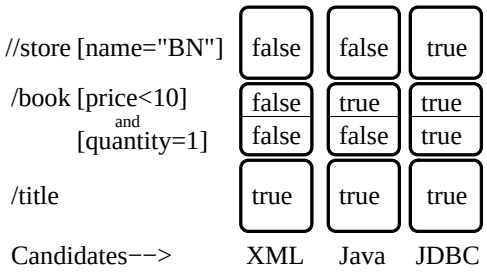


Figure 4: Marking flags

When the transducer recognizes an element that matches the pattern specified by the **main trunk** of the query (with the predicates are trimmed off) with some predicates pending, it first enqueues this element as a **candidate**. Each candidate in the queue is associated with n flags where n is the number of location steps in the query. The ith flag denotes the state of the predicate in the ith location step. Only when all the n flags are true do we conclude that the candidate is in the result.

This transducer, which we call a **QT**, is constructed from an XPath query. The operations in a QT include enqueuing a candidate, marking a flag of a candidate with new (partial) predicate results, emitting a candidate if all its flags are true, and so on. All these operations are assigned to the QT transitions during the construction of the QT. They convey the semantics of the original XPath query. The following example illustrates the basic idea of query evaluation by flag-marking.

Example 4 [Query evaluation by marking flags] Consider evaluating the following query over the data depicted in Figure 1:
`//store[name="BN"]/book[price<10 and quantity=1]/title`.
There are three `title` elements that match the main trunk of the query (`//store/book/title`). We associate three flags for each of them, as illustrated in Figure 4. Since the third location step does not have a predicate, the third flag is always true. Also, since the second predicate has two subqueries, we split the second flag of every element into two parts and denote the result of each part separately. Figure 4 illustrates the final results of all the flags (and parts). We can see that an element is in the result if and only if all the flags (not parts) are marked true.

Suppose we are evaluating this query in a streaming environment. For every candidate, all the flags that correspond to a non-null predicate are initially

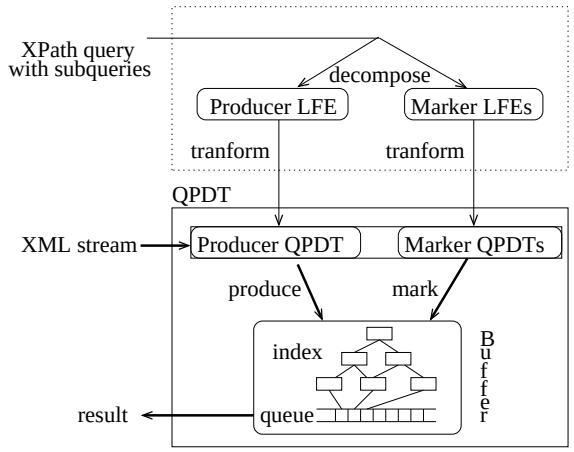


Figure 5: System Architecture

marked NA, which stands for pending results. These flags are updated dynamically during the evaluation. For example, when we encounter a `price` child of the `book` element with value less than 10, the first subquery of the second predicate evaluates to true. We then mark the first part of the second flag as TRUE for the `title` candidates that are the children of this `book` element, which may have been put in the queue or will be encounter in the stream later. If we encounter a `quantity` child of this `book` with value 1 later in the stream, we mark the second part of the second flag of those `title` descendants of this `book`. Those `title` candidates should be sent to output right away if the first flag of them (which corresponds to the first predicate in the query) has already been marked TRUE.

■

Given the basic idea, there are several tasks needed to be solved. The first is to transform an XPath query into a transducer that responses to the input XML stream and emits the result of the query. Second, we want the transducer to buffer the least amount of data and emit the result as early as possible to provide the three guarantees we proposed in Section 1. Third, given the transformed transducer, we also need to implement the marking operations efficiently.

3.2 System Architecture

The architecture for our method, which is implemented in the XSQ query engine, is illustrated in Figure 5. The actions that occur at query compilation time are enclosed in the dotted boundary box and those that occur at run time (stream processing time) are enclosed in the solid boundary box. The bold lines and arrows indicate the data flow at run time.

We first decompose an XPath query with subqueries into a set of **linear-form expressions (LFEs)** that convey the semantics of the query. An LFE is an XPath expression free of predicates, which is easy to evaluate over streaming data since it is essentially a regular expression and can therefore be recognized by an FSA. A unique **producer LFE** is generated from the main trunk of the query to specify the pattern that the candidates should match. In Example 4, the producer LFE is `//store/book/title`. A set of **marker LFEs** are generated to specify the conditions that the candidates should satisfy. Each marker LFE represents a part of a predicate. For example, a marker LFE for the query in Example 4 is `//store/book/quantity=1`. When this LFE is matched in the stream, we will mark the second flags of the `title` descendants of the `book` element matched by this LFE. How to locate those `title` elements that we should mark, however, is not a trivial problem and will be addressed later in Section 6.

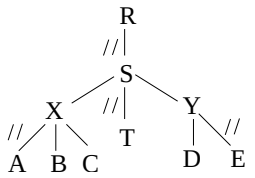
We use a **global queue** to store all the candidates to ensure the document order is preserved in the result. We also need to distinguish the candidates based on the predicates that they have satisfied. For this purpose, we use a **hierarchical index** that encodes the set of satisfied predicates for each candidate in the queue.

A **producer QT** is constructed from the producer LFE. It is used to create the index nodes and the candidates when it encounters elements that match the main trunk. A set of **marker QTs** are constructed from the marker LFEs. Each marker QT is responsible for evaluating a predicate or a part of a predicate. It marks an index node with the evaluation result. For each incoming event, this apparatus enables efficient determination of several facts, such as (1) whether any new data needs to be buffered; (2) which, if any, pending predicates are satisfied, and the candidates to which they apply; and (3) which, if any, candidates may be sent to output, and which may be purged from the buffer.

4 XPath Decomposition

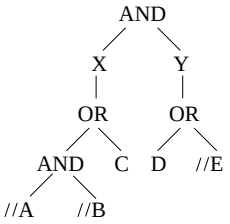
4.1 Flatten the XPath query

We first flatten an XPath query into a set of queries free of predicates. During this process, we need to maintain the semantics of the query among the



//R//S[X[(//A and //B) or C] and Y[D or //E]]//T

Figure 6: A Pattern Tree



[X[(//A and //B) or C] and Y[D or //E]]

Figure 7: The Syntax Tree for a Predicate

separate queries so that we can obtain the correct query result through the evaluation of the decomposed queries.

The flattening process builds the **pattern tree** of the XPath query and creates a linear-form expression (LFE) for every path in the pattern tree from the root node to a leaf node. Each node test `X` in the query corresponds to a node labeled with `X` in the pattern tree. For each segment `X/a::Y` in the query, we set node `X` as the parent of node `Y` and the edge between them is labeled with `a`. Child axis is the default axis and not labeled in the tree. The transformation from a path in the pattern tree to an LFE is straightforward. From the root node and using an empty string, we append the label of the next axis and the label of the next node to the string, whose value is the LFE after the leaf node is visited.

A single **producer LFE** is created from the path from the root to the leaf node that is in the main trunk, which will match the candidates. The LFEs that are created from other paths in the tree are called **marker LFEs**. If a marker LFE is matched in the stream, we know that a part of a predicate is satisfied and therefore we mark the candidates that are affected by this new result. We describe how to determine which candidates to mark and how to mark them in Section 4.2. Figure 6 displays a pattern tree of an XPath query and the LFEs are listed as follows:

Producer LFE: `//R//S//T`
Marker LFEs: `//R//S/X//A`
`//R//S/X//B`
`//R//S/X/C`
`//R//S/Y/D`
`//R//S/Y//E`

4.2 Two-flag marking scheme

We use a **two-flag marking scheme** in our method. For each predicate of a candidate, we associate two flags with it: a TRUE flag that is a bitmap that records which clauses in the disjunctive normal form (DNF) of the predicate have been evaluated true, and a FALSE flag which is a bitmap that records which clauses in the conjunctive normal form (CNF) of the predicate have been evaluated false. Each LFE is associated with a TRUE map that indicates which clauses it appears in the DNF and a FALSE map that indicates which clauses it appears in the CNF. When an LFE that does not use `not()` function is matched, we mark its TRUE map in the TRUE flag to denote that those clauses are satisfied. When an LFE with a `not()` function is matched or an LFE evaluates to false at the end of an element, we mark the TRUE map of the FALSE flag to denote that those clauses are failed.

Note that currently each candidate can be deemed as being associated with n pairs of flags, each of which corresponds to a location step. For location steps without predicate, the TRUE flag is always all set and the FALSE flag is always all zeros. In Section 5 we organize these flags such that common flags are stored together.

Why do we need two flags? Consider a predicate with two simple subqueries in the query `//X[A and B]`. We associate a two-bit flag with every `X` element, which initially is set to all zeros. When the LFE `//X/A` (`//X/B`) is matched, we set the first (second) bit of the predicate. It is easy to see that the predicate of the `X` element evaluates to true if and only if both bits of its TRUE flag are set.

However, using only TRUE flags is not enough when there are `not()` functions in the predicate. Consider the query `//X[A and not(B)]`. When the LFE `//X/B` is matched in the stream, we should mark the flag of the `X` element to denote that it is not in the result. Since the initial value of the flag is all zeros (which denotes all the subqueries are pending), we cannot denote such fact using only one flag.

We can associate a three-valued flag for each LFE

for a candidate. However, a drawback for such an approach is that the relations among the LFEs are lost. For example, for query `//X[A or B]`, we cannot determine whether the LFE `//X/A` and `//A/B` needs to be evaluated based on their own flags. We compute the maps using the CNF and DNF of the predicate so that we can determine from the flags that which LFEs need to be evaluated.

Marking operations When a subquery in the ith predicate is evaluated, i.e., an LFE is matched in the stream, we mark the ith pair of flags of the candidates to reflect the state change. The marking operations are performed using following rules (AND, OR, and XOR used in the rules are bitwise operators):

- **Marking rule:** When a subquery P_j evaluates to TRUE, we perform the following bitmap operations (tf stands for the TRUE flag, tm_j stands for the TRUE map of P_j, and etc.):
  tf $\leftarrow$ (tf OR tm_j)
  If P_j evaluates to FALSE:
  ff $\leftarrow$ (ff OR fm_j)
  (Note: we will show which candidates to mark in Section 5.)
- **Checking rule:** When we want to evaluate subquery P_j, we first perform the following two bitmap operations:
  (tf AND tm_j) XOR tm_j
  (ff AND fm_j) XOR fm_j
  If *either* result is all zeros, we do not evaluate this subquery, otherwise evaluate it.
- **Assessing rule:** A predicate evaluates to TRUE if and only if all bits in the TRUE flag are set to 1. It evaluates to FALSE if and only if all bits in the FALSE flag are set to 1.

The marking rule ensures that if a subquery evaluates to true (or false), the corresponding clauses in the DNF (or CNF) are marked accordingly. The checking rule ensures that if all the clauses in the DNF (or CNF) that a subquery appears in have been true (or false), we do not to evaluate this subquery. The following example illustrates these ideas. Note that this decomposition scheme of the marking tasks can only work when each subquery in the predicate has a single location step. When a subquery in the predicate itself contains subqueries, we need to further decompose the marking tasks among those nested subqueries, which will be described next.

Example 5 [CNF and DNF] Consider a predicate `[((A and B) or C) and ( D or E )]`:
Its CNF is:
`(A and B and D ) or (A and B and E) or (C and D) or (C and E)`
and its DNF is:
`(A or C ) and (B or C) and ( D or E )`
Therefore, the maps for each of the subqueries in the predicate are:

	true map	false map
A	100	1100
B	010	1100
C	110	0011
D	001	1010
E	001	0101

If we want to evaluate subquery `C`, we first check the current flags. If the TRUE flag's first two bits are 1's, the only possibility is that both `A` and `B` are true, and therefore `C` does not need to be evaluated. If the FALSE flag's last two bits are 1's, the only possibility is that both `D` and `E` are false, therefore `C` does not need to be evaluated either. ■

Bottom-up decomposition If the subqueries in a predicate may be complex queries themselves, we need to further decompose the marking tasks to those nested subqueries.

When we compute the maps, we first remove all the non-leaf nodes in the syntax tree that are not boolean operators. The DNF and CNF are computed for the expression generated from the reduced syntax tree. For each leaf node, its map is determined based on which clauses it appears in the DNF and the CNF. Then the maps are aggregated bottom-up in the original syntax tree to compute the maps for the non-leaf nodes. For example, for the predicate `[//X[//A and //B] or //Y[//C and //D]]`, we first remove the `X` and `Y` nodes from its syntax tree and compute the maps for the remaining syntax tree. The results are then aggregated to compute the maps for `X` and `Y`. The process is illustrated in the right part of Figure 8.

We have to compute the maps bottom-up in the syntax tree. A top-down decomposition makes indirect boolean relations among the leaf nodes direct, which leads to wrong evaluation result. An example of the top-down decomposition is illustrated in the left part of Figure 8. We can see that the OR relation between the `X` node and the`Y` node is passed on to the leaf nodes. Therefore, when we see an `X` element with only an `A` child and a `Y` element with only a `C` child, the predicate evaluates to true, which is not correct.

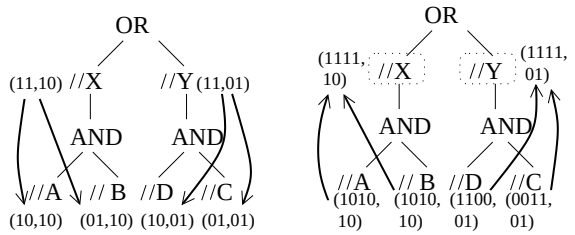
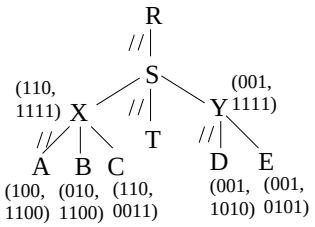


Figure 8: Top-down and Bottom-up Decomposition



//R//S[X[(//A and B) or C] and Y[D or //E]]//T

Figure 9: A Labeled Pattern Tree

4.3 Labeled pattern tree

The pattern tree depicts all the patterns specified by the query. To include the boolean relations among these patterns, we label the nodes in the pattern tree with the TRUE and FALSE maps of the nodes in the syntax tree of every predicate. Such a labeled pattern tree is uniquely defined by an XPath query and conveys the semantics of the query. If we compute the maps bottom-up for query in Figure 7 and label every node in the pattern tree with the result maps, the result labeled pattern tree is depicted in Figure 9.

The labels of the non-leaf nodes in the pattern tree are used to ensure that no *partial result* in a subquery will affect the final result. For example, for query in Figure 9, consider an `S` element that has two `X` children. The first `X` child has only one `A` child, while the second `X` element has only one `B` child. It is easy to see that this `S` element does not match the query, but it will be marked true since the `A` child of the first `X` child will set the first bit of its TRUE flag and the `B` child of the second `X` child will set the second bit. The correct operation is to clear the partial result when the first `X` element is ending since it cannot satisfy the subquery. Therefore, we have the following resetting rule:

Resetting rule: When the evaluation of the subquery P_j is finished, i.e., at the end of the element that matches the last location step in P_j, we perform the following bitmap operations:

$$(\text{tf } \textit{and } \text{tm}_j) \textit{ xor} \text{tm}_j$$
$$(\text{ff } \textit{and } \text{fm}_j) \textit{ xor } \text{fm}_j$$

If *none* of the results is 0, we perform the following bitmap operations:

$$\text{tf} \leftarrow \text{tf } \textit{and } (\textit{not } \text{tm}_j)$$
$$\text{ff} \leftarrow \text{ff } \textit{and } (\textit{not } \text{fm}_j)$$

■

Essentially the resetting rule states that all the subquery in a predicate of an element should be computed as a whole. If at the end of the element the predicate is only partially true (or false), the partial result should be blocked and not affect the future evaluation.

5 Queue with Hierarchical Index

We use a **global queue** to store all the candidates encountered in the stream. A **hierarchical index** is used to store the flags for the candidates and encodes the matching information. The basic idea is to group the candidates that always share the same flags. If we update the flags in an index node, all the candidates affected by it are processed. We have a simple fact that if two candidates have the same ancestor matched at the ith location step, they also have the same ancestor matched at the jth ($j \in [0, i-1]$) location step. Therefore, we can group them hierarchically.

The hierarchical index is a DAG (Directed Acyclic Graph) with a root node which always represents the document root. Each index node corresponds to an open element (for which we have seen the start tag but not the end tag) in the stream. It has a unique key as a quadruple (i, d, t, f), which denotes an element e at depth d that matches the ith location step and has TRUE flag t and FALSE flag f. An index node is also associated with a buffer that contains the links to the items in the queue who are descendants of e, and therefore are affected by the predicate result of e. An index node is said to be true (false) if its TRUE (FALSE) flag is marked with all 1's.

We construct the hierarchical index dynamically during runtime. When we encounter an element that matches a node test in the main trunk, we create an index node corresponding to this element. It is created as a child (in the DAG) of a parent index node that corresponds to an element that matches the previous node test in the main trunk. Since the new element may be the descendant of several elements that

match the previous location step, the structure is a DAG instead of a tree. When a candidate is encountered in the stream, it is put into the global queue. A link to the queue item is put into a buffer in the index node that corresponds to the element that generates the result. The index nodes are created using flags that are initially set to all 0's. If the matched node test does not have a predicate in the query, the initial TRUE flag is set to true.

The index nodes are transitional, which means at the end of an element, its corresponding index node will also be removed from the DAG. The links to the candidates should also be processed based on the result of the predicate, which is always determined at the end of the element it adorns. Therefore, we define two special operations to ensure that the links to the queue items are always included in the index node whose predicate they are pending on.

The first special operation on the hierarchical index is the **upload** operation. When we find that an index node is marked true, we upload the links (to the queue items) in its buffer to its nearest ancestor index node that is not marked true. The upload operation removes the set of links from the origin index node and add it to the set of links in the target index node. If all the ancestors of the index node are marked true, the queue items linked by this index node should be send to output since they have satisfied all the predicates.

A **finalize** operation is defined for every index node when it is removed from the DAG: If neither of its flags is marked with all 1's, we set all its pending subqueries to false and compute the predicate result. If the result is false, remove all the links, otherwise upload the links to its nearest pending ancestor.

Marking the index One benefit of using the hierarchical index is that every time an LFE is matched, we mark only one index node. Such a marking scheme is more efficient than storing n pairs of flags with each candidate and mark them one by one. We describes below an example of the hierarchical index and the upload operation performed on it.

Example 6 [Hierarchical index]
Consider the following query Q:

```
//store[//name="BN" or //price=10]
  //book[not(author!="john") and quantity]
  //title/text()
```

The LFEs for the query, together with the [TRUE map, FALSE map] label for each marker LFE, are listed as follows. We name the LFEs for future reference. (Note that we keep the *not* function in MLFE4.)

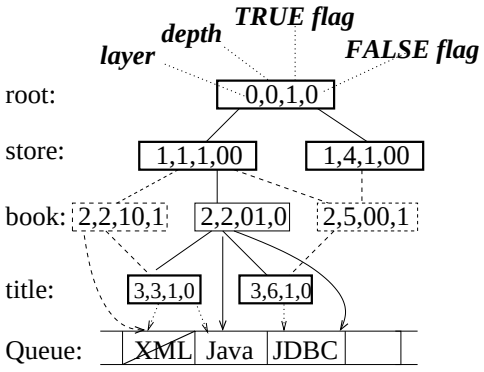


Figure 10: Example for hierarchical index

producer LFE:

`//store//book//title/text()`	(PLFE)

marker LFEs:

`//store//name="BN" [1,10]`	(MLFE1)
`//store//price=10 [1,01]`	(MLFE2)
`//store//book/not(author!="John")` [10,1]	(MLFE3)
`//store//book/quantity [01,1]`	(MLFE4)

Figure 10 illustrates a snapshot of the hierarchical index when we evaluate this query over the XML data depicted in Figure 1. The snapshot is taken when we just encounter the end event of the `book` element in line 20. In the figure, we put the tags of the elements at the left of each layer since they are shared by the index nodes in the same layer. The index nodes enclosed by bold boxes are nodes that are marked true. The index nodes enclosed by dotted boxes are nodes that are marked false. The dotted links are links that were created during the evaluation but have been removed.

Consider the `title` in line 18 whose value is "JDBC". It has three matchings with the query:

1. `store` in line 1, `book` in line 7;
2. `store` in line 1, `book` in line 14;
3. `store` in line 12, `book` in line 14;

All the matchings will be recognized by the producer LFE. Each of them corresponds to a path from the root to the candidate. Note that although the `book` in line 14 satisfy the second part of the predicate `[quantity]`, its TRUE flag is 00 instead of 01 since before we encounter the `quantity` child in line 24 (and MLFE4 is matched) , the first part of the query `not(author!="John")` has evaluated to false, and therefore the second part will never be evalu-

ated (according to the checking rule described in Section 4.2).

We note that the link between node (3,6,1,0) and candidate "JDBC" is dotted, and there is a solid link between node (2,2,01,0) and "JDBC". The dotted link is created initially when we enqueue the item "JDBC" (when PLFE is matched) and link it to the node (3,6,1,0). However, since this index node is marked TRUE, we upload the buffer item to its parent (2,2,01,0) (right after we enqueue the item). Although item (2,5,00,1) was also a parent of node (3,6,1,0), the link between them was removed when its false flag is marked as 1.

The first matching described above will satisfy the query when we encounter the end event of the book element in line 25. Since all the authors (only one here) equal to "John", the first part of the second predicate evaluates to true then. (Although the subquery `//store//book/author!="John"` evaluates to false at the time, the `not()` function will reverse the result.) All the candidates that are affected by this index node but not by any of its descendant index nodes have been uploaded to this index node. Therefore, we can output all the candidates this index node links to since it is the last pending index node in the path from the root to those candidates. ■

Multiple matchings Another benefit of the hierarchical index is that, as we shown in Example 6, given the above method to create the index, each matching between a candidate and the query is represented by a path from the root index node to the candidate. A candidate is in the result if and only if there exists a path from the root index node to it where all the index nodes in the path are marked true. Since the index node contains only a link to the candidate, a path that fails the query will only remove the link but not the candidate in the queue. We keep a reference count for every candidate so that only when the count is zero do we remove it from the queue. An output flag is associated with each candidate so that the first path that satisfy the query will set the flag to true and no further operations will be applied to this candidate. Moreover, to preserve the document order of the elements in the final result, only when the candidate has its output flag set to true and it is the head of the queue do we output the candidate.

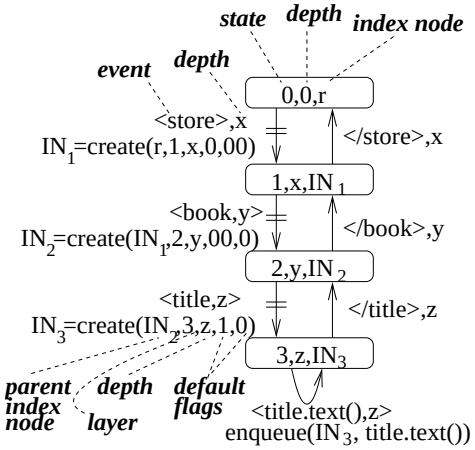


Figure 11: The producer QT

6 QT

We describe in this section how to transform an LFE into a transducer that responses to the input streams and operates on the hierarchical index and the queue that holds the candidates.

A QT is a non-deterministic finite state transducer that receives the SAX events generated from the XML streams as the input. For every SAX event, we determines for each current state whether it has an outgoing transition that accepts the event. If so, the transition takes place and the target state is added to the current state set. If there are operations defined along with the transition, the operation is executed as well.

Besides the above features that are similar to traditional FSTs, the QT is extended in the following perspectives. First, its state is dynamically computed. Second, there are different types of transitions in the automaton that accepts the inputs using different rules. Third, its operations are defined on a queue with a hierarchical index. The result of the operation may be used to compute the target state.

Section 6.1 describes the states and transitions of the QT. Then we specify how to create a QT from an LFE and assign operations to the generated QTs in Section 6.2. An example that uses the QTs to evaluate a query is shown in Section 6.4. The query Q used in Example 6 is used as a running example in this section.

6.1 Definition

Each **state** in the QT is a tuple (id, d, in), where id is a unique base identifier as in an FSA, d is the depth

of the events that lead to the current state, and *in* is the index node that is linked to this state. Only *id* is determined at compile time, while other two components are computed dynamically. A state in a QT definition may generate multiple runtime copies with same base identifiers but different depths or index nodes.

The **input** of the QT is the SAX events in the form of (g, e, d, al), where g is the name of the element that triggers the event, e is the type of the event which is one of BEGIN, END, or TEXT, d is the element's depth in the document tree (the document root has depth 0), and al is an attribute name-value list that stores the value of all the attributes (for END events, it is always empty; for TEXT events, it stores the contents of the event).

There are two types of transitions based on the depth of the input that they accept. The first is the **regular transition** that accepts either a BEGIN event whose depth is the depth of the source state plus one, or a END and TEXT event whose depth is equal to the depth of the source state. The second is the **closure transition**, which is marked with a "=" sign in the transition diagram. It accepts BEGIN events at any depth that is larger than the depth of the source state. Moreover, taking a closure transition will add the target state to the current state set, and keep the source state active.

Closure and non-closure axes are also distinguished by the depth variables in the static states. For example, a non-closure begin transition accepts the event at depth $x + 1$ if the depth variable for the source state is x, and the depth variable of the target state should be set to $x+1$ as well. For a closure transition, which could only be a begin transition, the depth of the event should be a new variable that denotes an event at any larger depth, and the target state should have the same new variable. The new variable should be larger than the depth of the source state, which we do not explicitly specify in the state transition diagram.

At run time, the initial current state set contains only the start state, $(0, 0, R_0)$ where R_0 is the root of the hierarchical index. For every incoming event, we check every current state to see whether it accepts the events: if yes, we process the event according to the transition function, including perform the state transition and execute the operations defined along with the transition; if no, the state is kept in the current state set, unlike in an FSA in which we usually report error when no current set accepts the input.

Example 7 [QT example] The state transition diagram of a QT is shown in Figure 11, which is the QT generated for the producer LFE in Example 6. We use variables for the depths and index nodes in the states to denote the relation among them. For example, the transition from state $(0, 0, R_0)$ to state $(1, x, IN_1)$ is labeled with the input `"<store>,x"`, which stands for the BEGIN event of the `store` element at any depth x, and an operation IN_1=`create(`R_0`,1,x,0,00)` (which will be explained in Section 6.2).

Suppose we use this QT to process the data in Figure 1. When we encounter the start tag of the `store` element in line 1, the QT first execute the operation, which creates and returns an index node *in* (as the child of R_0) that is bound to variable IN_1. The state transition is then taken place, which generates a new current state $(1, 1, in)$. Since the transition is a closure transition, the source state of the transition, $(0, 0, R_0)$, is kept active.

When we encounter the start tag of the `store` element at depth 4 in line 12, since $(0, 0, R_0)$ is still in the current state set, it accepts this event. After a new index node in' is created, a new state $(1, 4, in')$ is added to the current state set. As we will describe next in Section 6.2, these two index nodes are children of the root index node R_0 and will be parents of other index nodes created later, as illustrated in Figure 10. ∎

6.2 Operations

We have described the *upload* operation and the *finalize* operation on the hierarchical index in Section 5. We now define the operations that are specified explicitly in the QT: *create, enqueue, finalize*, and *mark*, while the following operations are executed automatically when certain condition is satisfied: *check, upload, clear*, and *output*.

The **create** operation generate a new index node in the hierarchical index. It is specified in the form of create(in,i,d,t,f), where *in* is the parent index node, i is the layer of the new index node, d is the depth of the event, t is the initial TRUE flag, and f is the initial FALSE flag. This operation creates and returns an index node in' with the key (i, d, t, f). The new index node in' is set as the child of *in*, which should be specified in the source state of this transition.

The **enqueue** operation is specified in the form of enqueue(in, v). It enqueues the value v into the global queue and put a link in the buffer of index node *in*.

The **finalize** operation is specified in the form of finalize(in, t, f), where t is the default TRUE flag and f is the default FALSE flag. This operation will

compare the current flags of *in* to see whether either t or f are all set in it. If yes, which means this subquery has been satisfied and no further action is required; if no, which means this subquery is either partially true or never evaluated at all, we should set this subquery to false (i.e., mark the false map f of the false flag of *in*). The finalizing operations in the LFEs that are generated from the same predicate will set every pending subquery as false and compute the result of the predicate. If the result is true, it either output or upload the buffer items that are linked from index node *in*; if the result is false, the links are removed from the index node.

The **mark** operation is specified in the form of mark(*in*, t, f, v). It will mark the TRUE flag of index node *in* using map t if v is true, or FALSE flag using map f if v is false.

The following operations that are not specified in the transition diagram are executed implicitly by the QT. Whenever the flag of an index node is changed, the **check** operation is executed. We first check whether the index node is true or false. If it is true, we use the **output** operation to emit the contents in its buffer to output if there is a path from the root to this index node in which every index node is true; otherwise we use the **upload** operation to upload the links in the buffer to the nearest ancestor node in every path that is pending. If a node is marked false, we use the **clear** operation to remove the links in the buffer, i.e., the reference count of those queue items are decreased by one. If the count is zero. the queue item is removed from the global queue.

6.3 QT Construction

For an LFE $A_1N_1 \ldots A_nN_n/O$, we generate an automaton that has $n+1$ states. Two transitions are created between state i and $i+1$: the transition from i to $i+1$ is labeled with <N_i>, which is called the *begin transition of* N_i and accepts the BEGIN events that has name N_i; the transition from $i+1$ to i is labeled with </N_i>, which is called the *end transition of* N_i and accepts the END events with name N_i.

We then assign the operations to the transitions to carry out the semantics of the query. For the producer LFE $A_1N_1 \ \ldots \ A_nN_n/O$, two types of operations are assigned to the transitions as follows:

- For every begin transition of N_i that accepts the BEGIN event of an element e (with tag N_i) at depth d, we assign to it a **create(***in*,*i*,*d*,*t*,*f***)** operation. For the corresponding end transition of N_i, if the ith location step has a predicate, we assign a **finalize(***in*, *t*, *f***)** operation.

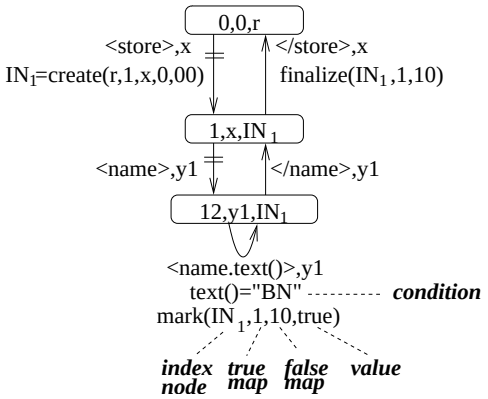


Figure 12: QT for LFE1

- For the output function O, we create the following operations:
  - If O is `text()`, we create a self-transition from state $(n+1, x, in)$ to itself, where the index node *in* should be already associated with the state $n+1$. The new transition is labeled with N_n.text(), x, which means that it accepts the text event of elements with tag N_n. The new transition is assigned with an **enqueue(***in*, N_n**.text())** operation.
  - If O is `@a` in which a is an attribute name, we assign an **enqueue(***in*, **@a)** operation to the begin transition of N_n, which means that it enqueues the content of attribute a and connect it to index node *in*. The enqueued buffer item is linked to the index node that is associated with the state n+1. This operation must be executed after the create() operation on the same transition, otherwise the index node *in* would not exist.
  - If there is no output function assigned, we create a self-transition from state n+1 to itself. The new transition is labeled with <$\bar{*}$>, which is called a **catchall** transition and accepts every event that is generated from the descendants of the element matching N_n. An **enqueue(***in*, $\bar{*}$**)** operation, which enqueues every event in its serialized form, is assigned to this new begin transition of N_n, and the end transition of N_n.

The operations in the marker LFEs are used to mark the index nodes created by the producer QT. For the marker LFEs in the form of

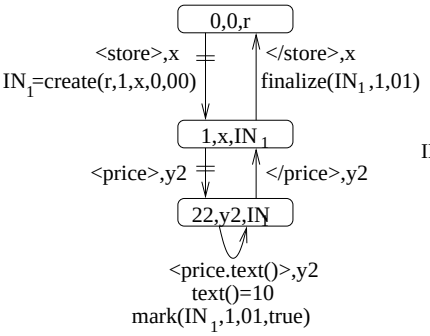


Figure 13: QT for LFE2

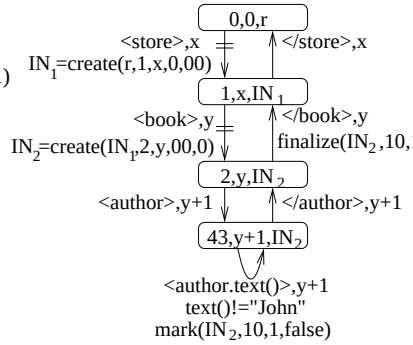


Figure 14: QT for LFE3

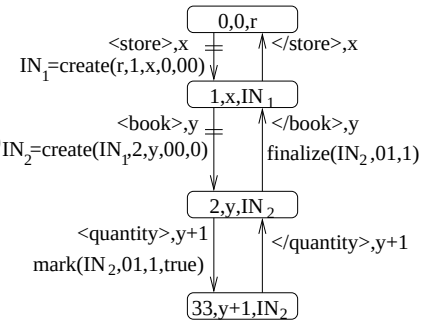


Figure 15: QT for LFE4

$A_1N_1 \ \ldots \ A_kN_Ta_{k+1}N_{k+1} \ \ldots \ A_mN_m/O$, where the first k location steps are from the main trunk and the remains are from the predicate in the ith location step. The create operations are similarly assigned to the transitions between the states that are created from the node tests in the main trunk. Note that although we duplicate these operations here, they will be performed only once since we will combine all the QTs at run time.

For transitions that are created from the node tests from the ith predicate, the assignment is as follows:

- For the test operation O in the marker LFE, we need to create a mark operation.
  - If O uses `text()`, we create a new transition from state $(m+1, d, in)$ to itself, as shown in Figure 12. The new transition is labeled with $N_m.text(), d$, which means that it accepts the TEXT event of element that has node test N_m at depth d. It is also labeled with the condition specified in O and assigned with the mark operation, which denotes that if the condition is satisfied by the event, it will perform the mark operation.
  - If O uses `@a` in which a is an attribute name, we label the begin transition of N_m with the condition specified in O and assign the mark operation to the transition.
  - If there is no output function assigned, we assign the mark operation to be begin transition of N_m, as shown in Figure 15. There will be no conditions assigned to this transition, which means that the mark operation is executed when the transition is performed.
- For every node test N_j, $j \in [k+1, n-1]$, add a **reset**(IN_k, t, f) operation on the end transition from state $j+1$ to $j.IN_k$ is the index node in state k. The t and f are the true and false maps labeled with the node test N_{j+1} in the LFE. As we described in Section 4.3, this operation resets the flags if the current subquery is neither true nor false and therefore prevent partial result affect future evaluation.

6.4 A test run for the QT

We now evaluate the query Q in Example 6, using the producer QT depicted in Figure 11 and the markers QTs depicted in Figure 12 to Figure 15. We illustrate some highlights during the evaluation in Figure 16. The hierarchical index are shown at the left of each figure, and the current state set is shown in the box at the right. We use I_i to uniquely identify an index node, whose key (l, d, t, f) is shown inside the box under I_i. In the following discussion, we use $S(id, d, I)$ to denote a state with identifier id, depth d, and index node I. When the context is clear, we also use the id to denote a state.

First consider the item "XML" in line 5. As shown in step 1 in Figure 16, it is enqueued by the transition from state 2 ($S(2, 2, I_2)$ to state 3 ($S(3, 3, I_3)$) in Figure 11. Since the index node I_3 is always true, the item is uploaded to the nearest ancestor index node I_2. Although the `price` element in line 4 is process by marker LFE in Figure 13 from state 1 to state 22, the QT does nothing since the condition `text()=10` is not satisfied.

The step 2 in Figure 16 illustrates that at the end event of the book element in line 6, the QTs returns to state 1 from state 2 and the finalize(I_2) is executed. Although both parts of the predicate in the second location step is pending, the first part is evaluated to true then since it uses the `not()` function, and we then set the TRUE flag of I_2 to "10". Since the second part `[quantity]` is evaluated to false then,

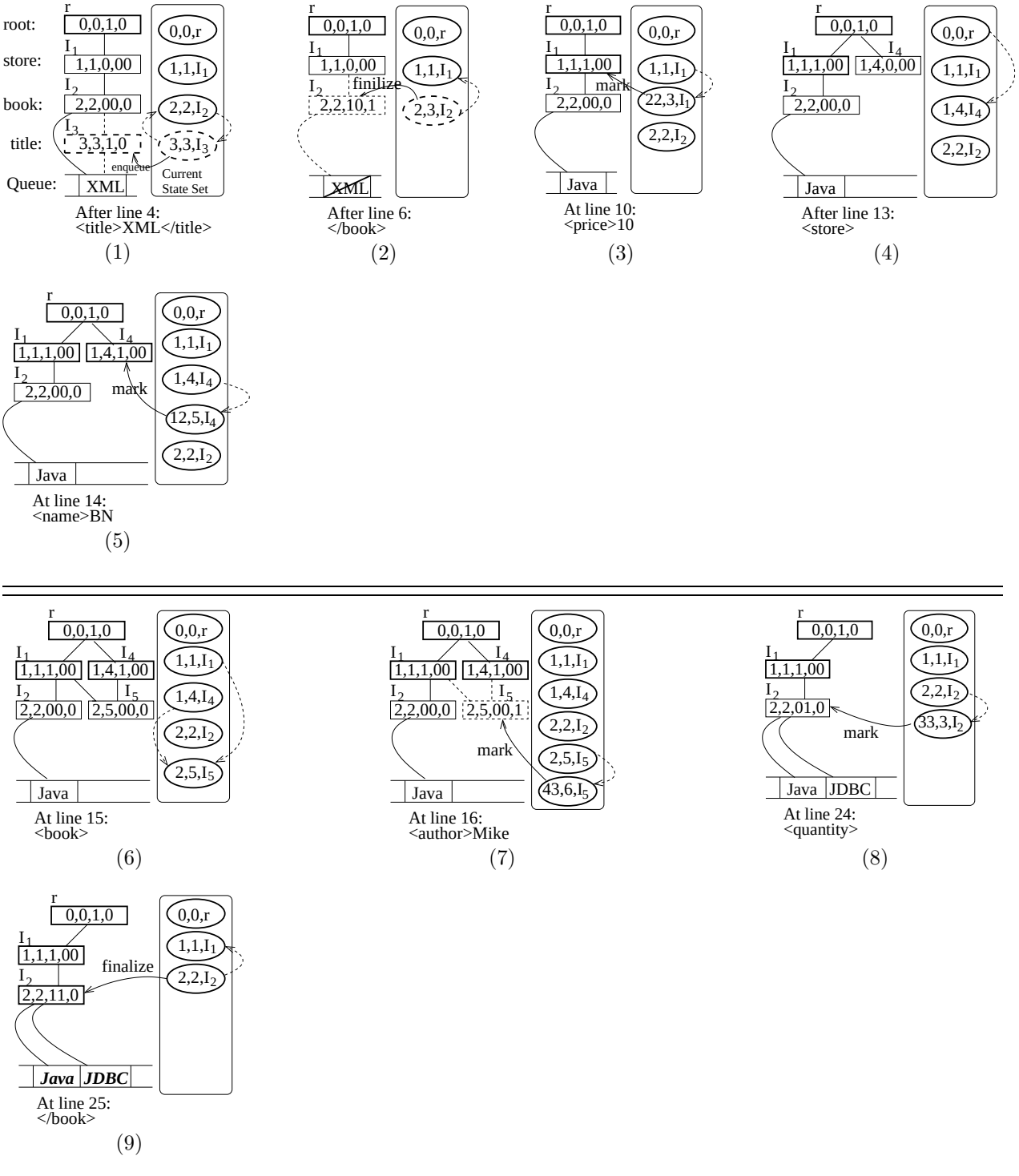


Figure 16: A test run for QT

according to the marking rule, we set its FALSE map in the FALSE flag the index node. The key of the result index node is $(2, 2, 10, 1)$, which means that I_2 is now marked false, and we remove the link to "XML" from I_2 and remove itself from its parent. Since there is no other links to this "XML" item in the queue, it is removed from the queue.

The item "Java" in line 8 is enqueued to the index node I_2 first, following a similar procedure for the item "XML". Step 3 in Figure 16 shows that when we encounter the `price` element is line 10, the self-transition from state 22 to itself is performed and the condition is satisfied. Therefore the TRUE flag of I_1 is set and the key of I_1 becomes $(1, 1, 1, 00)$.

We then encounter the `store` element in line 12. Since the state $(0, 0, r)$ is still in the current set, the transition from state 1 to state 2 is executed again: a new index node I_4 with key $(1, 4, 0, 00)$ is created and becomes the child of r; another state $(1, 4, I_4)$ is created and put in the current state set. The process is shown in step 4 of Figure 16.

When we encounter the `name` element in line 13, there are two states that accept this event: $S(1, 1, I_1)$ and $S(1, 4, I_4$. According to the checking rule, the first state will ignore this event since its index node has already been marked true. The second state will process this name element, as in the step 5 in Figure 16, and the key of I_4 is updated to $(1, 4, 1, 00)$.

As shown in step 6 in Figure 16, for the `book` element in line 14, the two states $S(1, 1, I_1)$ and $S(1, 4, I_4)$ both accept this BEGIN event, but will create the same new state associated with a new index node I_5 in the current state set: $S(2, 5, I_5)$.

When we encounter the `author` element in line 16, only the state $S(2, 5, I_5)$ will accept the begin event since the other state $S(2, 2, I_2)$ does not have a matched depth, i.e., the `author` element has to be the `book` element's direct child. As shown in step 7 in Figure 16, this `author` element satisfies the condition in marker QT in Figure 14, and the QT will mark the index node I_5. Since the last parameter of the mark operation is "false", which means that this mark operation will mark the FALSE flag of the index node with the FALSE map of the LFE, the result key of the index node I_5 is $(2, 5, 00, 1)$. Therefore, index node I_5 is removed from the hierarchical index, and so is the state $S(2, 5, I_5)$.1

The state $S(2, 2, I_2)$ will accept the `title` element in line 18 since the transition from state 2 to state 3 is a closure transition and will accept `title` elements at any depth. Step 8 in Figure 16 shows that when we encounter the `quantity` element in line 24, the first part of the TRUE flag of index node I_2 is set to 1, as defined in the QT in Figure 15.

At the end of the `book` element in line 25, the finalize operation is executed and the second part of the TRUE flag of index node I_2 is set to 1 (since we have not seen any `author` child of this `book` element that is not "John"). Since I_2 is marked true now and all the index nodes in the path from the root R_0 to this index node are true, the two items in the buffer are sent to output.

7 Related Work

Generally speaking, evaluation methods of XPath queries can be categorized into two main flavors: join-based and navigation-based [6]. The prior [16] usually evaluates all the steps separately and join the intermediate results later. It can make use of the index of data, which could be created on th fly or offline, to speed up the evaluation. The latter (such as the method we propose in this paper) usually needs to parse the data and navigate through the document tree every time it evaluates a path query. The navigation could be step-based where a location step is always evaluated on the result set of previous location step, such as the method used for XPath processing in the XSLT processor SAXON. The navigation could also be pattern-based, in which the pattern specified by the query is matched dynamically during the traversal of the document tree [7]. The traversal could be mixed as well: pattern-based traversal is used for simpler pattern or sub-pattern, while complex patterns involves backward traversal usually need to be evaluated step by step.

Many recent research on streaming XPath processing focus on filtering applications such as subscription-publishing systems [1, 12, 17]. The filtering systems use XPath expressions as *filters* and focus on grouping large number of XPath filters to share the computation on common segments. Unlike our querying system, the filtering systems does not need to handle the buffering problems since any non-empty result set will satisfy the filter expression and the whole document or document id is returned to the user. Therefore, an XPath query in a filtering system always degrades to a simpler boolean query. For example, filter expression `//R//S` is equivalent to filter expression `//R[//S]`, which returns true if one `R` element has an `S` descendant. However, in a querying system, we have to output *all* the `S` descendants of *all* the `R` elements.

The XPush machine [17] uses a bottom-up tree pattern matching algorithm to match the patterns specified by XPath queries with the XML streams.

Each subquery is evaluated bottom-up in its syntax tree, which is converted to an alternating automaton [9]. It is efficient in the filtering setting, but not easy to be applied in the querying system that needs to return the required portion of the document. Moreover, since it matches the patterns bottom-up and thus predicates are evaluated at the end of the elements, it is not straightforward to extend the method to return the result as early as possible.

In the information broker system [14] that can evaluate large number of XQuery queries by path sharing, an automaton-based method is used to match the path expression with the data and an iterator-based approach is used to evaluate the predicate and construct the result. It is a novel approach to evaluate XQuery queries on streams, but the system has to perform join operations on the intermediate results to evaluate the predicates. To avoid such join operations, we can use our method to directly return the results for the path expression with the predicates and then use an iterator-based method to construct the final result.

The *XAOS* system [3] handles reverse axes in XPath expressions in streaming environment. It uses two data structures called *X-tree* and *X-dag* to filter out the irrelevant nodes in the stream and store only the relevant nodes in a *matching structure*. At the end of the stream, the XAOS system traverses the matching structure and output the results. As far as we know, XAOS is the first streaming XPath evaluation system that handles the reverse axes. It also allows subqueries (without value comparisons and `not()` function) in the predicate. Although we can modify the algorithm to check the data structures periodically to output the results that are currently available, it is not clear how we can extend the method to guarantee the minimum-latency output and optimal buffering. Moreover, if the `not()` function is allowed in the predicate, the relevant element used inside the `not()` function may actually falsify the predicate, which means store the relevant elements may not be a good choice.

The XQuery query engine [15] uses an iterator-based approach in which each function and operator is implemented as an iterator. An iterator consumes the output streams from its input iterators and produces a single stream, which may used as the input of the other stream. XPath expressions are also implemented in the form of XPath steps using iterators. The paper does not specifically distinguish between the subqueries in the XPath predicates and the whole XPath queries.

Unlike the above streaming XML processing systems that mainly deal with the complexities caused by XML queries in a streaming environment (such as tree pattern matching in a restricted pre-order traversal of the document tree), there are several streaming systems that focus on general streaming processing. Designed for monitoring applications, the *Aurora* system [8, 10] processes data streams using a large trigger network where each trigger is essentially a dataflow graph and generated from a persistent query. The original application that issues the query obtains the result of the query from the trigger network, also in streaming form. The *Fjords* architecture [19] consists of a queuing system and a set of sensor proxies to manage multiple queries over streaming sensor data. The queues are used to route data between the operators in a query plan. The sensor proxy can adjust the sample rate of the sensor based on the queries and permit different users share data from the sensor.

These system architectures are designed to be working with operators in query plans. Instead of letting the operator retrieves data from the stream directly, they provide mechanisms (such as the trigger network in Aurora and the queuing system in Fjords) to handle the stream in an aggregated manner and provide those operators with the data they need (usually in streaming form as well). Such mechanisms are the key to support large number of queries simultaneously while keeping high throughput. These architectures are easier to be applied on iterator-based XML query processor. It is an interesting problem to study whether we can use the automaton-based approach in these architectures to process XML streams.

8 Performance Evaluation

8.1 Experimental Setup

We have implemented the methods presented in earlier sections in the XSQ system using Java (SDK 1.4.0_01) and the Xerces2 parser (2.4.0). We refer to this implementation as XSQ-S. We have released our system under the GNU GPL license at `http://www.cs.umd.edu/projects/xsq/`. We conducted our experiments on a PC-class machine with an Intel PIII 900MHz processor and 1GB of RAM running the Redhat 7.2 distribution of GNU/Linux (kernel 2.4.9). The memory limit for the Java virtual machine was set to 750MB.

Systems We compared XSQ-S with the five other systems listed in Figure 17. For each system, we used the latest version available at the time of experimentation. (At that time, we were unable to obtain the

Name	Version	Environment	Query	Subquery	Method	Parser
XSQ	2.0	streaming	XPath	yes	navigation	Xerces
XMLTK	1.01	streaming	XPath	no	navigation	xparse
Joost	20030914	streaming	STX	manually	navigation	Xerces
Saxon	6.5.2	main memory	XSLT	yes	navigation	Xerces
XALAN	2.4.0	main memory	XSLT	yes	N/A	Xerces
XXTF	2003-01-30	main memory	XPath	yes	join	Expat

Figure 17: Systems

Name	Size (MB)	Text (MB)	Elements (K)	Elements per MB	Avg depth	Max depth	Throughput (MB/s)		
							Xerces	Expat	xparse
NASA	25	15	477	19,028	5.58	8	5.59	16.8	21.5
DBLP	119	56	2,990	25,032	2.90	6	4.44	15.9	18.8
PSD	716	286	21,300	29,757	5.57	7	4.83	14.4	16.5
XMark-s	29	20	417	14,276	5.55	12	6.52	20.6	27.0
XMark-m	117	81	1,666	14,242	5.55	12	8.47	20.6	26.7
XMark-l	1172	811	16,703	14,251	5.19	12	9.23	20.3	26.2

Figure 18: Datasets

XAOS and BEA systems, discussed in Section 7.) The systems differ considerably in their goals and features. Here, we highlight only the features relevant to streaming XPath processing. **XML toolkit (XMLTK)** [2] is a set of XML tools developed at the University of Washington. The xrun program in this toolkit evaluates an XPath query using a DFA generated from the query. **Joost** is an implementation of the streaming XML transformation language STX [4], which uses XSLT-like stylesheets. XMLTK and Joost do not support queries that need buffering. XMLTK does not accept queries whose semantics may require buffering at runtime (e.g., a predicate that tests the existence of a child element). In Joost, if an item in the stream matches the pattern but some of the predicates cannot be determined by the current available data, we have to manually program the stylesheet to buffer the contents and set flags for the result of the predicate. We did not consider this manual buffering approach in our evaluation. **Saxon** and **Xalan** are two widely used high-performance XSLT processors. **XPATH from XMLTaskForce** (XXTF) is an implementation of the algorithms of Gottlob, Koch, and Pichler [16]. These systems (Saxon, Xalan, and XXTF) are non-streaming systems that need to build a DOM tree in main memory before query evaluation commences.

Datasets Some properties of the principal datasets used in our experiments are summarized in Figure 18. We used three real datasets that vary in size and other characteristics: (1) the **ADC** astronomy research dataset from NASA [5], (2) the **DBLP** bibliographic dataset [18], and PIR-International Protein Sequence Database (**PSD**) [23]. We also used three synthetic datasets, generated using the XMark benchmark [22]. The small (**Xmark-s**), medium (**Xmark-m**), and large (**Xmark-l**) datasets were generated using XMark scale factors of 0.25, 1, and 5, respectively.

Metrics We study three main metrics for streaming query processors: throughput, memory footprint, and output latency. We define the **normalized throughput** of a system as the ratio of its raw throughput (rate of input consumption) to the raw throughput its parser. We use normalized throughput instead of raw throughput in order to factor out the effect of varying parser efficiency, which is not the focus of our study. As depicted in Figure 18, the Xerces Java parser [24], used by the systems implemented in Java, is much slower than the C parsers: expat [11] used by XXTF and xparse [2] used by XMLTK. (The xparse program in XMLTK counts the number of elements in a document using the same parser as xrun. Its throughput should be close to that of the pure parser.)

We measure the **memory footprint** of each system using the *ps* program. The maximum amount of memory that each system allocated during the streaming evaluation of the entire dataset is recorded. For Java-based systems, this footprint includes the memory used by the Java virtual machine.

We also measure the **output latency** of each system, defined as the time elapsed after the system starts query evaluation and before it returns the re-

sult. We measure both **response time**, which is the elapsed time before first result element is produced, and **average latency**, which is the average of the latency of each element in the result. The response time and average latency is measured as follows. The result of every system is directed to the 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. The time we receive the first result element is the response time of the system. For each result item, we refer to this time as its latency, and the average latency of the system is the average of the latency of all result items.

Queries We used a large set of test queries in our experiments. Since the performance of the systems depends on the query, the dataset, and the relation between them, we varied the features of the tested queries for every system on every dataset (that it can handle). Figure 19 lists the organization of our experimental results and shows the varied features of the queries.

Two important features of the queries we studied are the element-selectivity and event-selectivity of a query (or a specific location step in a query). The **element-selectivity** stands for the number of elements that are used in the evaluation of a query, which could be matched by either the main trunk or a subquery of the query. The **event-selectivity** stands for the number of SAX events that the query engine has to response to evaluate the query. For example, consider the query `/sites` for the XMark dataset, where the `sites` element is the only top-level element. Its element-selectivity is 1 since it selects only one element. However, its event-selectivity is the number of SAX events generated from the dataset since a query engine has to respond to every event to construct the result. As we illustrate in this section, in general, streaming systems are more sensitive to the event-selectivity of the query while the navigation-based main-memory systems are more sensitive to the element-selectivity.

Main results We highlight here the main results of the experimental study. They are explained in more detail later.

- XSQ can evaluate XPath queries with complex subqueries on datasets as large as a few gigabytes, using almost constant amount of main-memory, and providing very high throughput (usually 50% of the throughput of a parser).
- For XPath queries with subqueries, XSQ can output the results as soon as they are available, while other systems that support such queries cannot return the result until the evaluation is finished. Therefore, even given enough memory so that we can evaluate the query using the main-memory systems, the streaming evaluation approach still has the benefit of smaller output latency.
- Navigation-based systems and join-based system are sensitive to different query features. The join-based XXTF system requires larger amount of memory for the intermediate result if the query contains value comparisons in the subquery. Main-memory navigation-based Saxon and Xalan's performances degrade when the query contains closure axes in location steps with large element-selectivity. For the streaming systems (XSQ, XMLTK, and Joost), their performance is affected mostly by the *event-selectivity* of the queries.
- The streaming systems perform worse in denser datasets that have smaller elements than in datasets that have larger elements since the prior generate fewer SAX events for data of the same size. Performance of main-memory systems is not sensitive to the density of the dataset (although they use SAX parser to parse the data).
- Shortcutting the evaluation of the subqueries can improve the performance significantly in certain queries. However, not all the systems are taking the advantage of the shortcuts.

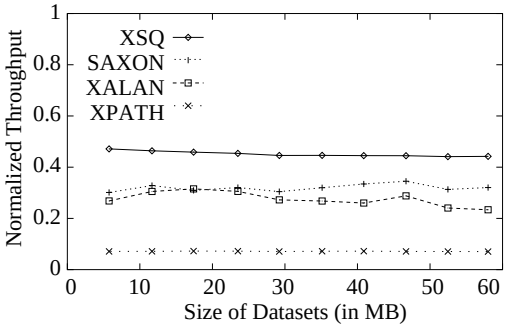
8.2 Subqueries on main-memory datasets

We first compare the performance of the systems when they evaluate XPath queries with subqueries on *main-memory datasets*, whose whole DOM tree can fit in the main memory. In our experiments, although main-memory systems (Saxon, Xalan, and XXTF) use different amount of memory to build the DOM tree, they can all process the XMark-s, XMark-m, NASA, and DBLP datasets. Since XMLTK and Joost do not support complex subqueries, we do not include them in the experiments in this section.

We first tested the scalability of the systems when applied to different sizes of datasets given enough main memory. We used XMark to generate 10 datasets with the scale factor set from 0.05 to 0.5 stepping 0.05 (sizes from 5.7MB to 58MB). **Fig-**

Section	Subquery	Datasets	Supporting Systems	Varied features
8.2	yes	NASA,XMark-s DBLP,XMark-m	XSQ,Saxon,Xalan,XXTF	selectivity, number of location steps, number of closure axes
8.3	yes	PSD ,XMark-L	XSQ	selectivity, number location steps, number of closure axes
8.4	no	NASA,XMark-s DBLP,XMark-m	XSQ,Saxon,Xalan,XXTF XMLTK, Joost	selectivity, number of location steps number of closure axes
8.5	no	PSD,XMark-L	XSQ, XMLTK, Joost	selectivity, number of location steps number of closure axes
8.6	yes	NASA	XSQ,Saxon,Xalan,XXTF	boolean operators

Figure 19: Organization of experimental study



Query: `//regions/samerica[//payment and //mailbox[//from]] //item [quantity>=2 or shipping]/name`

Figure 20: Throughputs of query on XMark datasets

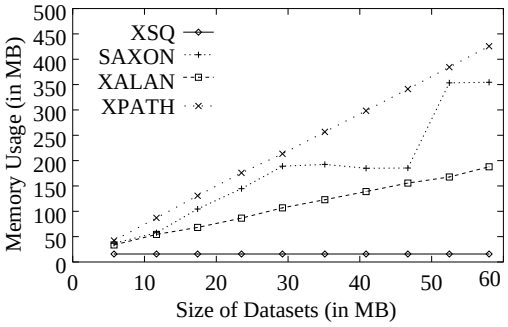
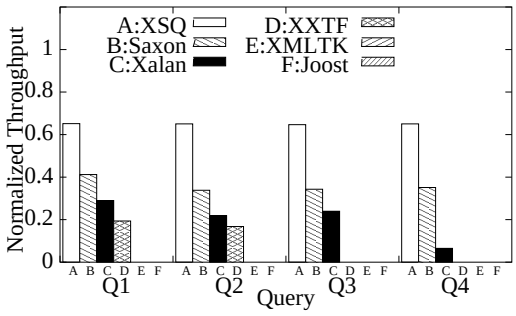
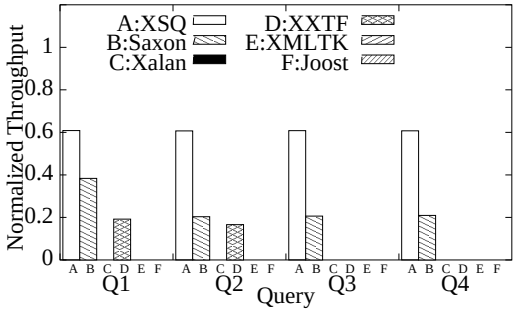


Figure 21: Memory usage of experiments in Figure 20



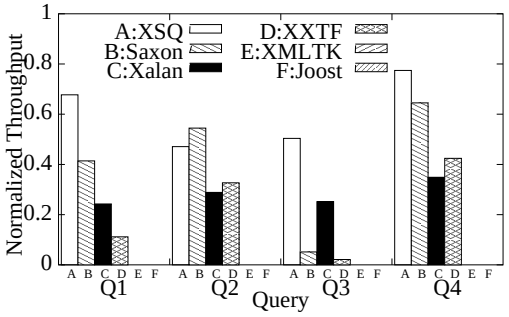

```
Q1:/site/regions/samerica[//payment and
//mailbox[//mail//from]]/item[quantity or shipping]/name
Q2://site//regions//samerica[//payment and
//mailbox[//mail//from]]//item[quantity or shipping]//name
Q3://regions//samerica[//payment="Creditcard" and
//mailbox[//mail//from]]//item[quantity=1 or shipping]//name
Q4://regions//samerica//item[quantity=1 or shipping and
//payment="Creditcard" and //mailbox[//mail//from]]//name
```

Figure 22: Complex queries on XMark-s dataset



Queries are the same as in Figure 22.

Figure 23: Complex queries on XMark-m dataset



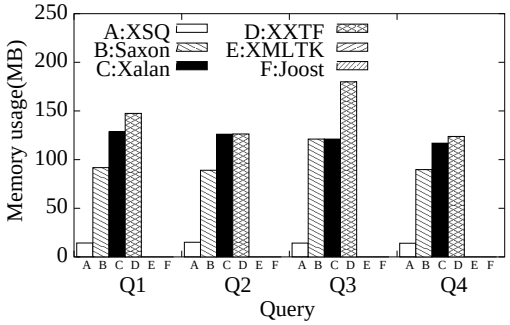


Figure 25: Memory usage for experiments in Figure 24

```
Q1://dataset[//initial and //date[year>=1950]]
//reference[source//publisher]//name
Q2://dataset[not(altname)]//reference[source//publisher]//name
Q3://dataset[reference[source[journal[author[initial="B"]]]]]//name
Q4:/datasets[not(dataset)]//name
```

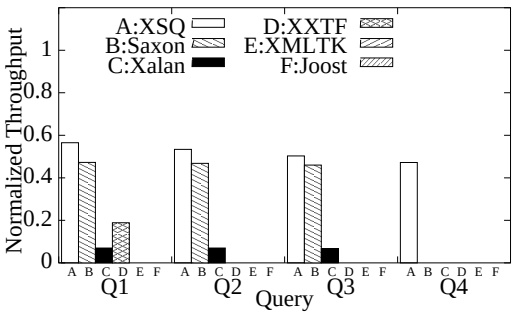
Figure 24: Complex queries on NASA dataset

ure 20 depicts the normalized throughput of the systems when querying the datasets. **Figure 21** depicts the maximum memory usage during the evaluation. The linear memory usage for the three main-memory systems, who need to load all the data into main memory, is as expected. It is not clear why Saxon uses almost the same amount of memory for datasets from 29MB to 47MB, and then uses around 150MB more memory for 5MB size increase from 47MB to 52MB. Since XSQ does not need store all the data in main memory, its memory usage is almost constant. From Figure 20 we can see the normalized throughputs of Saxon, XSQ, and XXTF are almost constants, while Xalan's performance degrades when the size of the dataset increases. We also tested (not included here) larger XMark datasets generated using larger scale factors. Xalan fails to evaluate the query in our experiments when the factor becomes larger than 0.9. For dataset with scale factor 0.8, it evaluated the query in around 65 seconds; with factor 0.9, it ran for more than three hours and we terminated the process. The reason why Xalan did not scale up in our experiment is not clear, but should not be insufficient memory since the memory usage was stable at around 360MB during the three hours' running.

We also tested other complex queries on the XMark datasets. For the same set of queries, **Figure 22** and **Figure 23** depict the throughput of the systems on XMark-s(29MB) and XMark-m(117MB) datasets. We chose the queries so that `Q1` contains only child axes in the main trunk while `Q2` replaces all the child axes with descendant-or-self axes. `Q3` has similar structure as `Q2` but has two value comparisons in two of the subqueries. `Q4` is transformed from `Q3` by move two subqueries from the second location step to the third location step. The two moved subqueries in `Q4` need to be evaluated for every `item` elements insides the `samerica` element. In contrast, they need to be evaluated once in `Q3` for the only `samerica` element in the document. All the predicates evaluate to true for these queries, and the result sets are the same. The normalized throughput of XSQ and Saxon are almost the same for both datasets. XSQ has almost the same throughput for all the four queries since they involve almost the same set of elements. As a support for our previous speculation, Xalan can answer the queries over the XMark-s dataset but not over the XMark-m dataset.

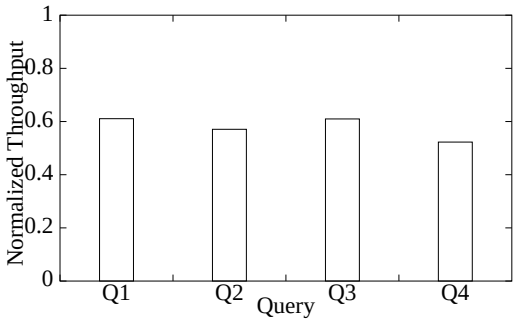
Performance of navigation-based systems is not affected as much as the join-based systems by the comparisons required by subqueries. Since they compare the values during the traversal of the data, either in main memory or streaming form, usually the intermediate result set is small. **Figure 25** depicts the memory usage for the queries in **Figure 24**. XXTF is slower and uses more memory in query `Q1` and `Q3` than in `Q2` and `Q4` since evaluating `Q1` and `Q3` involves value comparisons. Moreover, evaluating `Q1` requires less memory and is faster than `Q3`. In the NASA dataset, there are 14,512 `initial` elements whose value are tested by `Q3`, while there are only 5,935 `year` elements whose value is tested by `Q1`.

We also tested complex queries on the DBLP dataset. The results are illustrated in **Figure 26**. The previous conclusions are supported by this set of experiments as well.




```
Q1:  /dblp[inproceedings]/article/title
Q2:  /dblp[inproceedings[url and booktitle="B"]]
/article/title
Q3:  /dblp[inproceedings[url and booktitle="B"] and
phdthesis[school and year=1984]]/article/title
Q4:  /dblp[inproceedings[url and booktitle="B"]]
//article[journal and year=1989]//title
```

Figure 26: Complex queries on DBLP dataset




```
Q1://regions/samerica[//payment and
//mailbox[//from]]//item[quantity>=2 or shipping]/name
Q2:/site[regions[africa or //mail[date]]]//item[//shipping
and not(//quantity=2)]//name
Q3://regions[africa]/asia[//location]//item[//shipping and
not(//quantity=2)]//name
Q4:/site[//seller and //price>10 and //item[//shipping and
not(//quantity=2)]]//name
```

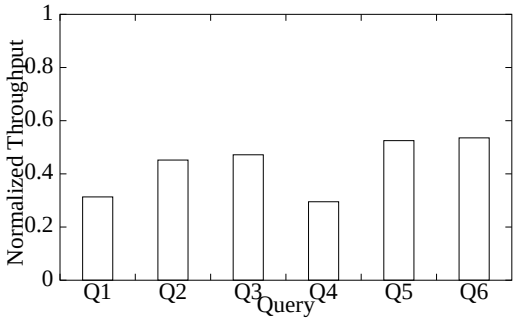
Figure 27: Complex queries on XMark-l dataset

8.3 Subqueries on large datasets

We also tested XSQ for complex queries on large datasets. Since the systems that support such queries (XXTF, Saxon, and Xalan) need to build the entire DOM tree in the main memory, we cannot test them using our current setting. Therefore, we only used the XSQ system to test complex queries over the PSD dataset (716MB) and the XMark-l dataset(1172MB). The results are illustrated in **Figure 27** and **Figure 28**.

For queries on the XMark-l dataset, we used several queries different than the queries in Figure 22 and 23. Those queries lead to similar results as illustrated in those two figures. In this experiment we varied the structure of the queries, e.g., `Q1` and `Q3` have four location steps and three of them have predicates, two predicates in `Q2` use all the boolean operators and the first predicate is deeply nested, and `Q4` has only two location steps and the first location step has a very complex predicate. For all these complex queries, XSQ achieves consistent high throughput.

XSQ's performance is affected by *event-selectivity* of the query, i.e., how many SAX events XSQ has to response in order to evaluate the query. For example, in the PSD dataset, each of the first three queries selects only a small amount of elements as the result. The last three queries have similar structures as the previous three but with much larger result set, since they do not specify value comparisons in the predicates in the subqueries. The result sizes (in bytes)




```
Q1:  //reference[//refinfo[year=1981 and citation]]//author
Q2:  //ProteinEntry[organism/source="human" and
header/accession="A31764"]//protein//name
Q3:  //ProteinDatabase[//reference[refinfo[//volume=238 and
year=1963]]]//ProteinEntry[header/uid="CCCZ"]//genetics
Q4:  //reference[//refinfo[year and citation]]//author
Q5:  //ProteinEntry[organism/source and
header/accession]//protein//name
Q6:  //ProteinDatabase[//reference[refinfo[volume and
year]]]//ProteinEntry[header/uid]//genetics
```

Figure 28: Complex queries on PSD dataset

are listed below.

Q1:	98,229	Q4:	161,058,001
Q2:	37	Q5:	12,936,858
Q3:	44	Q6:	16,106,474

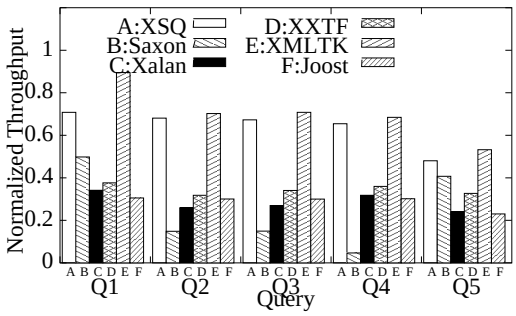
As Figure 28 illustrates, the throughput of XSQ is not affected by the result size as much as one may expected. One reason is that the first three queries need to test the value comparisons against the incoming data, while the last three do not need to perform such value comparisons. When evaluating `Q2` and `Q3`, since XSQ has to perform string comparison for all the `ProteinEntry` elements, the throughputs are even slightly slower than those of `Q5` and `Q6` who have larger result set.

The performance of XSQ is affected by the selectively of the query, not the result size. XSQ is significantly slower when evaluating `Q1` and `Q4` than the other queries because there are 314,763 `reference` element and 5,983,050 `author` subelements in the dataset. In contrast, there are only 262,525 `ProteinEntry` elements and each of them has only one `name` and at most one `genetics` subelement.

8.4 Simple queries on main-memory datasets

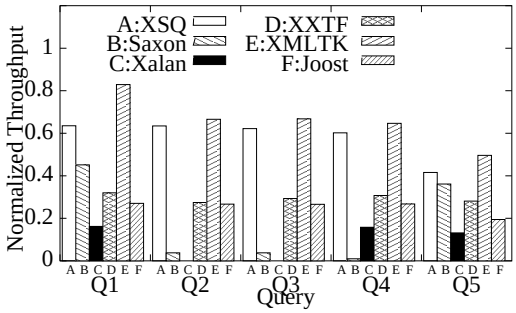
For simple queries without predicates, the performance of different system is influenced by the features of the query differently. Streaming systems get lower throughput for queries with higher selectivities. Saxon gets lower throughput for queries with closure axes on location steps that are matched with large number of elements. Surprisingly, the join-based XXTF is not affected significantly by the number of location steps and the number of closure axes.

We first tested a set of queries with different number of location steps on XMark-s. The results are depicted in **Figure 29**. In general, the streaming systems are not affected much by the number of location steps. Their performance, however, is affected by the event-selectivity of the query. Main-memory systems are not sensitive to the event-selectivity of the query. Since they load the whole document tree into the main memory, the difference between the evaluation cost of a processed element and the cost of an unprocessed element is not as significant as in the streaming system. In this set of experiments, Xalan and XXTF's throughputs are similar for all the five queries. Saxon's performance, however, degrades when the query contains closures axes in the location steps of large element-selectivity. For example, it performs better in query `Q1` and `Q5` than the other three




```
Q1:/site/regions/samerica/item/name
Q2://regions//item//name
Q3://regions//name
Q4://name
Q5://regions
```

Figure 29: Simple queries on XMark-s datasets



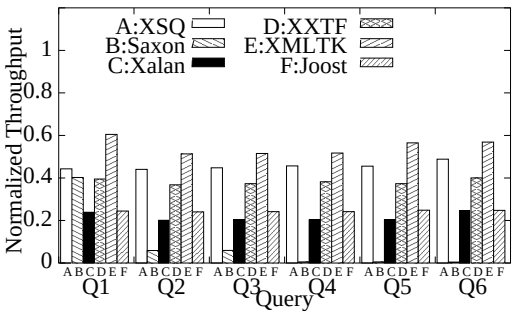
Queries are the same as in Figure 29.

Figure 30: Simple queries on XMark-m datasets

queries since there are only a few `regions` elements but thousands of `name` elements in the dataset.

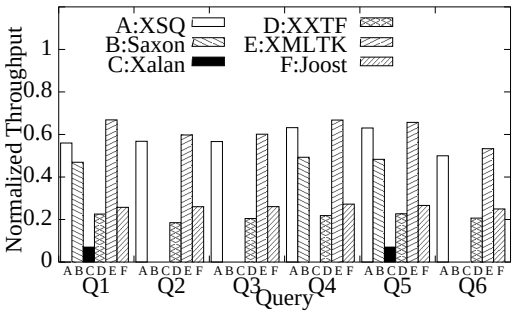
We tested the same queries in Figure 29 on the XMark-m dataset. The results are illustrated in **Figure 30**, which are similar to those illustrated in Figure 29. We note that since the dataset is 117MB, Xalan fails to evaluate the query `Q2` and `Q3`. However, it can evaluate query `Q1`, `Q4`, and `Q5`, who either has no closure axes or has only one location step. If we compare the queries used in Figure 30 with the queries in Figure 23 and Figure 26, it seems that Xalan's method to handle multiple closure axes cannot scale up to large datasets.

Simple queries with different number of location steps and closure axes are also tested in the NASA dataset and DBLP dataset. The results are depicted in **Figure 31** and **Figure 32**. Our previous conclu-



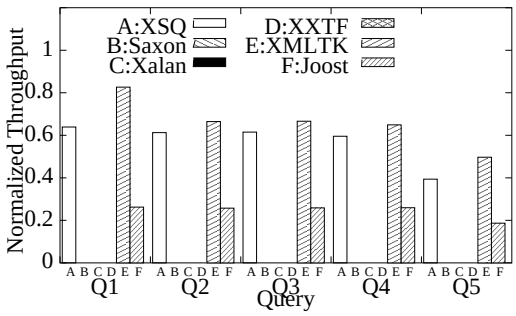

```
Q1:/datasets/dataset/tableHead/fields/field/name
Q2:/datasets//tableHead/fields/field/name
Q3:/datasets//fields/field/name
Q4:/datasets//field/name
Q5://field//name
Q6://name
```

Figure 31: Simple queries on NASA datasets




```
Q1:/dblp/article/title
Q2://dblp//article//title
Q3://article//title
Q4://phdthesis//title
Q5://phdthesis
Q6://title
```

Figure 32: Simple queries on DBLP dataset



Queries are the same as in Figure 29.

Figure 33: Simple queries on XMark-l datasets

sions, e.g., XXTF and the streaming systems are not sensitive to the number of location steps, are supported in these Figures.

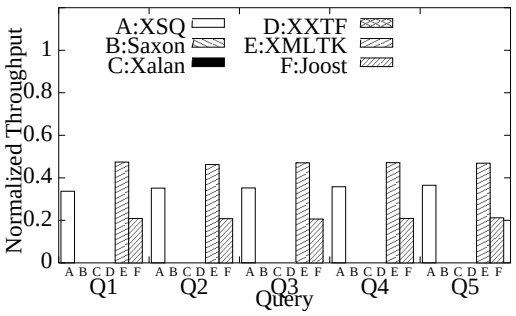
We note here that the throughputs of the streaming systems on the NASA dataset and DBLP dataset are smaller than the throughputs on the XMark datasets. As we illustrated in Figure 18, the NASA data and DBLP dataset have more elements per megabytes of data than the XMark dataset, i.e., the datasets are *denser*. Denser datasets generate more SAX events and lead to longer parsing time for the SAX parsers, as illustrated in Figure 18. Therefore, streaming systems need to process more events in a denser dataset, and the throughput will be smaller in the denser datasets than in the XMark datasets.

8.5 Simple Queries on Large Datasets

We also tested simple queries for the XMark-l and the PSD dataset. Since the main-memory systems cannot process these large datasets due to the memory limit, we only tested the streaming systems on these datasets.

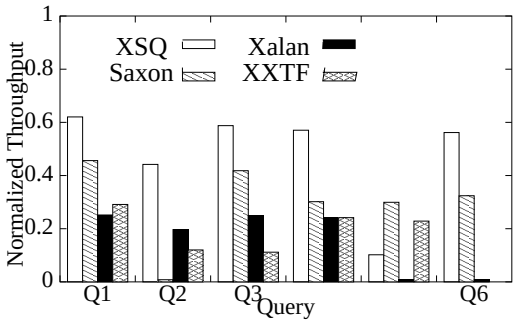
For the same set of queries in Figure 29, we apply them on the XMark-l dataset of size 1,172MB, which is generated using XMark with scale factor set to 10. The results, which are depicted in **Figure 33**, illustrate similar pattern as Figure 29 and Figure 30. For example, XMLTK performs best for queries without closure axes.

We also varied the number of location steps and number of closure axes for queries over the PSD dataset. The result are depicted in Figure 34. It also illustrates that these streaming systems are not sensitive to the number of location steps and closure axes. Moreover, since PSD is the densest dataset among the datasets we used (its number of elements




```
Q1:/ProteinDatabase/ProteinEntry/reference/refinfo
/authors/author
Q2://ProteinDatabase//ProteinEntry//reference//refinfo
//authors//author
Q3://ProteinDatabase//reference//refinfo//author
Q4://ProteinEntry//author
Q5://author
```

Figure 34: Simple queries for PSD dataset




```
Q1://dataset[//initial]//reference[source//publisher]//name
Q2://dataset[//initial and //date[year>=1950]]//name
Q3://dataset[//initial and //date[year>=1950]]//reference[source
//publisher]//name
Q4://dataset[//initial and //date[year]]//reference[source
//publisher and //related]//name
Q5://dataset[//initial and //date[year]]//reference[source
//publisher and not(//related//related)]//name
Q6://dataset[//initial and //date[year>=1950]]//reference[source
//publisher and //related//related]//name
```

Figure 35: Queries with AND operators

per megabytes is almost twice as many as the number for the XMark datasets), the throughput of the streaming systems are smaller than the throughputs for other datasets for the queries have similar structures.

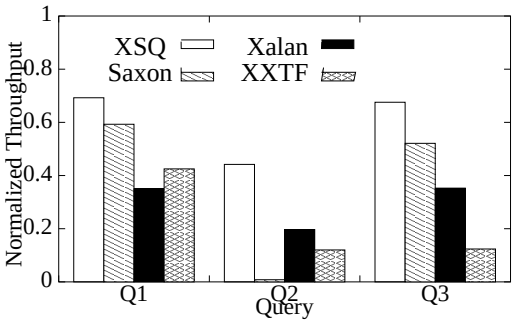
8.6 Processing boolean operators

We use the next set of experiments to explore how different boolean operators used in the predicates affect the performance of the systems. This set of queries is executed on the NASA dataset, which is small (25MB) so that the main-memory system can evaluate most queries over it. It is also denser than the XMark-s dataset so that XSQ is not benefited from less SAX events.

Figure 35 illustrates the normalized throughputs of the systems when they evaluate queries with *AND* operators. All the subqueries evaluate to true except `//related//related` in `Q6`. Navigation-based systems seem to be insensitive to the number of subqueries in the predicate, since they need to traverse the document tree once no matter the predicate contains how many subqueries. For example, XSQ performs almost the same for the `Q1` and `Q6` although `Q6` has two more subqueries. However, XSQ's performance degrades in `Q5` since the `not(//related//related)` subquery can evaluate to true only at the end of every `reference` element. Therefore, XSQ has to buffer every `name` element and output them at that time, which is the worst-case scenario for a streaming system that buffers the candidates. It is expected that the main-memory systems should not be sensitive to the number of queries connected by the `and` operators. Saxon performs better in the other queries than in `Q2` since there are 9,788 `name` elements in the single `datasets` element. with total size 1,349KB (while there are 2,667 `name` elements in the `reference` element with total size 98KB). However, it is not clear why Xalan's performance degrades significantly for `Q5` and `Q6`.

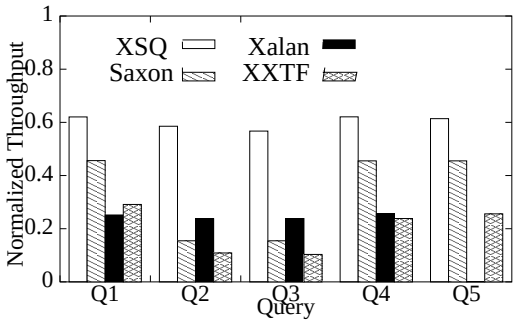
The join-based system XXTF is sensitive to the number of subqueries, since it needs to generate more intermediate result. However, for subqueries that does not involves value comparisons, the intermediate result set is small (and does not affect the performance as much as the subqueries that generates larger intermediate result set), since we only need to record the existence of the tested element. For example, although `Q4` and `Q5` contain more subqueries than `Q1`, XXTF's throughputs for them is similar to the throughput for `Q1`. However, XXTF performs better in `Q1` than in `Q2` and `Q3` since the latter two contain value comparisons. (XXTF reports runtime error for `Q6`.)

Figure 36 illustrates the throughput of the systems when they evaluate queries with *NOT* operators. Since predicates with the NOT operator can be falsified before the end of the elements, we ex-




```
Q1:/datasets[not(dataset)]//name
Q2:/datasets[//initial and //date[year>=1950]]//name
Q3:/datasets[//initial and //date[year>=1950] and
not(dataset)]//name
```

Figure 36: Queries with NOT functions




```
Q1://dataset[//initial]//reference[source//publisher]//name
Q2://dataset[//initial or //date[year>=1950]]//
reference[source//publisher or //related]//name
Q3://dataset[//initial or //date[year>=1950] or
//altname]//reference[source//publisher or //related or
//author]//name
Q4://dataset[//initial or dummy]//reference[source//publisher]//name
Q5://dataset[//initial]//reference[source//publisher or
//related//related]//name
```

Figure 37: Queries with OR operators

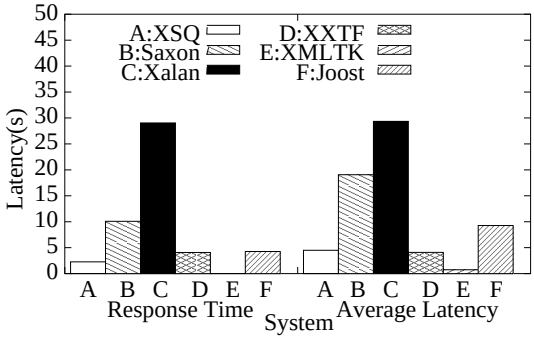
pected that some predicates that can be decided earlier will affect the performance. In the test queries, since the `datasets` element has only `dataset` children, the subquery `not(dataset)` should be falsified very early. Navigation-based system can take advantage of shortcutting the evaluation while the join-based system seems cannot. For example, the throughput of Saxon in `Q2` is very small (as explained in the results of Figure 35), but its throughput increases significantly in `Q3` where the additional subquery `not(dataset)` can be falsified early in the traversal of the document tree. XSQ and Xalan also take this shortcut and evaluate `Q3` faster than `Q2`. XXTF, however, does not benefit from this fact and performs almost the same for `Q2` and `Q3`.

The next set of queries are used to illustrate the performance of the systems when they evaluate queries with *OR* operators. The results are depicted in **Figure 37**. In the queries, all the subqueries evaluate to true except for `dummy` in `Q4`, which is a node test that never appears in the dataset, and `//related//related` in `Q5`. XSQ performs almost the same for all five queries, which are as expected, since XSQ stops evaluating these predicate as soon as one of the subqueries is true. It is not clear why XALAN performs the same for the first four while degrades significantly for `Q5`, which is unusual since the subquery `source//publisher` should be evaluated to true very early. It is also not clear why Saxon performs better in `Q1`, `Q4`, and `Q5` than in `Q2` and `Q3`. Although XXTF also performs better in `Q1`, `Q4`, and `Q5`, we believe the value comparisons in `Q2` and `Q3` degrades the performance of the system.

In general, shortcutting in predicate evaluation can improve the performance of the system. However, it is not straightforward in XPath evaluation due to the hierarchical structure of the XPath queries. It is clear that not all systems are taking it into account.

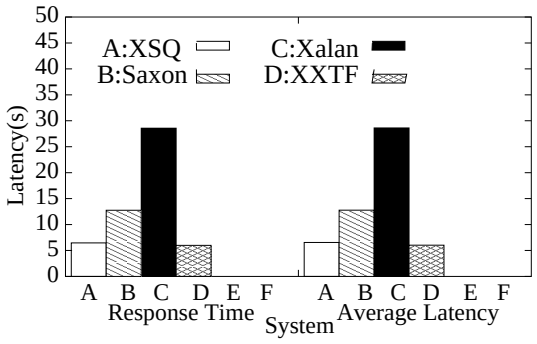
8.7 Output Latency

We also tested the output latency for the systems on the NASA dataset. The results are illustrated in **Figure 38** and **Figure 39**. The left part of each figure illustrates the time every system returns the first result item, and the right part illustrates the average time every system uses to return a result element. (The time to load the JVM is included in




```
Q://dataset//reference//name
```

Figure 38: Output Latency



`Q://dataset[//initial]//reference[source//publisher]//name`

Figure 39: Output Latency

both metrics for the Java-based systems.) The query used in Figure 38 has no predicate. The query used in Figure 39 has two predicates, and we did not test the XMLTK and Joost systems in the second experiments.

As we would expect, streaming systems usually have smaller response time and average output latency than the main-memory systems. Note that in extreme cases (e.g., set a predicate that cannot be evaluated until the end of the data), the streaming system, like XSQ, has to wait until the end to output result. For main-memory systems, the response time is usually close to the average latency since they usually output all the result at the same time after the evaluation is finished.

9 Conclusion

We addressed the problem of subquery evaluation in XPath. We focused on streaming data, but our methods also benefit non-streaming environments because of the advantages of sequential access. Subquery evaluation in a streaming environment is a challenging problem if one is to avoid naive approaches that lead to excessive buffering and low throughput. We defined three desirable properties for a streaming XPath processor: optimal buffering, minimum-latency output, and parsimonious predicate evaluation. These properties further complicate the query evaluation task.

Our solution is organized around a pushdown transducer that is augmented with a queue. Although such a transducer is not of theoretical interest (since it is equivalent to a Turing machine), this design results in a clean design that permits an efficient implementation. Our method supports features such as nested subqueries with complex predicates, and are fully implemented in the XSQ stream processor, which is publicly available. To our knowledge, it is the only streaming XPath engine that supports subqueries.

We characterized the performance of our method using a detailed experimental study. Although there are very few systems that are directly comparable to XSQ in their goals and features (queries instead of filters, streaming data, complex queries, etc.), we presented experimental results for several related systems (many non-streaming) to illustrate the costs of supporting various XPath features.

In continuing work, we are investigating streaming evaluation for more complex features (e.g., reverse axes) and query languages (e.g., XQuery). We are also working on combined processing of a large number of XPath queries. We believe that some of the work on Information Dissemination systems (e.g., [13]) may be combined with our method for this purpose.

References

[1] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 53–64, Cairo, Egypt, September 2000.

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

[3] C. M. Barton, P. G. Charles, D. Goyal, M. Raghavachari, V. Josifovski, and M. F. Fontoura. Streaming XPath Processing with Forward and Backward Axes. In *Proceedings of the International Conference on Data Engineering*, pages 455–466, Bangalore, India, March 2003.

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

[5] K. D. Borne. ADC Dataset, GSFC/NASA XML Project, 2002. `http://xml.gsfc.nasa.gov/archive/`.

[6] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based XML multi-query processing. In *Proceedings of the International Conference on Data Engineering*, March 2003.

[7] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the ACM SIGMOD International Con-*

ference on Management of Data (SIGMOD), pages 310–321, June 2002.

[8] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 215–226, Hong Kong, China, August 2002.

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

[10] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *The First Biennial Conference on Innovative Database Systems*, Asilomar,California, January 2003.

[11] J. Clark. XML parser toolkit, 2000. `http://www.jclark.com/xml/expat.html`.

[12] Y. Diao, P. Fischer, and M. J. Franklin. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proceedings of the International Conference on Data Engineering*, pages 341–344, San Jose, California, February 2002.

[13] Y. Diao and M. J. Franklin. High-performance XML filtering: An overview of YFilter. *IEEE Data Engineering Bulletin*, 26, 2003.

[14] Y. Diao and M. J. Franklin. Query processing for High-Volume XML message brokering. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 261–272, September 2003.

[15] D. Florescu, C. Hillary, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL Streaming XQuery Processor. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Berlin, Germany, August 2003.

[16] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, Aug. 2002.

[17] A. K. Gupta and D. Suciu. Streaming Processing of XPath Queries with Predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 419–430, San Diego, California, June 2003.

[18] M. Ley. Computer Science Bibliography, 2003. `http://dblp.uni-trier.de/xml/`.

[19] S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *Proceedings of the International Conference on Data Engineering*, pages 555–566, San Jose, California, February 2002.

[20] D. Megginson et al. Simple API for XML, 2002. `http://www.saxproject.org/`.

[21] F. Peng and S. S. Chawathe. XPath Queries on Streaming Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 431–442, San Diego, California, June 2003.

[22] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 974–985, Hong Kong, China, August 2002.

[23] C. H. Wu, H. Huang, L. Arminski, et al. The Protein Information Resource: an integrated public resource of functional annotation of proteins, 2002. Nucleic Acids Ressearch 30,35-37.

[24] The Xerces Java Parser Readme. `http://xml.apache.org/`, 2000.